Contents

Introduction to ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

By Daniel Roth, Rick Anderson, and Shaun Luttin

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps. With ASP.NET Core, you can:

- Build web apps and services, Internet of Things (IoT) apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on .NET Core.

Why choose ASP.NET Core?

Millions of developers use or have used ASP.NET 4.x to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, including architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
  - Kestrel
  - IIS
  - HTTP.sys
  - Nginx
  - Apache
  - Docker
- Side-by-side versioning.
- Tooling that simplifies modern web development.

Build web APIs and web UI using ASP.NET Core MVC

ASP.NET Core MVC provides features to build web APIs and web apps:

- The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
- Razor Pages is a page-based programming model that makes building web UI easier and more productive.

- Razor markup provides a productive syntax for Razor Pages and MVC views.
- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- Model binding automatically maps data from HTTP requests to action method parameters.
- Model validation automatically performs client-side and server-side validation.

Client-side development

ASP.NET Core integrates seamlessly with popular client-side frameworks and libraries, including Blazor, Angular, React, and Bootstrap. For more information, see Introduction to ASP.NET Core Blazor and related topics under *Client-side development*.

ASP.NET Core target frameworks

ASP.NET Core 3.x and later can only target .NET Core. Generally, ASP.NET Core is composed of .NET Standard libraries. Libraries written with .NET Standard 2.0 run on any .NET platform that implements .NET Standard 2.0.

There are several advantages to targeting .NET Core, and these advantages increase with each release. Some advantages of .NET Core over .NET Framework include:

- Cross-platform. Runs on Windows, macOS, and Linux.
- Improved performance
- Side-by-side versioning
- New APIs
- Open source

Recommended learning path

We recommend the following sequence of tutorials for an introduction to developing ASP.NET Core apps:

1. Follow a tutorial for the app type you want to develop or maintain.

APP TYPE	SCENARIO	TUTORIAL
Web app	New server-side web UI development	Get started with Razor Pages
Web app	Maintaining an MVC app	Get started with MVC
Web app	Client-side web UI development	Get started with Blazor
Web API	RESTful HTTP services	Create a web API†
Remote Procedure Call app	Contract-first services using Protocol Buffers	Get started with a gRPC service
Real-time app	Bidirectional communication between servers and connected clients	Get started with SignalR

2. Follow a tutorial that shows how to do basic data access.

SCENARIO	TUTORIAL
New development	Razor Pages with Entity Framework Core
Maintaining an MVC app	MVC with Entity Framework Core

3. Read an overview of ASP.NET Core fundamentals that apply to all app types.

4. Browse the table of contents for other topics of interest.

†There's also an interactive web API tutorial. No local installation of development tools is required. The code runs in an Azure Cloud Shell in your browser, and curl is used for testing.

Migrate from .NET Framework

For a reference guide to migrating ASP.NET 4.x apps to ASP.NET Core, see Migrate from ASP.NET to ASP.NET Core.

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps. With ASP.NET Core, you can:

- Build web apps and services, Internet of Things (IoT) apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on .NET Core or .NET Framework.

Why choose ASP.NET Core?

Millions of developers use or have used ASP.NET 4.x to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, with architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
  - Kestrel
  - IIS
  - HTTP.sys
  - Nginx
  - Apache
  - Docker
- Side-by-side versioning.

- Tooling that simplifies modern web development.

Build web APIs and web UI using ASP.NET Core MVC

ASP.NET Core MVC provides features to build web APIs and web apps:

- The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
- Razor Pages is a page-based programming model that makes building web UI easier and more productive.
- Razor markup provides a productive syntax for Razor Pages and MVC views.
- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- Model binding automatically maps data from HTTP requests to action method parameters.
- Model validation automatically performs client-side and server-side validation.

Client-side development

ASP.NET Core integrates seamlessly with popular client-side frameworks and libraries, including Blazor, Angular, React, and Bootstrap. For more information, see Introduction to ASP.NET Core Blazor and related topics under *Client-side development*.

ASP.NET Core targeting .NET Framework

ASP.NET Core 2.x can target .NET Core or .NET Framework. ASP.NET Core apps targeting .NET Framework aren't cross-platform—they run on Windows only. Generally, ASP.NET Core 2.x is made up of .NET Standard libraries. Libraries written with .NET Standard 2.0 run on any .NET platform that implements .NET Standard 2.0.

ASP.NET Core 2.x is supported on .NET Framework versions that implement .NET Standard 2.0:

- .NET Framework latest version is recommended.
- .NET Framework 4.6.1 and later.

ASP.NET Core 3.0 and later will only run on .NET Core. For more details regarding this change, see A first look at changes coming in ASP.NET Core 3.0.

There are several advantages to targeting .NET Core, and these advantages increase with each release. Some advantages of .NET Core over .NET Framework include:

- Cross-platform. Runs on macOS, Linux, and Windows.
- Improved performance
- Side-by-side versioning
- New APIs
- Open source

To help close the API gap from .NET Framework to .NET Core, the Windows Compatibility Pack made thousands of Windows-only APIs available in .NET Core. These APIs weren't available in .NET Core 1.x.

Recommended learning path

We recommend the following sequence of tutorials and articles for an introduction to developing ASP.NET Core apps:

1. Follow a tutorial for the type of app you want to develop or maintain.

APP TYPE	SCENARIO	TUTORIAL
Web app	For new development	Get started with Razor Pages
Web app	For maintaining an MVC app	Get started with MVC
Web API		Create a web API†
Real-time app		Get started with SignalR

2. Follow a tutorial that shows how to do basic data access.

SCENARIO	TUTORIAL
For new development	Razor Pages with Entity Framework Core
For maintaining an MVC app	MVC with Entity Framework Core

3. Read an overview of ASP.NET Core fundamentals that apply to all app types.

4. Browse the Table of Contents for other topics of interest.

†There's also a web API tutorial that you follow entirely in the browser, no local IDE installation required. The code runs in an Azure Cloud Shell, and curl is used for testing.

Migrate from .NET Framework

For a reference guide to migrating ASP.NET apps to ASP.NET Core, see Migrate from ASP.NET to ASP.NET Core.

How to download a sample

Many of the articles and tutorials include links to sample code.

1. Download the ASP.NET repository zip file.
2. Unzip the *Docs-master.zip* file.
3. Use the URL in the sample link to help you navigate to the sample directory.

Preprocessor directives in sample code

To demonstrate multiple scenarios, sample apps use the `#define` and `#if-#else/#elif-#endif` preprocessor directives to selectively compile and run different sections of sample code. For those samples that make use of this approach, set the `#define` directive at the top of the C# files to define the symbol associated with the scenario that you want to run. Some samples require defining the symbol at the top of multiple files in order to run a scenario.

For example, the following `#define` symbol list indicates that four scenarios are available (one scenario per symbol). The current sample configuration runs the `TemplateCode` scenario:

```
#define TemplateCode // or LogFromMain or ExpandDefault or FilterInCode
```

To change the sample to run the `ExpandDefault` scenario, define the `ExpandDefault` symbol and leave the remaining symbols commented-out:

```
#define ExpandDefault // TemplateCode or LogFromMain or FilterInCode
```

For more information on using C# preprocessor directives to selectively compile sections of code, see #define (C# Reference) and #if (C# Reference).

Regions in sample code

Some sample apps contain sections of code surrounded by #region and #endregion C# directives. The documentation build system injects these regions into the rendered documentation topics.

Region names usually contain the word "snippet." The following example shows a region named `snippet_WebHostDefaults`:

```
#region snippet_WebHostDefaults
Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
#endregion
```

The preceding C# code snippet is referenced in the topic's markdown file with the following line:

```
[!code-csharp[](sample/SampleApp/Program.cs?name=snippet_WebHostDefaults)]
```

You may safely ignore (or remove) the `#region` and `#endregion` directives that surround the code. Don't alter the code within these directives if you plan to run the sample scenarios described in the topic. Feel free to alter the code when experimenting with other scenarios.

For more information, see Contribute to the ASP.NET documentation: Code snippets.

Next steps

For more information, see the following resources:

- Get started with ASP.NET Core
- Publish an ASP.NET Core app to Azure with Visual Studio
- ASP.NET Core fundamentals
- The weekly ASP.NET community standup covers the team's progress and plans. It features new blogs and third-party software.

Choose between ASP.NET 4.x and ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

ASP.NET Core is a redesign of ASP.NET 4.x. This article lists the differences between them.

ASP.NET Core

ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based web apps on Windows, macOS, or Linux.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
  - Kestrel
  - IIS
  - HTTP.sys
  - Nginx
  - Apache
  - Docker
- Side-by-side versioning.
- Tooling that simplifies modern web development.

ASP.NET 4.x

ASP.NET 4.x is a mature framework that provides the services needed to build enterprise-grade, server-based web apps on Windows.

Framework selection

The following table compares ASP.NET Core to ASP.NET 4.x.

ASP.NET CORE	ASP.NET 4.X
Build for Windows, macOS, or Linux	Build for Windows

ASP.NET CORE	ASP.NET 4.X
Razor Pages is the recommended approach to create a Web UI as of ASP.NET Core 2.x. See also MVC, Web API, and SignalR.	Use Web Forms, SignalR, MVC, Web API, WebHooks, or Web Pages
Multiple versions per machine	One version per machine
Develop with Visual Studio, Visual Studio for Mac, or Visual Studio Code using C# or F#	Develop with Visual Studio using C#, VB, or F#
Higher performance than ASP.NET 4.x	Good performance
Use .NET Core runtime	Use .NET Framework runtime

See ASP.NET Core targeting .NET Framework for information on ASP.NET Core 2.x support on .NET Framework.

ASP.NET Core scenarios

- Websites
- APIs
- Real-time
- Deploy an ASP.NET Core app to Azure

ASP.NET 4.x scenarios

- Websites
- APIs
- Real-time
- Create an ASP.NET 4.x web app in Azure

Additional resources

- Introduction to ASP.NET
- Introduction to ASP.NET Core
- Deploy ASP.NET Core apps to Azure App Service

Tutorial: Get started with ASP.NET Core

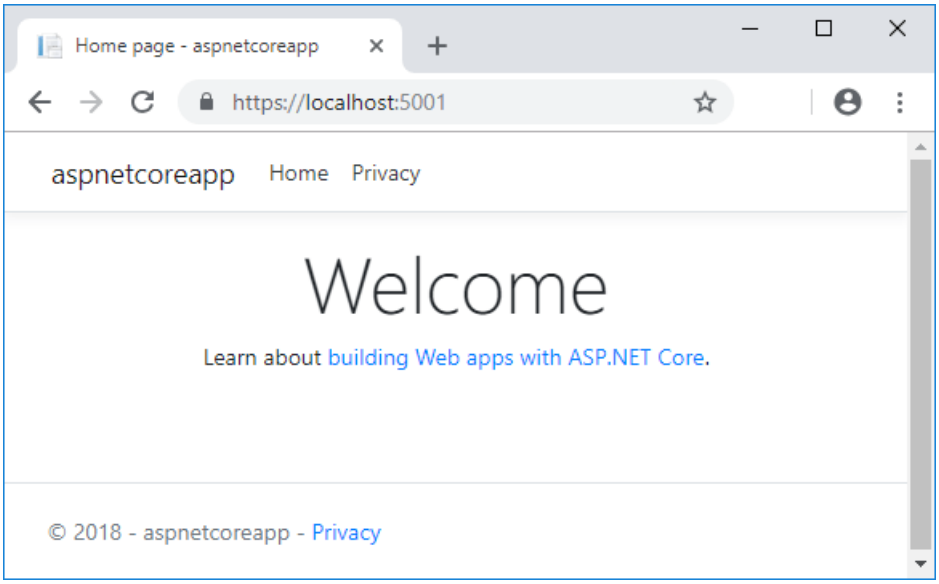
9/22/2020 • 2 minutes to read • Edit Online

This tutorial shows how to create and run an ASP.NET Core web app using the .NET Core CLI.

You'll learn how to:

- Create a web app project.
- Trust the development certificate.
- Run the app.
- Edit a Razor page.

At the end, you'll have a working web app running on your local machine.



Prerequisites

.NET Core 3.1 SDK or later

Create a web app project

Open a command shell, and enter the following command:

```
dotnet new webapp -o aspnetcoreapp
```

The preceding command:

- Creates a new web app.
- The `-o aspnetcoreapp` parameter creates a directory named *aspnetcoreapp* with the source files for the app.

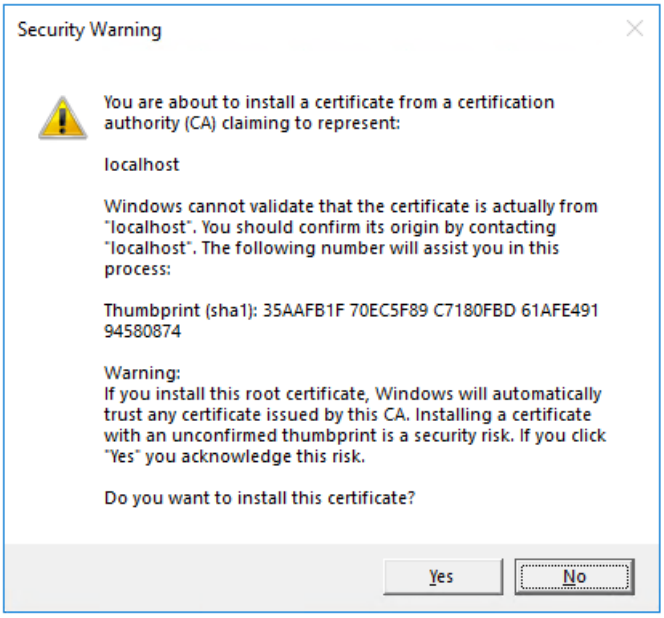
Trust the development certificate

Trust the HTTPS development certificate:

- Windows
- macOS
- Linux

```
dotnet dev-certs https --trust
```

The preceding command displays the following dialog:



Select **Yes** if you agree to trust the development certificate.

For more information, see Trust the ASP.NET Core HTTPS development certificate

Run the app

Run the following commands:

```
cd aspnetcoreapp
dotnet watch run
```

After the command shell indicates that the app has started, browse to `https://localhost:5001`.

Edit a Razor page

Open *Pages/Index.cshtml* and modify and save the page with the following highlighted markup:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Hello, world! The time on the server is @DateTime.Now</p>
</div>
```

Browse to `https://localhost:5001`, refresh the page, and verify the changes are displayed.

Next steps

In this tutorial, you learned how to:

- Create a web app project.
- Trust the development certificate.
- Run the project.
- Make a change.

To learn more about ASP.NET Core, see the recommended learning path in the introduction:

Introduction to ASP.NET Core

What's new in ASP.NET Core 3.1

9/22/2020 • 2 minutes to read • Edit Online

This article highlights the most significant changes in ASP.NET Core 3.1 with links to relevant documentation.

Partial class support for Razor components

Razor components are now generated as partial classes. Code for a Razor component can be written using a code-behind file defined as a partial class rather than defining all the code for the component in a single file. For more information, see Partial class support.

Blazor Component Tag Helper and pass parameters to top-level components

In Blazor with ASP.NET Core 3.0, components were rendered into pages and views using an HTML Helper (`Html.RenderComponentAsync`). In ASP.NET Core 3.1, render a component from a page or view with the new Component Tag Helper:

```
<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

The HTML Helper remains supported in ASP.NET Core 3.1, but the Component Tag Helper is recommended.

Blazor Server apps can now pass parameters to top-level components during the initial render. Previously you could only pass parameters to a top-level component with RenderMode.Static. With this release, both RenderMode.Server and RenderMode.ServerPrerendered are supported. Any specified parameter values are serialized as JSON and included in the initial response.

For example, prerender a `Counter` component with an increment amount (`IncrementAmount`):

```
<component type="typeof(Counter)" render-mode="ServerPrerendered"
    param-IncrementAmount="10" />
```

For more information, see Integrate components into Razor Pages and MVC apps.

Support for shared queues in HTTP.sys

HTTP.sys supports creating anonymous request queues. In ASP.NET Core 3.1, we've added to ability to create or attach to an existing named HTTP.sys request queue. Creating or attaching to an existing named HTTP.sys request queue enables scenarios where the HTTP.sys controller process that owns the queue is independent of the listener process. This independence makes it possible to preserve existing connections and enqueued requests between listener process restarts:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
     Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            // ...
            webBuilder.UseHttpSys(options =>
            {
                options.RequestQueueName = "MyExistingQueue";
                options.RequestQueueMode = RequestQueueMode.CreateOrAttach;
            });
        });
```

Breaking changes for SameSite cookies

The behavior of SameSite cookies has changed to reflect upcoming browser changes. This may affect authentication scenarios like AzureAd, OpenIdConnect, or WsFederation. For more information, see Work with SameSite cookies in ASP.NET Core.

Prevent default actions for events in Blazor apps

Use the `@on{EVENT}:preventDefault` directive attribute to prevent the default action for an event. In the following example, the default action of displaying the key's character in the text box is prevented:

```
<input value="@_count" @onkeypress="KeyHandler" @onkeypress:preventDefault />
```

For more information, see Prevent default actions.

Stop event propagation in Blazor apps

Use the `@on{EVENT}:stopPropagation` directive attribute to stop event propagation. In the following example, selecting the check box prevents click events from the child `<div>` from propagating to the parent `<div>`:

```
<input @bind="_stopPropagation" type="checkbox" />

<div @onclick="OnSelectParentDiv">
    <div @onclick="OnSelectChildDiv" @onclick:stopPropagation="_stopPropagation">
        ...
    </div>
</div>

@code {
    private bool _stopPropagation = false;
}
```

For more information, see Stop event propagation.

Detailed errors during Blazor app development

When a Blazor app isn't functioning properly during development, receiving detailed error information from the app assists in troubleshooting and fixing the issue. When an error occurs, Blazor apps display a gold bar at the bottom of the screen:

- During development, the gold bar directs you to the browser console, where you can see the exception.
- In production, the gold bar notifies the user that an error has occurred and recommends refreshing the browser.

For more information, see Detailed errors during development.

What's new in ASP.NET Core 3.0

9/22/2020 • 14 minutes to read • Edit Online

This article highlights the most significant changes in ASP.NET Core 3.0 with links to relevant documentation.

Blazor

Blazor is a new framework in ASP.NET Core for building interactive client-side web UI with .NET:

- Create rich interactive UIs using C# instead of JavaScript.
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.

Blazor framework supported scenarios:

- Reusable UI components (Razor components)
- Client-side routing
- Component layouts
- Support for dependency injection
- Forms and validation
- Build component libraries with Razor class libraries
- JavaScript interop

For more information, see Introduction to ASP.NET Core Blazor.

Blazor Server

Blazor decouples component rendering logic from how UI updates are applied. Blazor Server provides support for hosting Razor components on the server in an ASP.NET Core app. UI updates are handled over a SignalR connection. Blazor Server is supported in ASP.NET Core 3.0.

Blazor WebAssembly (Preview)

Blazor apps can also be run directly in the browser using a WebAssembly-based .NET runtime. Blazor WebAssembly is in preview and *not* supported in ASP.NET Core 3.0. Blazor WebAssembly will be supported in a future release of ASP.NET Core.

Razor components

Blazor apps are built from components. Components are self-contained chunks of user interface (UI), such as a page, dialog, or form. Components are normal .NET classes that define UI rendering logic and client-side event handlers. You can create rich interactive web apps without JavaScript.

Components in Blazor are typically authored using Razor syntax, a natural blend of HTML and C#. Razor components are similar to Razor Pages and MVC views in that they both use Razor. Unlike pages and views, which are based on a request-response model, components are used specifically for handling UI composition.

gRPC

gRPC:

- Is a popular, high-performance RPC (remote procedure call) framework.
- Offers an opinionated contract-first approach to API development.

- Uses modern technologies such as:
  - HTTP/2 for transport.
  - Protocol Buffers as the interface description language.
  - Binary serialization format.
- Provides features such as:
  - Authentication
  - Bidirectional streaming and flow control.
  - Cancellation and timeouts.

gRPC functionality in ASP.NET Core 3.0 includes:

- Grpc.AspNetCore: An ASP.NET Core framework for hosting gRPC services. gRPC on ASP.NET Core integrates with standard ASP.NET Core features like logging, dependency injection (DI), authentication, and authorization.
- Grpc.Net.Client: A gRPC client for .NET Core that builds upon the familiar `HttpClient`.
- Grpc.Net.ClientFactory: gRPC client integration with `HttpClientFactory`.

For more information, see Introduction to gRPC on .NET Core.

SignalR

See Update SignalR code for migration instructions. SignalR now uses `System.Text.Json` to serialize/deserialize JSON messages. See Switch to Newtonsoft.Json for instructions to restore the `Newtonsoft.Json`-based serializer.

In the JavaScript and .NET Clients for SignalR, support was added for automatic reconnection. By default, the client tries to reconnect immediately and retry after 2, 10, and 30 seconds if necessary. If the client successfully reconnects, it receives a new connection ID. Automatic reconnect is opt-in:

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

The reconnection intervals can be specified by passing an array of millisecond-based durations:

```
.withAutomaticReconnect([0, 3000, 5000, 10000, 15000, 30000])
//.withAutomaticReconnect([0, 2000, 10000, 30000]) The default intervals.
```

A custom implementation can be passed in for full control of the reconnection intervals.

If the reconnection fails after the last reconnect interval:

- The client considers the connection is offline.
- The client stops trying to reconnect.

During reconnection attempts, update the app UI to notify the user that the reconnection is being attempted.

To provide UI feedback when the connection is interrupted, the SignalR client API has been expanded to include the following event handlers:

- `onreconnecting`: Gives developers an opportunity to disable UI or to let users know the app is offline.
- `onreconnected`: Gives developers an opportunity to update the UI once the connection is reestablished.

The following code uses `onreconnecting` to update the UI while trying to connect:

```
connection.onreconnecting((error) => {
    const status = `Connection lost due to error "${error}". Reconnecting.`;
    document.getElementById("messageInput").disabled = true;
    document.getElementById("sendButton").disabled = true;
    document.getElementById("connectionStatus").innerText = status;
});
```

The following code uses `onreconnected` to update the UI on connection:

```
connection.onreconnected((connectionId) => {
    const status = `Connection reestablished. Connected.`;
    document.getElementById("messageInput").disabled = false;
    document.getElementById("sendButton").disabled = false;
    document.getElementById("connectionStatus").innerText = status;
});
```

SignalR 3.0 and later provides a custom resource to authorization handlers when a hub method requires authorization. The resource is an instance of `HubInvocationContext`. The `HubInvocationContext` includes the:

- `HubCallerContext`
- Name of the hub method being invoked.
- Arguments to the hub method.

Consider the following example of a chat room app allowing multiple organization sign-in via Azure Active Directory. Anyone with a Microsoft account can sign in to chat, but only members of the owning organization can ban users or view users' chat histories. The app could restrict certain functionality from specific users.

```
public class DomainRestrictedRequirement :
    AuthorizationHandler<DomainRestrictedRequirement, HubInvocationContext>,
    IAuthorizationRequirement
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        DomainRestrictedRequirement requirement,
        HubInvocationContext resource)
    {
        if (context.User?.Identity?.Name == null)
        {
            return Task.CompletedTask;
        }

        if (IsUserAllowedToDoThis(resource.HubMethodName, context.User.Identity.Name))
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }

    private bool IsUserAllowedToDoThis(string hubMethodName, string currentUsername)
    {
        if (hubMethodName.Equals("banUser", StringComparison.OrdinalIgnoreCase))
        {
            return currentUsername.Equals("bob42@jabbr.net", StringComparison.OrdinalIgnoreCase);
        }

        return currentUsername.EndsWith("@jabbr.net", StringComparison.OrdinalIgnoreCase));
    }
}
```

In the preceding code, `DomainRestrictedRequirement` serves as a custom `IAuthorizationRequirement`. Because the

`HubInvocationContext` resource parameter is being passed in, the internal logic can:

- Inspect the context in which the Hub is being called.
- Make decisions on allowing the user to execute individual Hub methods.

Individual Hub methods can be marked with the name of the policy the code checks at run-time. As clients attempt to call individual Hub methods, the `DomainRestrictedRequirement` handler runs and controls access to the methods. Based on the way the `DomainRestrictedRequirement` controls access:

- All logged-in users can call the `SendMessage` method.
- Only users who have logged in with a `@jabbr.net` email address can view users' histories.
- Only `bob42@jabbr.net` can ban users from the chat room.

```
[Authorize]
public class ChatHub : Hub
{
    public void SendMessage(string message)
    {
    }

    [Authorize("DomainRestricted")]
    public void BanUser(string username)
    {
    }

    [Authorize("DomainRestricted")]
    public void ViewUserHistory(string username)
    {
    }
}
```

Creating the `DomainRestricted` policy might involve:

- In *Startup.cs*, adding the new policy.
- Provide the custom `DomainRestrictedRequirement` requirement as a parameter.
- Registering `DomainRestricted` with the authorization middleware.

```
services
    .AddAuthorization(options =>
    {
        options.AddPolicy("DomainRestricted", policy =>
        {
            policy.Requirements.Add(new DomainRestrictedRequirement());
        });
    });
```

SignalR hubs use Endpoint Routing. SignalR hub connection was previously done explicitly:

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
});
```

In the previous version, developers needed to wire up controllers, Razor pages, and hubs in a variety of places. Explicit connection results in a series of nearly-identical routing segments:

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
});

app.UseRouting(routes =>
{
    routes.MapRazorPages();
});
```

SignalR 3.0 hubs can be routed via endpoint routing. With endpoint routing, typically all routing can be configured in `UseRouting`:

```
app.UseRouting(routes =>
{
    routes.MapRazorPages();
    routes.MapHub<ChatHub>("hubs/chat");
});
```

ASP.NET Core 3.0 SignalR added:

Client-to-server streaming. With client-to-server streaming, server-side methods can take instances of either an `IAsyncEnumerable<T>` or `ChannelReader<T>`. In the following C# sample, the `UploadStream` method on the Hub will receive a stream of strings from the client:

```
public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        // process content
    }
}
```

.NET client apps can pass either an `IAsyncEnumerable<T>` or `ChannelReader<T>` instance as the `stream` argument of the `UploadStream` Hub method above.

After the `for` loop has completed and the local function exits, the stream completion is sent:

```
async IAsyncEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
        var data = await FetchSomeData();
        yield return data;
    }
}

await connection.SendAsync("UploadStream", clientStreamData());
```

JavaScript client apps use the SignalR `Subject` (or an RxJS Subject) for the `stream` argument of the `UploadStream` Hub method above.

```
let subject = new signalR.Subject();
await connection.send("StartStream", "MyAsciiArtStream", subject);
```

The JavaScript code could use the `subject.next` method to handle strings as they are captured and ready to be sent to the server.

```
subject.next("example");
subject.complete();
```

Using code like the two preceding snippets, real-time streaming experiences can be created.

New JSON serialization

ASP.NET Core 3.0 now uses System.Text.Json by default for JSON serialization:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically higher performance than `Newtonsoft.Json`.

To add Json.NET to ASP.NET Core 3.0, see Add Newtonsoft.Json-based JSON format support.

New Razor directives

The following list contains new Razor directives:

- `@attribute`: The `@attribute` directive applies the given attribute to the class of the generated page or view. For example, `@attribute [Authorize]`.
- `@implements`: The `@implements` directive implements an interface for the generated class. For example, `@implements IDisposable`.

IdentityServer4 supports authentication and authorization for web APIs and SPAs

ASP.NET Core 3.0 offers authentication in Single Page Apps (SPAs) using the support for web API authorization. ASP.NET Core Identity for authenticating and storing users is combined with IdentityServer4 for implementing OpenID Connect.

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core 3.0. It enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see the IdentityServer4 documentation or Authentication and authorization for SPAs.

Certificate and Kerberos authentication

Certificate authentication requires:

- Configuring the server to accept certificates.
- Adding the authentication middleware in `Startup.Configure`.
- Adding the certificate authentication service in `Startup.ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
            .AddCertificate();
    // Other service configuration removed.
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Options for certificate authentication include the ability to:

- Accept self-signed certificates.
- Check for certificate revocation.
- Check that the proffered certificate has the right usage flags in it.

A default user principal is constructed from the certificate properties. The user principal contains an event that enables supplementing or replacing the principal. For more information, see Configure certificate authentication in ASP.NET Core.

Windows Authentication has been extended onto Linux and macOS. In previous versions, Windows Authentication was limited to IIS and HttpSys. In ASP.NET Core 3.0, Kestrel has the ability to use Negotiate, Kerberos, and NTLM on Windows, Linux, and macOS for Windows domain-joined hosts. Kestrel support of these authentication schemes is provided by the Microsoft.AspNetCore.Authentication.Negotiate NuGet package. As with the other authentication services, configure authentication app wide, then configure the service:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
        .AddNegotiate();
    // Other service configuration removed.
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Host requirements:

- Windows hosts must have Service Principal Names (SPNs) added to the user account hosting the app.
- Linux and macOS machines must be joined to the domain.
  - SPNs must be created for the web process.
  - Keytab files must be generated and configured on the host machine.

For more information, see Configure Windows Authentication in ASP.NET Core.

Template changes

The web UI templates (Razor Pages, MVC with controller and views) have the following removed:

- The cookie consent UI is no longer included. To enable the cookie consent feature in an ASP.NET Core 3.0 template-generated app, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

- Scripts and related static assets are now referenced as local files instead of using CDNs. For more information, see Scripts and related static assets are now referenced as local files instead of using CDNs based on the current environment (aspnet/AspNetCore.Docs #14350).

The Angular template updated to use Angular 8.

The Razor class library (RCL) template defaults to Razor component development by default. A new template option in Visual Studio provides template support for pages and views. When creating an RCL from the template in a command shell, pass the `--support-pages-and-views` option (`dotnet new razorclasslib --support-pages-and-views`).

Generic Host

The ASP.NET Core 3.0 templates use .NET Generic Host. Previous versions used WebHostBuilder. Using the .NET Core Generic Host (HostBuilder) provides better integration of ASP.NET Core apps with other server scenarios that aren't web-specific. For more information, see HostBuilder replaces WebHostBuilder.

Host configuration

Prior to the release of ASP.NET Core 3.0, environment variables prefixed with `ASPNETCORE_` were loaded for host configuration of the Web Host. In 3.0, `AddEnvironmentVariables` is used to load environment variables prefixed with `DOTNET_` for host configuration with `CreateDefaultBuilder` .

Changes to Startup constructor injection

The Generic Host only supports the following types for `Startup` constructor injection:

- IHostEnvironment
- `IWebHostEnvironment`
- IConfiguration

All services can still be injected directly as arguments to the `Startup.Configure` method. For more information, see Generic Host restricts Startup constructor injection (aspnet/Announcements #353).

Kestrel

- Kestrel configuration has been updated for the migration to the Generic Host. In 3.0, Kestrel is configured on the web host builder provided by `ConfigureWebHostDefaults` .
- Connection Adapters have been removed from Kestrel and replaced with Connection Middleware, which is similar to HTTP Middleware in the ASP.NET Core pipeline but for lower-level connections.
- The Kestrel transport layer has been exposed as a public interface in `Connections.Abstractions` .
- Ambiguity between headers and trailers has been resolved by moving trailing headers to a new collection.
- Synchronous I/O APIs, such as `HttpRequest.Body.Read` , are a common source of thread starvation leading to app crashes. In 3.0, `AllowSynchronousIO` is disabled by default.

For more information, see Migrate from ASP.NET Core 2.2 to 3.0.

HTTP/2 enabled by default

HTTP/2 is enabled by default in Kestrel for HTTPS endpoints. HTTP/2 support for IIS or HTTP.sys is enabled when supported by the operating system.

EventCounters on request

The Hosting EventSource, `Microsoft.AspNetCore.Hosting` , emits the following new EventCounter types related to incoming requests:

- `requests-per-second`
- `total-requests`
- `current-requests`
- `failed-requests`

Endpoint routing

Endpoint Routing, which allows frameworks (for example, MVC) to work well with middleware, is enhanced:

- The order of middleware and endpoints is configurable in the request processing pipeline of `Startup.Configure`.
- Endpoints and middleware compose well with other ASP.NET Core-based technologies, such as Health Checks.
- Endpoints can implement a policy, such as CORS or authorization, in both middleware and MVC.
- Filters and attributes can be placed on methods in controllers.

For more information, see Routing in ASP.NET Core.

Health Checks

Health Checks use endpoint routing with the Generic Host. In `Startup.Configure`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
});
```

Health Checks endpoints can:

- Specify one or more permitted hosts/ports.
- Require authorization.
- Require CORS.

For more information, see the following articles:

- Migrate from ASP.NET Core 2.2 to 3.0
- Health checks in ASP.NET Core

Pipes on HttpContext

It's now possible to read the request body and write the response body using the System.IO.Pipelines API. The `HttpRequest.BodyReader` property provides a PipeReader that can be used to read the request body. The `HttpResponse.BodyWriter` property provides a PipeWriter that can be used to write the response body. `HttpRequest.BodyReader` is an analogue of the `HttpRequest.Body` stream. `HttpResponse.BodyWriter` is an analogue of the `HttpResponse.Body` stream.

Improved error reporting in IIS

Startup errors when hosting ASP.NET Core apps in IIS now produce richer diagnostic data. These errors are reported to the Windows Event Log with stack traces wherever applicable. In addition, all warnings, errors, and unhandled exceptions are logged to the Windows Event Log.

Worker Service and Worker SDK

.NET Core 3.0 introduces the new Worker Service app template. This template provides a starting point for writing

long running services in .NET Core.

For more information, see:

- .NET Core Workers as Windows Services
- Background tasks with hosted services in ASP.NET Core
- Host ASP.NET Core in a Windows Service

Forwarded Headers Middleware improvements

In previous versions of ASP.NET Core, calling UseHsts and UseHttpsRedirection were problematic when deployed to an Azure Linux or behind any reverse proxy other than IIS. The fix for previous versions is documented in Forward the scheme for Linux and non-IIS reverse proxies.

This scenario is fixed in ASP.NET Core 3.0. The host enables the Forwarded Headers Middleware when the `ASPNETCORE_FORWARDEDHEADERS_ENABLED` environment variable is set to `true`. `ASPNETCORE_FORWARDEDHEADERS_ENABLED` is set to `true` in our container images.

Performance improvements

ASP.NET Core 3.0 includes many improvements that reduce memory usage and improve throughput:

- Reduction in memory usage when using the built-in dependency injection container for scoped services.
- Reduction in allocations across the framework, including middleware scenarios and routing.
- Reduction in memory usage for WebSocket connections.
- Memory reduction and throughput improvements for HTTPS connections.
- New optimized and fully asynchronous JSON serializer.
- Reduction in memory usage and throughput improvements in form parsing.

ASP.NET Core 3.0 only runs on .NET Core 3.0

As of ASP.NET Core 3.0, .NET Framework is no longer a supported target framework. Projects targeting .NET Framework can continue in a fully supported fashion using the .NET Core 2.1 LTS release. Most ASP.NET Core 2.1.x related packages will be supported indefinitely, beyond the three-year LTS period for .NET Core 2.1.

For migration information, see Port your code from .NET Framework to .NET Core.

Use the ASP.NET Core shared framework

The ASP.NET Core 3.0 shared framework, contained in the Microsoft.AspNetCore.App metapackage, no longer requires an explicit `<PackageReference />` element in the project file. The shared framework is automatically referenced when using the `Microsoft.NET.Sdk.Web` SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

Assemblies removed from the ASP.NET Core shared framework

The most notable assemblies removed from the ASP.NET Core 3.0 shared framework are:

- Newtonsoft.Json (Json.NET). To add Json.NET to ASP.NET Core 3.0, see Add Newtonsoft.Json-based JSON format support. ASP.NET Core 3.0 introduces `System.Text.Json` for reading and writing JSON. For more information, see New JSON serialization in this document.
- Entity Framework Core

For a complete list of assemblies removed from the shared framework, see Assemblies being removed from Microsoft.AspNetCore.App 3.0. For more information on the motivation for this change, see Breaking changes to Microsoft.AspNetCore.App in 3.0 and A first look at changes coming in ASP.NET Core 3.0.

What's new in ASP.NET Core 2.2

9/22/2020 • 5 minutes to read • Edit Online

This article highlights the most significant changes in ASP.NET Core 2.2, with links to relevant documentation.

OpenAPI Analyzers & Conventions

OpenAPI (formerly known as Swagger) is a language-agnostic specification for describing REST APIs. The OpenAPI ecosystem has tools that allow for discovering, testing, and producing client code using the specification. Support for generating and visualizing OpenAPI documents in ASP.NET Core MVC is provided via community driven projects such as NSwag and Swashbuckle.AspNetCore. ASP.NET Core 2.2 provides improved tooling and runtime experiences for creating OpenAPI documents.

For more information, see the following resources:

- Use web API analyzers
- Use web API conventions
- ASP.NET Core 2.2.0-preview1: OpenAPI Analyzers & Conventions

Problem details support

ASP.NET Core 2.1 introduced `ProblemDetails`, based on the RFC 7807 specification for carrying details of an error with an HTTP Response. In 2.2, `ProblemDetails` is the standard response for client error codes in controllers attributed with `ApiControllerAttribute`. An `IActionResult` returning a client error status code (4xx) now returns a `ProblemDetails` body. The result also includes a correlation ID that can be used to correlate the error using request logs. For client errors, `ProducesResponseType` defaults to using `ProblemDetails` as the response type. This is documented in OpenAPI / Swagger output generated using NSwag or Swashbuckle.AspNetCore.

Endpoint Routing

ASP.NET Core 2.2 uses a new *endpoint routing* system for improved dispatching of requests. The changes include new link generation API members and route parameter transformers.

For more information, see the following resources:

- Endpoint routing in 2.2
- Route parameter transformers (see **Routing** section)
- Differences between IRouter- and endpoint-based routing

Health checks

A new health checks service makes it easier to use ASP.NET Core in environments that require health checks, such as Kubernetes. Health checks includes middleware and a set of libraries that define an `IHealthCheck` abstraction and service.

Health checks are used by a container orchestrator or load balancer to quickly determine if a system is responding to requests normally. A container orchestrator might respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might respond to a health check by routing traffic away from the failing instance of the service.

Health checks are exposed by an application as an HTTP endpoint used by monitoring systems. Health checks can

be configured for a variety of real-time monitoring scenarios and monitoring systems. The health checks service integrates with the BeatPulse project. which makes it easier to add checks for dozens of popular systems and dependencies.

For more information, see Health checks in ASP.NET Core.

HTTP/2 in Kestrel

ASP.NET Core 2.2 adds support for HTTP/2.

HTTP/2 is a major revision of the HTTP protocol. Notable features of HTTP/2 include:

- Support for header compression.
- Fully multiplexed streams over a single connection.

While HTTP/2 preserves HTTP's semantics (for example, HTTP headers and methods), it's a breaking change from HTTP/1.x on how data is framed and sent between the client and server.

As a consequence of this change in framing, servers and clients need to negotiate the protocol version used. Application-Layer Protocol Negotiation (ALPN) is a TLS extension that allows the server and client to negotiate the protocol version used as part of their TLS handshake. While it is possible to have prior knowledge between the server and the client on the protocol, all major browsers support ALPN as the only way to establish an HTTP/2 connection.

For more information, see HTTP/2 support.

Kestrel configuration

In earlier versions of ASP.NET Core, Kestrel options are configured by calling `UseKestrel`. In 2.2, Kestrel options are configured by calling `ConfigureKestrel` on the host builder. This change resolves an issue with the order of `IServer` registrations for in-process hosting. For more information, see the following resources:

- Mitigate UseIIS conflict
- Configure Kestrel server options with ConfigureKestrel

IIS in-process hosting

In earlier versions of ASP.NET Core, IIS serves as a reverse proxy. In 2.2, the ASP.NET Core Module can boot the CoreCLR and host an app inside the IIS worker process (*w3wp.exe*). In-process hosting provides performance and diagnostic gains when running with IIS.

For more information, see in-process hosting for IIS.

SignalR Java client

ASP.NET Core 2.2 introduces a Java Client for SignalR. This client supports connecting to an ASP.NET Core SignalR Server from Java code, including Android apps.

For more information, see ASP.NET Core SignalR Java client.

CORS improvements

In earlier versions of ASP.NET Core, CORS Middleware allows `Accept`, `Accept-Language`, `Content-Language`, and `Origin` headers to be sent regardless of the values configured in `CorsPolicy.Headers`. In 2.2, a CORS Middleware policy match is only possible when the headers sent in `Access-Control-Request-Headers` exactly match the headers stated in `WithHeaders`.

For more information, see CORS Middleware.

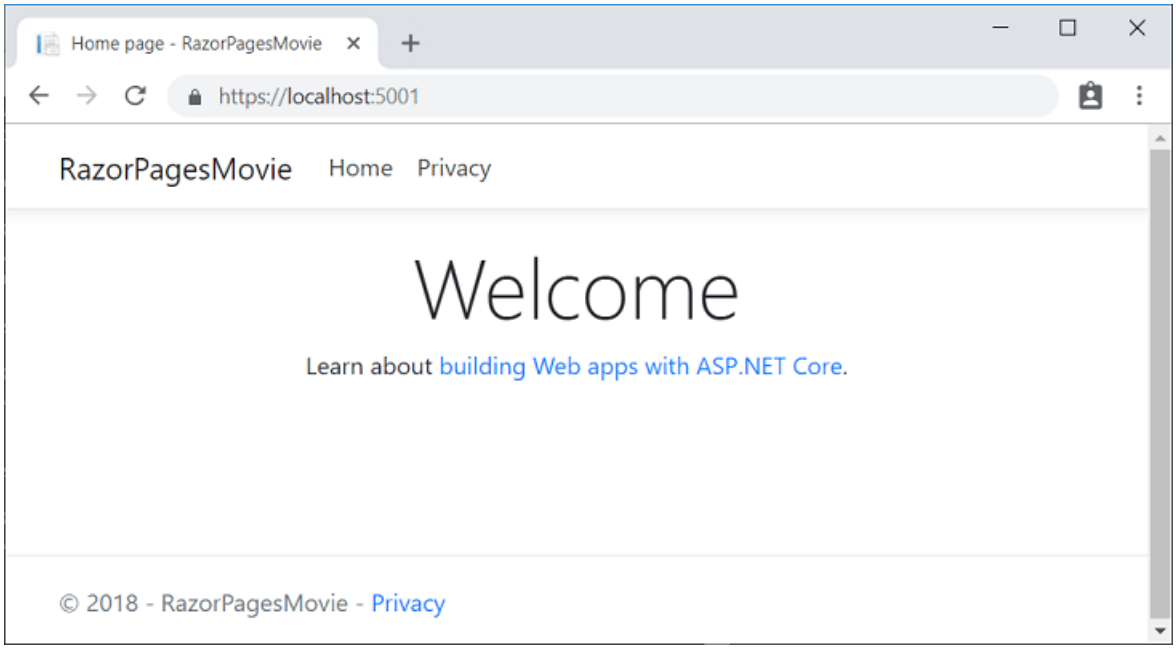
Response compression

ASP.NET Core 2.2 can compress responses with the Brotli compression format.

For more information, see Response Compression Middleware supports Brotli compression.

Project templates

ASP.NET Core web project templates were updated to Bootstrap 4 and Angular 6. The new look is visually simpler and makes it easier to see the important structures of the app.



Validation performance

MVC's validation system is designed to be extensible and flexible, allowing you to determine on a per request basis which validators apply to a given model. This is great for authoring complex validation providers. However, in the most common case an application only uses the built-in validators and don't require this extra flexibility. Built-in validators include DataAnnotations such as [Required] and [StringLength], and `IValidatableObject`.

In ASP.NET Core 2.2, MVC can short-circuit validation if it determines that a given model graph doesn't require validation. Skipping validation results in significant improvements when validating models that can't or don't have any validators. This includes objects such as collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`), or complex object graphs without many validators.

HTTP Client performance

In ASP.NET Core 2.2, the performance of `SocketsHttpHandler` was improved by reducing connection pool locking contention. For apps that make many outgoing HTTP requests, such as some microservices architectures, throughput is improved. Under load, `HttpClient` throughput can be improved by up to 60% on Linux and 20% on Windows.

For more information, see the pull request that made this improvement.

Additional information

For the complete list of changes, see the ASP.NET Core 2.2 Release Notes.

What's new in ASP.NET Core 2.1

9/22/2020 • 6 minutes to read • Edit Online

This article highlights the most significant changes in ASP.NET Core 2.1, with links to relevant documentation.

SignalR

SignalR has been rewritten for ASP.NET Core 2.1. ASP.NET Core SignalR includes a number of improvements:

- A simplified scale-out model.
- A new JavaScript client with no jQuery dependency.
- A new compact binary protocol based on MessagePack.
- Support for custom protocols.
- A new streaming response model.
- Support for clients based on bare WebSockets.

For more information, see ASP.NET Core SignalR.

Razor class libraries

ASP.NET Core 2.1 makes it easier to build and include Razor-based UI in a library and share it across multiple projects. The new Razor SDK enables building Razor files into a class library project that can be packaged into a NuGet package. Views and pages in libraries are automatically discovered and can be overridden by the app. By integrating Razor compilation into the build:

- The app startup time is significantly faster.
- Fast updates to Razor views and pages at runtime are still available as part of an iterative development workflow.

For more information, see Create reusable UI using the Razor Class Library project.

Identity UI library & scaffolding

ASP.NET Core 2.1 provides ASP.NET Core Identity as a Razor Class Library. Apps that include Identity can apply the new Identity scaffolder to selectively add the source code contained in the Identity Razor Class Library (RCL). You might want to generate source code so you can modify the code and change the behavior. For example, you could instruct the scaffolder to generate the code used in registration. Generated code takes precedence over the same code in the Identity RCL.

Apps that do **not** include authentication can apply the Identity scaffolder to add the RCL Identity package. You have the option of selecting Identity code to be generated.

For more information, see Scaffold Identity in ASP.NET Core projects.

HTTPS

With the increased focus on security and privacy, enabling HTTPS for web apps is important. HTTPS enforcement is becoming increasingly strict on the web. Sites that don't use HTTPS are considered insecure. Browsers (Chromium, Mozilla) are starting to enforce that web features must be used from a secure context. GDPR requires the use of HTTPS to protect user privacy. While using HTTPS in production is critical, using HTTPS in development can help prevent issues in deployment (for example, insecure links). ASP.NET Core 2.1 includes a number of improvements

that make it easier to use HTTPS in development and to configure HTTPS in production. For more information, see Enforce HTTPS.

On by default

To facilitate secure website development, HTTPS is now enabled by default. Starting in 2.1, Kestrel listens on `https://localhost:5001` when a local development certificate is present. A development certificate is created:

- As part of the .NET Core SDK first-run experience, when you use the SDK for the first time.
- Manually using the new `dev-certs` tool.

Run `dotnet dev-certs https --trust` to trust the certificate.

HTTPS redirection and enforcement

Web apps typically need to listen on both HTTP and HTTPS, but then redirect all HTTP traffic to HTTPS. In 2.1, specialized HTTPS redirection middleware that intelligently redirects based on the presence of configuration or bound server ports has been introduced.

Use of HTTPS can be further enforced using HTTP Strict Transport Security Protocol (HSTS). HSTS instructs browsers to always access the site via HTTPS. ASP.NET Core 2.1 adds HSTS middleware that supports options for max age, subdomains, and the HSTS preload list.

Configuration for production

In production, HTTPS must be explicitly configured. In 2.1, default configuration schema for configuring HTTPS for Kestrel has been added. Apps can be configured to use:

- Multiple endpoints including the URLs. For more information, see Kestrel web server implementation: Endpoint configuration.
- The certificate to use for HTTPS either from a file on disk or from a certificate store.

GDPR

ASP.NET Core provides APIs and templates to help meet some of the EU General Data Protection Regulation (GDPR) requirements. For more information, see GDPR support in ASP.NET Core. A sample app shows how to use and lets you test most of the GDPR extension points and APIs added to the ASP.NET Core 2.1 templates.

Integration tests

A new package is introduced that streamlines test creation and execution. The Microsoft.AspNetCore.Mvc.Testing package handles the following tasks:

- Copies the dependency file (**.deps*) from the tested app into the test project's *bin* folder.
- Sets the content root to the tested app's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the tested app with TestServer.

The following test uses xUnit to check that the Index page loads with a success status code and with the correct Content-Type header:

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task GetHomePage()
    {
        // Act
        var response = await _client.GetAsync("/");

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

For more information, see the Integration tests topic.

[ApiController], ActionResult<T>

ASP.NET Core 2.1 adds new programming conventions that make it easier to build clean and descriptive web APIs. `ActionResult<T>` is a new type added to allow an app to return either a response type or any other action result (similar to IActionResult), while still indicating the response type. The `[ApiController]` attribute has also been added as the way to opt in to Web API-specific conventions and behaviors.

For more information, see Build Web APIs with ASP.NET Core.

IHttpClientFactory

ASP.NET Core 2.1 includes a new `IHttpClientFactory` service that makes it easier to configure and consume instances of `HttpClient` in apps. `HttpClient` already has the concept of delegating handlers that could be linked together for outgoing HTTP requests. The factory:

- Makes registering of instances of `HttpClient` per named client more intuitive.
- Implements a Polly handler that allows Polly policies to be used for Retry, CircuitBreakers, etc.

For more information, see Initiate HTTP Requests.

Kestrel transport configuration

With the release of ASP.NET Core 2.1, Kestrel's default transport is no longer based on Libuv but instead based on managed sockets. For more information, see Kestrel web server implementation: Transport configuration.

Generic host builder

The Generic Host Builder (`HostBuilder`) has been introduced. This builder can be used for apps that don't process HTTP requests (Messaging, background tasks, etc.).

For more information, see .NET Generic Host.

Updated SPA templates

The Single Page Application templates for Angular, React, and React with Redux are updated to use the standard project structures and build systems for each framework.

The Angular template is based on the Angular CLI, and the React templates are based on create-react-app.

For more information, see:

- Use the Angular project template with ASP.NET Core
- Use the React project template with ASP.NET Core
- Use the React-with-Redux project template with ASP.NET Core

Razor Pages search for Razor assets

In 2.1, Razor Pages search for Razor assets (such as layouts and partials) in the following directories in the listed order:

1. Current Pages folder.
2. */Pages/Shared/*
3. */Views/Shared/*

Razor Pages in an area

Razor Pages now support areas. To see an example of areas, create a new Razor Pages web app with individual user accounts. A Razor Pages web app with individual user accounts includes */Areas/Identity/Pages*.

MVC compatibility version

The SetCompatibilityVersion method allows an app to opt-in or opt-out of potentially breaking behavior changes introduced in ASP.NET Core MVC 2.1 or later.

For more information, see Compatibility version for ASP.NET Core MVC.

Migrate from 2.0 to 2.1

See Migrate from ASP.NET Core 2.0 to 2.1.

Additional information

For the complete list of changes, see the ASP.NET Core 2.1 Release Notes.

What's new in ASP.NET Core 2.0

9/22/2020 • 5 minutes to read • Edit Online

This article highlights the most significant changes in ASP.NET Core 2.0, with links to relevant documentation.

Razor Pages

Razor Pages is a new feature of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

For more information, see the introduction and tutorial:

- Introduction to Razor Pages
- Get started with Razor Pages

ASP.NET Core metapackage

A new ASP.NET Core metapackage includes all of the packages made and supported by the ASP.NET Core and Entity Framework Core teams, along with their internal and 3rd-party dependencies. You no longer need to choose individual ASP.NET Core features by package. All features are included in the Microsoft.AspNetCore.All package. The default templates use this package.

For more information, see Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0.

Runtime Store

Applications that use the `Microsoft.AspNetCore.All` metapackage automatically take advantage of the new .NET Core Runtime Store. The Store contains all the runtime assets needed to run ASP.NET Core 2.0 applications. When you use the `Microsoft.AspNetCore.All` metapackage, no assets from the referenced ASP.NET Core NuGet packages are deployed with the application because they already reside on the target system. The assets in the Runtime Store are also precompiled to improve application startup time.

For more information, see Runtime store

.NET Standard 2.0

The ASP.NET Core 2.0 packages target .NET Standard 2.0. The packages can be referenced by other .NET Standard 2.0 libraries, and they can run on .NET Standard 2.0-compliant implementations of .NET, including .NET Core 2.0 and .NET Framework 4.6.1.

The `Microsoft.AspNetCore.All` metapackage targets .NET Core 2.0 only, because it's intended to be used with the .NET Core 2.0 Runtime Store.

Configuration update

An `IConfiguration` instance is added to the services container by default in ASP.NET Core 2.0. `IConfiguration` in the services container makes it easier for applications to retrieve configuration values from the container.

For information about the status of planned documentation, see the GitHub issue.

Logging update

In ASP.NET Core 2.0, logging is incorporated into the dependency injection (DI) system by default. You add providers and configure filtering in the *Program.cs* file instead of in the *Startup.cs* file. And the default `ILoggerFactory` supports filtering in a way that lets you use one flexible approach for both cross-provider filtering and specific-provider filtering.

For more information, see Introduction to Logging.

Authentication update

A new authentication model makes it easier to configure authentication for an application using DI.

New templates are available for configuring authentication for web apps and web APIs using Azure AD B2C.

For information about the status of planned documentation, see the GitHub issue.

Identity update

We've made it easier to build secure web APIs using Identity in ASP.NET Core 2.0. You can acquire access tokens for accessing your web APIs using the Microsoft Authentication Library (MSAL).

For more information on authentication changes in 2.0, see the following resources:

- Account confirmation and password recovery in ASP.NET Core
- Enable QR Code generation for authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core 2.0

SPA templates

Single Page Application (SPA) project templates for Angular, Aurelia, Knockout.js, React.js, and React.js with Redux are available. The Angular template has been updated to Angular 4. The Angular and React templates are available by default; for information about how to get the other templates, see Create a new SPA project. For information about how to build a SPA in ASP.NET Core, see Use JavaScript Services to Create Single Page Applications in ASP.NET Core.

Kestrel improvements

The Kestrel web server has new features that make it more suitable as an Internet-facing server. A number of server constraint configuration options are added in the `KestrelServerOptions` class's new `Limits` property. Add limits for the following:

- Maximum client connections
- Maximum request body size
- Minimum request body data rate

For more information, see Kestrel web server implementation in ASP.NET Core.

WebListener renamed to HTTP.sys

The packages `Microsoft.AspNetCore.Server.WebListener` and `Microsoft.Net.Http.Server` have been merged into a new package `Microsoft.AspNetCore.Server.HttpSys`. The namespaces have been updated to match.

For more information, see HTTP.sys web server implementation in ASP.NET Core.

Enhanced HTTP header support

When using MVC to transmit a `FileStreamResult` or a `FileContentResult`, you now have the option to set an `ETag`

or a `LastModified` date on the content you transmit. You can set these values on the returned content with code similar to the following:

```
var data = Encoding.UTF8.GetBytes("This is a sample text from a binary array");
var entityTag = new EntityTagHeaderValue("\"MyCalculatedEtagValue\"");
return File(data, "text/plain", "downloadName.txt", lastModified: DateTime.UtcNow.AddSeconds(-5), entityTag:
entityTag);
```

The file returned to your visitors has the appropriate HTTP headers for the `ETag` and `LastModified` values.

If an application visitor requests content with a Range Request header, ASP.NET Core recognizes the request and handles the header. If the requested content can be partially delivered, ASP.NET Core appropriately skips and returns just the requested set of bytes. You don't need to write any special handlers into your methods to adapt or handle this feature; it's automatically handled for you.

Hosting startup and Application Insights

Hosting environments can now inject extra package dependencies and execute code during application startup, without the application needing to explicitly take a dependency or call any methods. This feature can be used to enable certain environments to "light-up" features unique to that environment without the application needing to know ahead of time.

In ASP.NET Core 2.0, this feature is used to automatically enable Application Insights diagnostics when debugging in Visual Studio and (after opting in) when running in Azure App Services. As a result, the project templates no longer add Application Insights packages and code by default.

For information about the status of planned documentation, see the GitHub issue.

Automatic use of anti-forgery tokens

ASP.NET Core has always helped HTML-encode content by default, but with the new version an extra step is taken to help prevent cross-site request forgery (XSRF) attacks. ASP.NET Core will now emit anti-forgery tokens by default and validate them on form POST actions and pages without extra configuration.

For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Automatic precompilation

Razor view pre-compilation is enabled during publish by default, reducing the publish output size and application startup time.

For more information, see Razor view compilation and precompilation in ASP.NET Core.

Razor support for C# 7.1

The Razor view engine has been updated to work with the new Roslyn compiler. That includes support for C# 7.1 features like Default Expressions, Inferred Tuple Names, and Pattern-Matching with Generics. To use C# 7.1 in your project, add the following property in your project file and then reload the solution:

```
<LangVersion>latest</LangVersion>
```

For information about the status of C# 7.1 features, see the Roslyn GitHub repository.

Other documentation updates for 2.0

- Visual Studio publish profiles for ASP.NET Core app deployment
- Key Management
- Configure Facebook authentication
- Configure Twitter authentication
- Configure Google authentication
- Configure Microsoft Account authentication

Migration guidance

For guidance on how to migrate ASP.NET Core 1.x applications to ASP.NET Core 2.0, see the following resources:

- Migrate from ASP.NET Core 1.x to ASP.NET Core 2.0
- Migrate Authentication and Identity to ASP.NET Core 2.0

Additional Information

For the complete list of changes, see the ASP.NET Core 2.0 Release Notes.

To connect with the ASP.NET Core development team's progress and plans, tune in to the ASP.NET Community Standup.

What's new in ASP.NET Core 1.1

9/22/2020 • 2 minutes to read • Edit Online

ASP.NET Core 1.1 includes the following new features:

- URL Rewriting Middleware
- Response Caching Middleware
- View Components as Tag Helpers
- Middleware as MVC filters
- Cookie-based TempData provider
- Azure App Service logging provider
- Azure Key Vault configuration provider
- Azure and Redis Storage Data Protection Key Repositories
- WebListener Server for Windows
- WebSockets support

Choosing between versions 1.0 and 1.1 of ASP.NET Core

ASP.NET Core 1.1 has more features than ASP.NET Core 1.0. In general, we recommend you use the latest version.

Additional Information

- ASP.NET Core 1.1.0 Release Notes
- To connect with the ASP.NET Core development team's progress and plans, tune in to the ASP.NET Community Standup.

Tutorial: Create a Razor Pages web app with ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

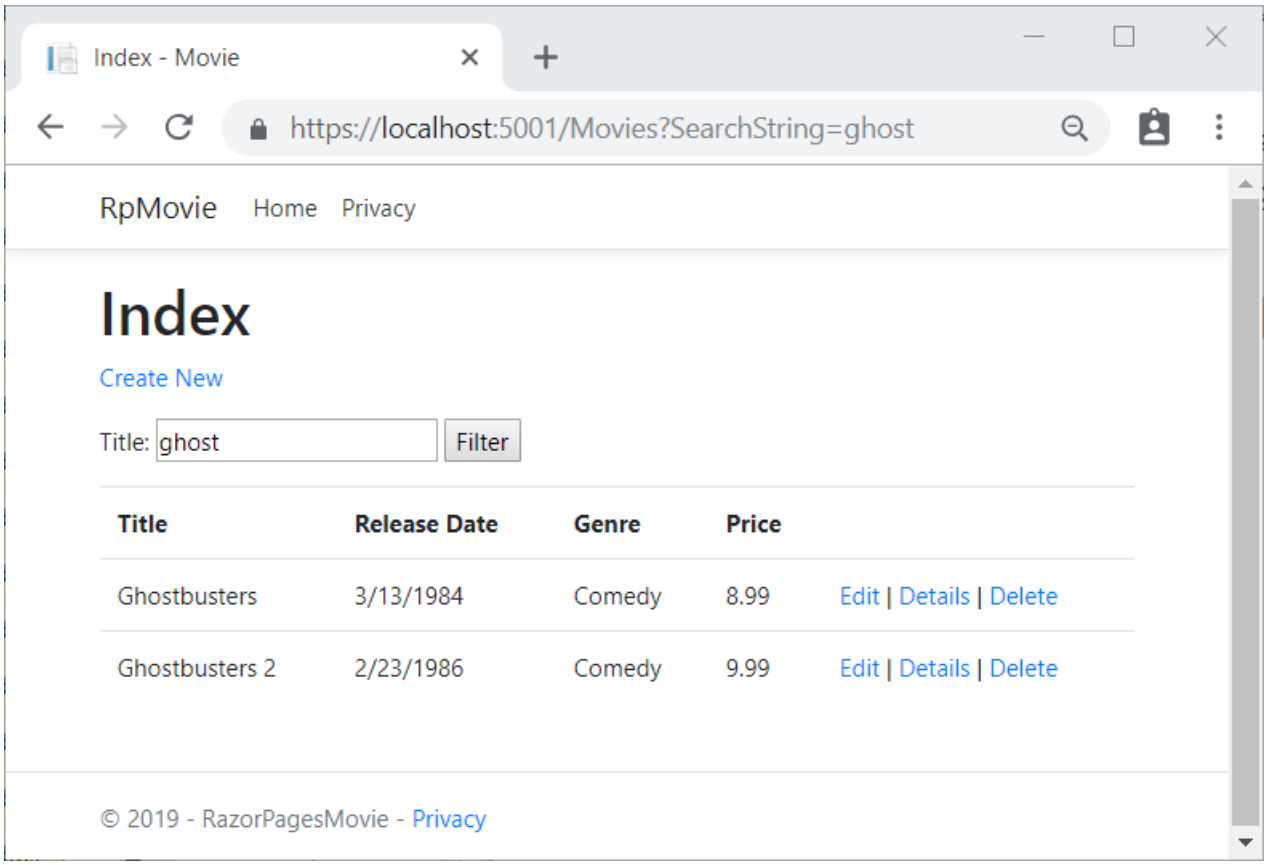
This series of tutorials explains the basics of building a Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages.

This series includes the following tutorials:

1. Create a Razor Pages web app
2. Add a model to a Razor Pages app
3. Scaffold (generate) Razor pages
4. Work with a database
5. Update Razor pages
6. Add search
7. Add a new field
8. Add validation

At the end, you'll have an app that can display and manage a database of movies.



Tutorial: Get started with Razor Pages in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

This is the first tutorial of a series that teaches the basics of building an ASP.NET Core Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages.

At the end of the series, you'll have an app that manages a database of movies.

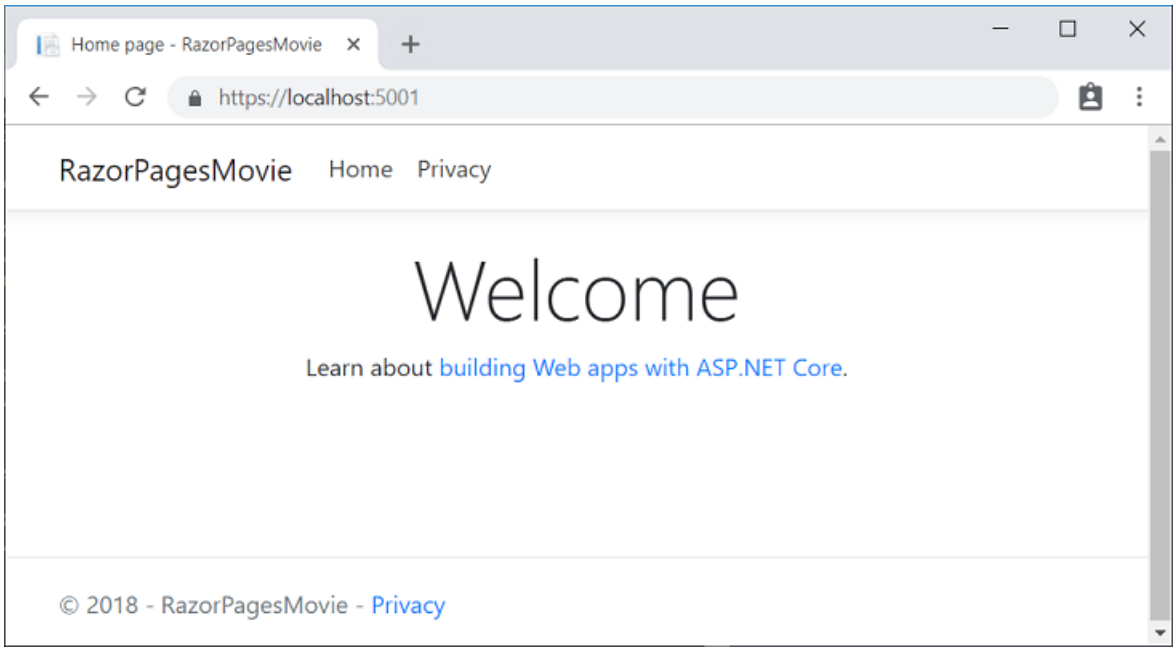
View or download sample code (how to download).

View or download sample code (how to download).

In this tutorial, you:

- Create a Razor Pages web app.
- Run the app.
- Examine the project files.

At the end of this tutorial, you'll have a working Razor Pages web app that you'll build on in later tutorials.



Prerequisites

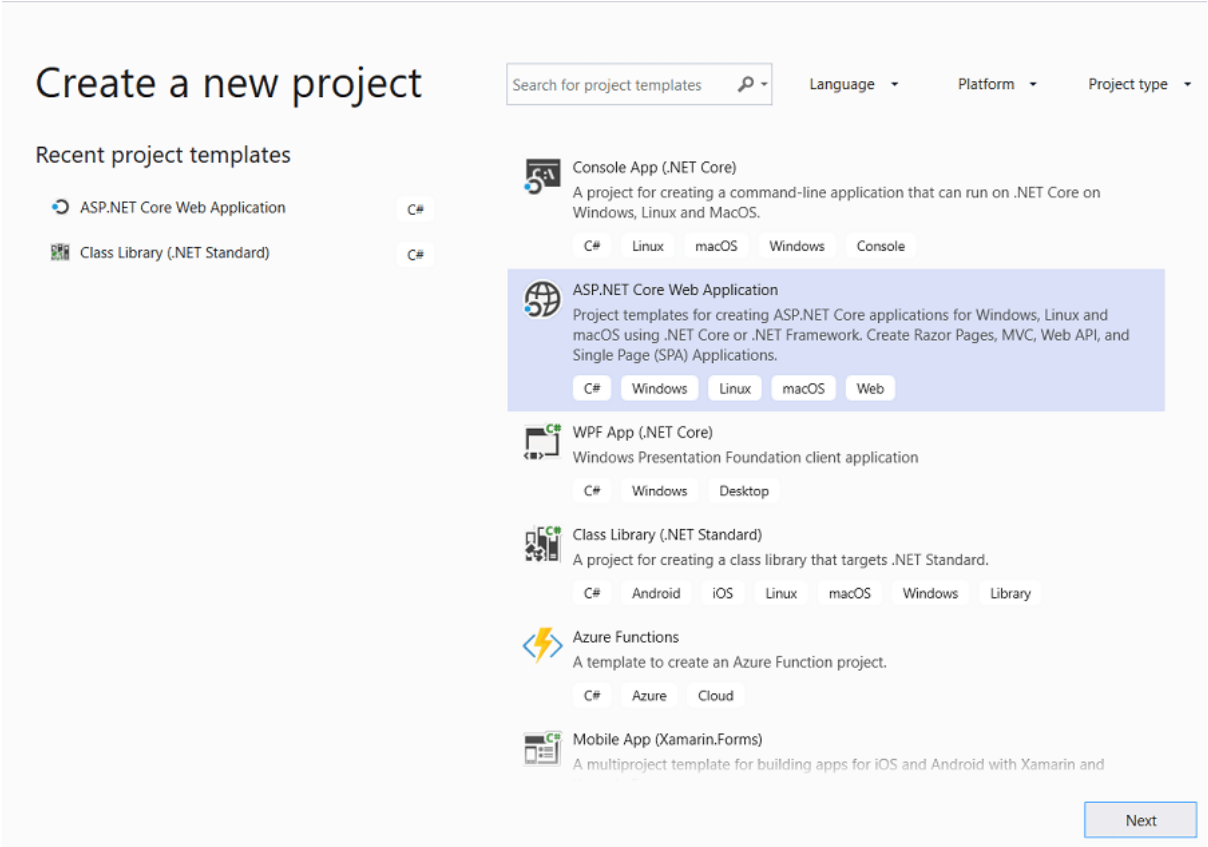
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

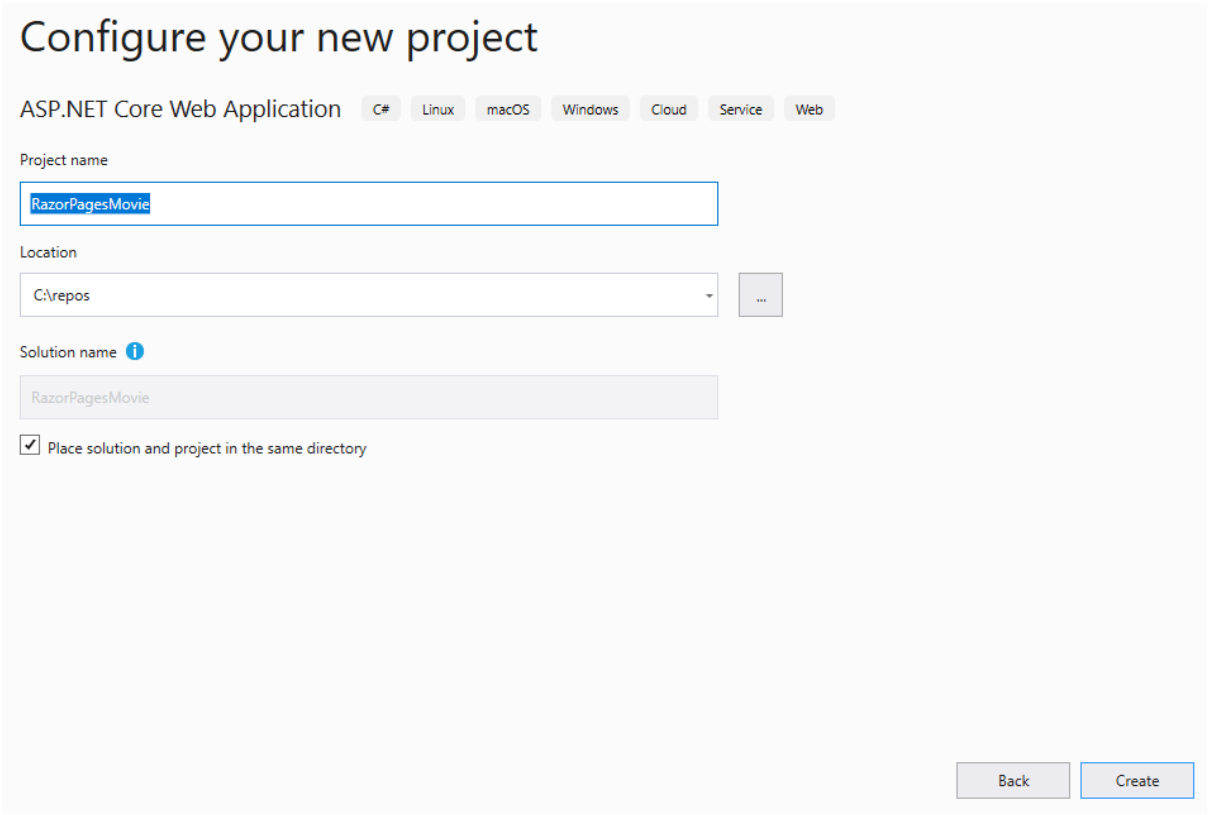
Create a Razor Pages web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

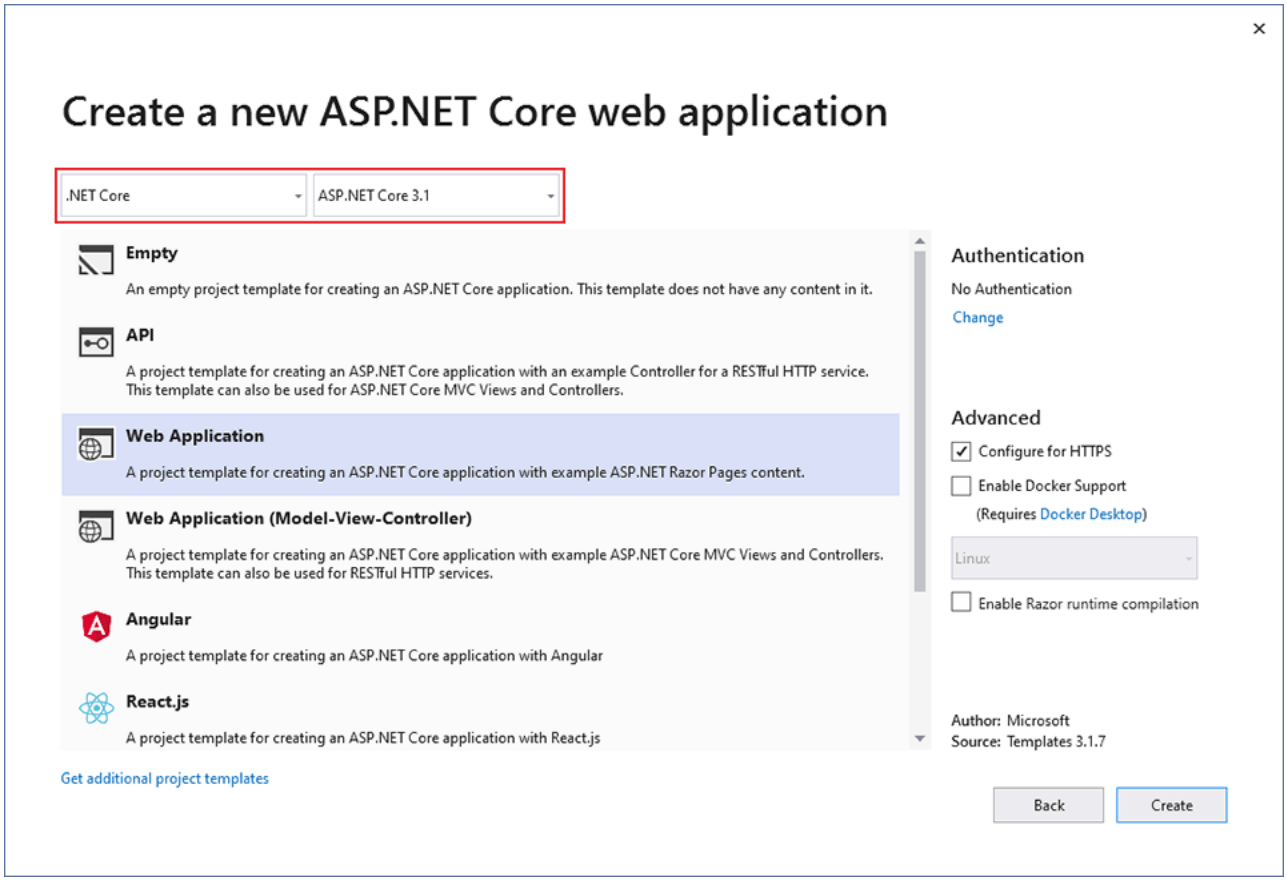
- From the Visual Studio **File** menu, select **New** > **Project**.
- Create a new ASP.NET Core Web Application and select **Next**.



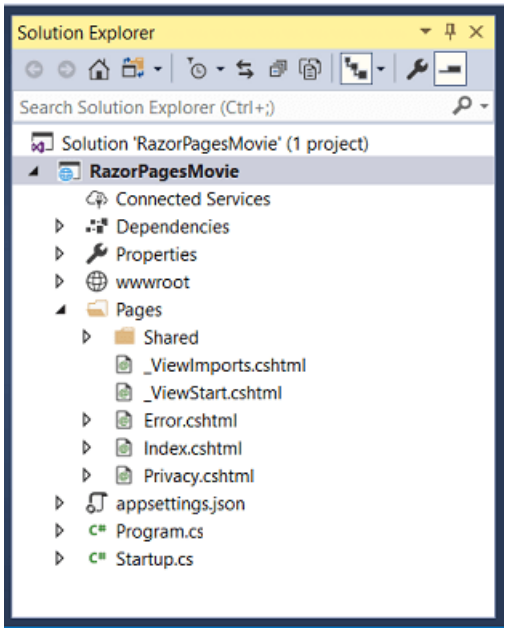
- Name the project **RazorPagesMovie**. It's important to name the project *RazorPagesMovie* so the namespaces will match when you copy and paste code.



- Select **ASP.NET Core 3.1** in the dropdown, **Web Application**, and then select **Create**.



The following starter project is created:

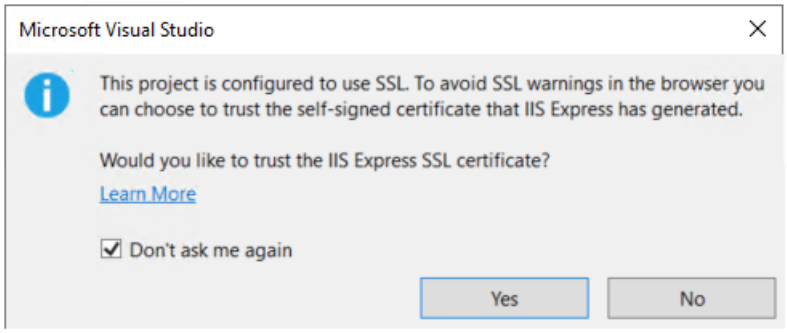


Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

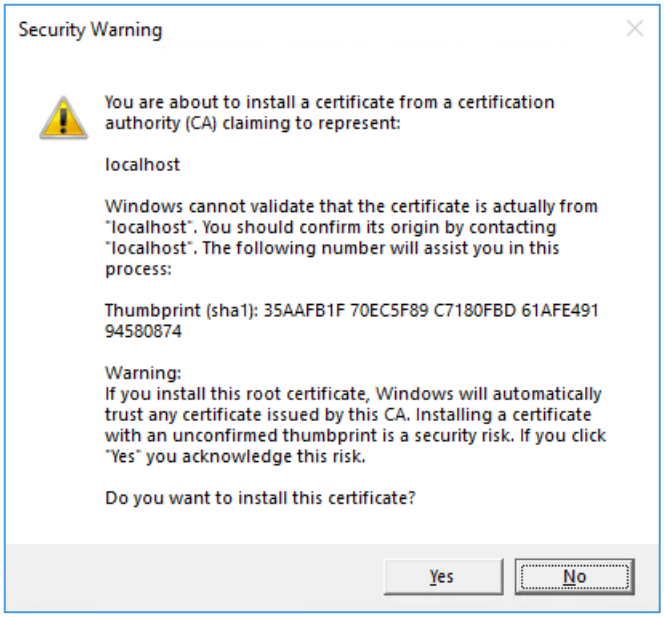
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog:



  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

Visual Studio starts IIS Express and runs the app. The address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for the local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

Examine the project files

Here's an overview of the main project folders and files that you'll work with in later tutorials.

Pages folder

Contains Razor pages and supporting files. Each Razor page is a pair of files:

- A *.cshtml* file that contains HTML markup with C# code using Razor syntax.
- A *.cshtml.cs* file that contains C# code that handles page events.

Supporting files have names that begin with an underscore. For example, the *_Layout.cshtml* file configures UI elements common to all pages. This file sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page. For more information, see Layout in ASP.NET Core.

wwwroot folder

Contains static files, such as HTML files, JavaScript files, and CSS files. For more information, see Static files in ASP.NET Core.

appSettings.json

Contains configuration data, such as connection strings. For more information, see Configuration in ASP.NET Core.

Program.cs

Contains the entry point for the program. For more information, see .NET Generic Host.

Startup.cs

Contains code that configures app behavior. For more information, see App startup in ASP.NET Core.

Next steps

Advance to the next tutorial in the series:

This is the first tutorial of a series. The series teaches the basics of building an ASP.NET Core Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages.

At the end of the series, you'll have an app that manages a database of movies.

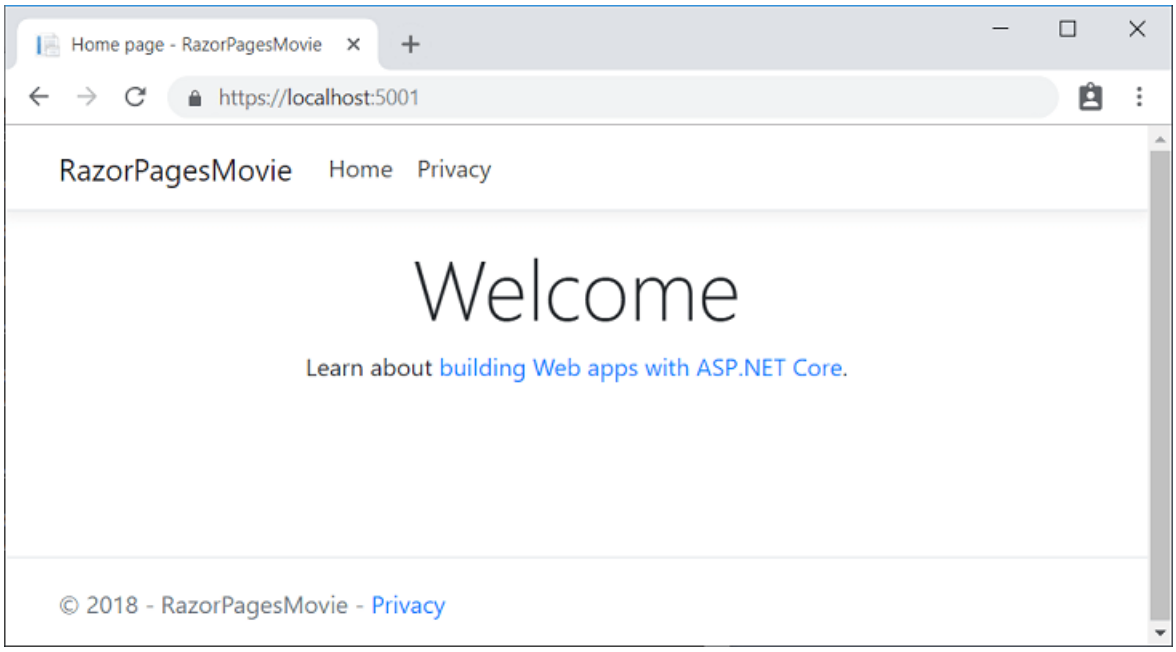
View or download sample code (how to download).

View or download sample code (how to download).

In this tutorial, you:

- Create a Razor Pages web app.
- Run the app.
- Examine the project files.

At the end of this tutorial, you'll have a working Razor Pages web app that you'll build on in later tutorials.



Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

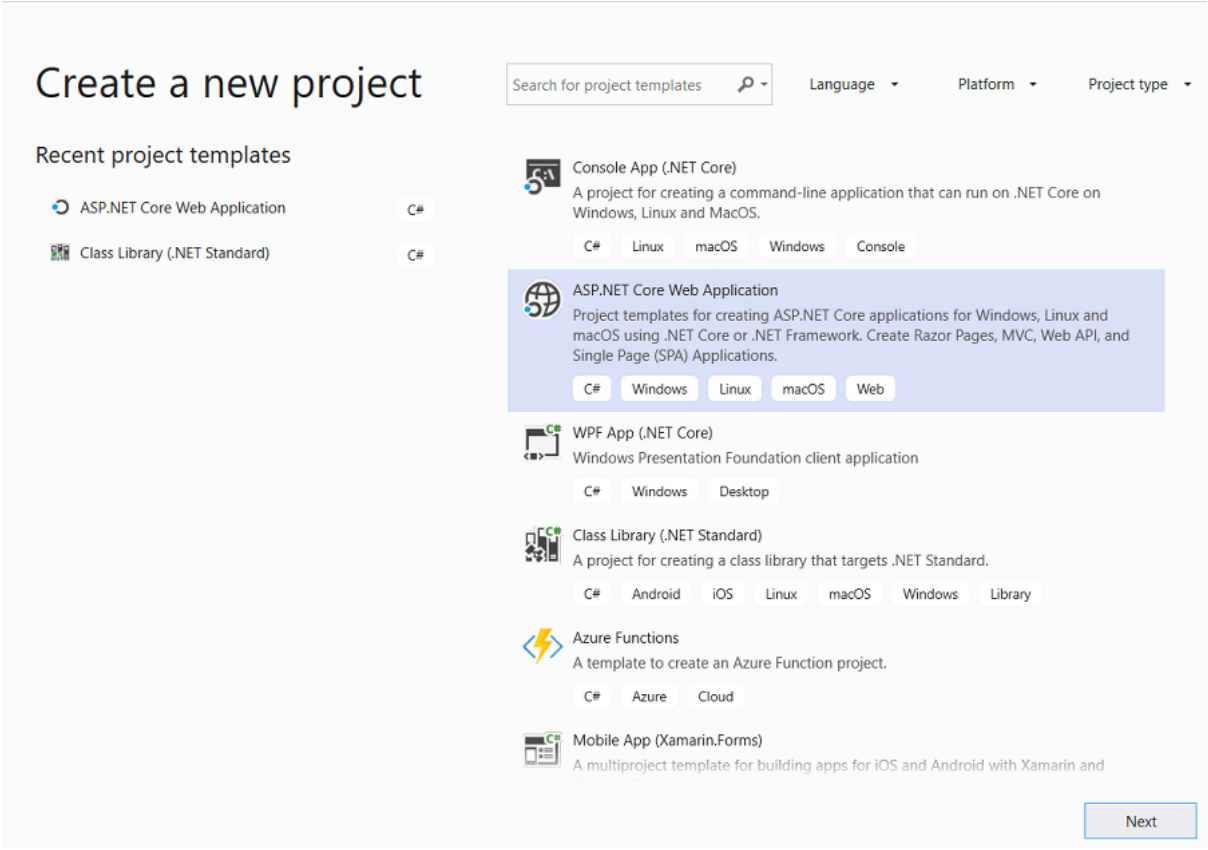
WARNING

If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

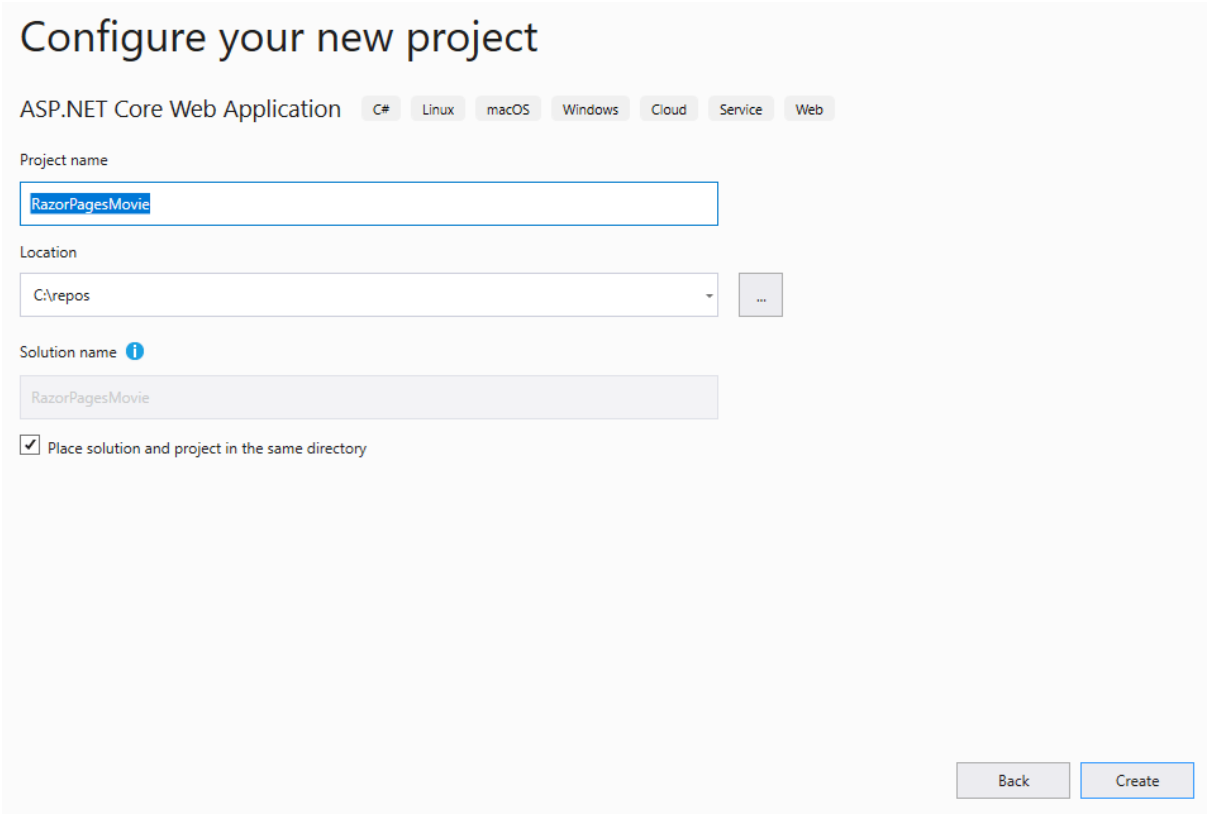
Create a Razor Pages web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

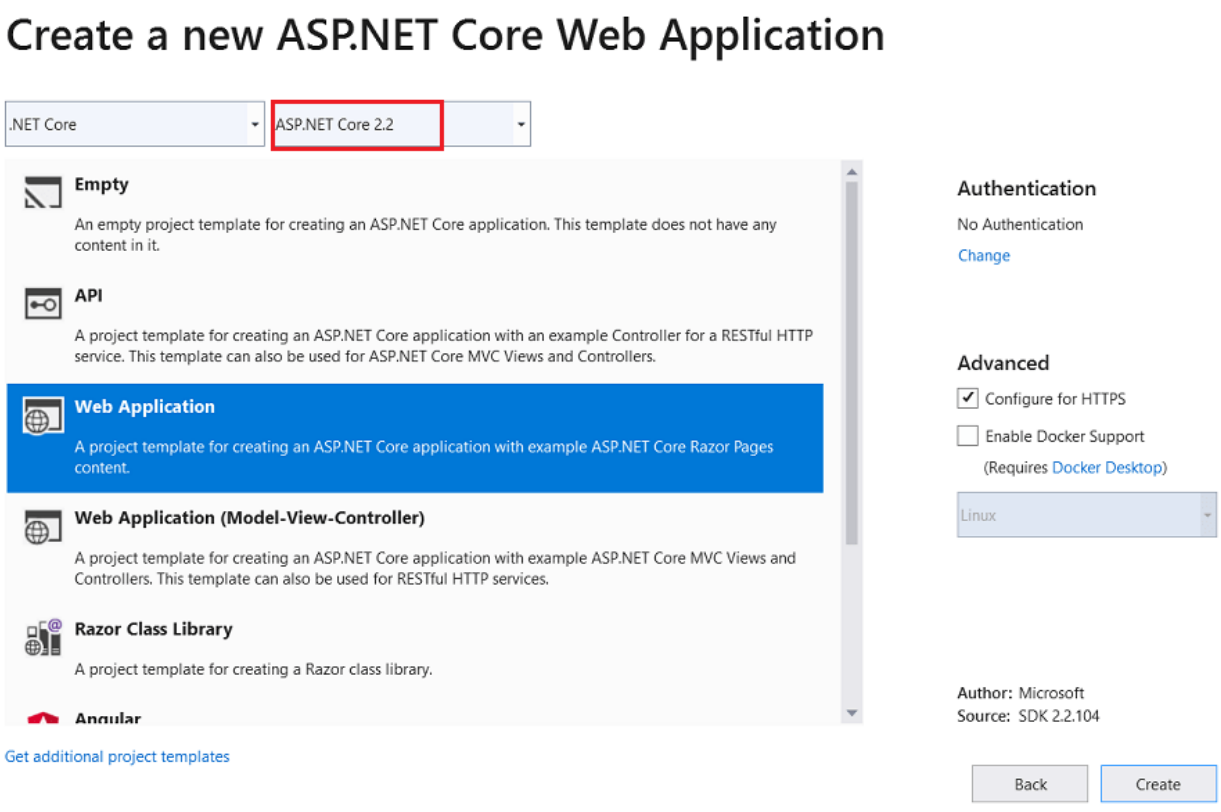
- From the Visual Studio **File** menu, select **New** > **Project**.
- Create a new ASP.NET Core Web Application and select **Next**.



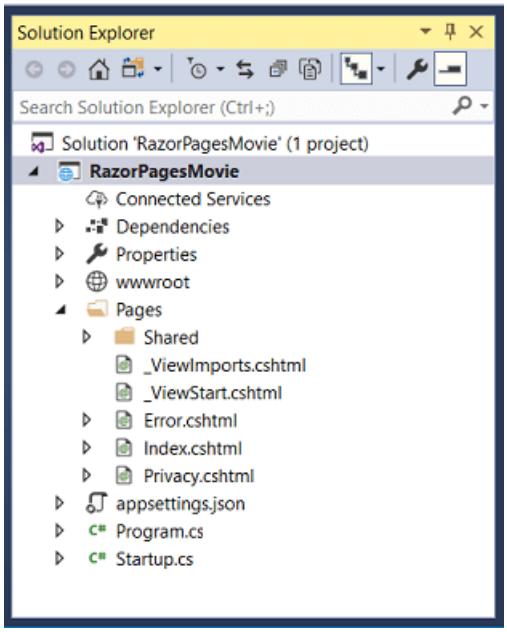
- Name the project **RazorPagesMovie**. It's important to name the project *RazorPagesMovie* so the namespaces will match when you copy and paste code.



- Select **ASP.NET Core 2.2** in the dropdown, **Web Application**, and then select **Create**.



The following starter project is created:

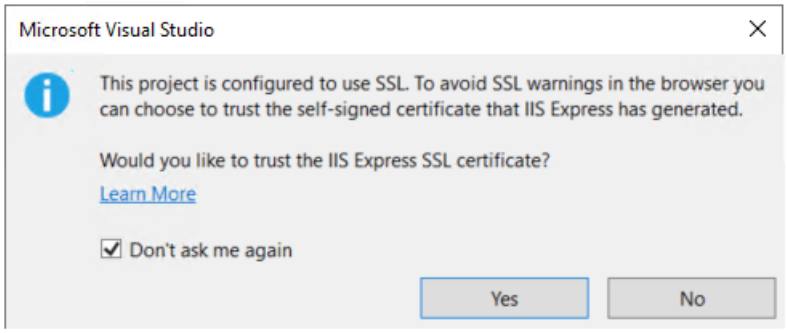


Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

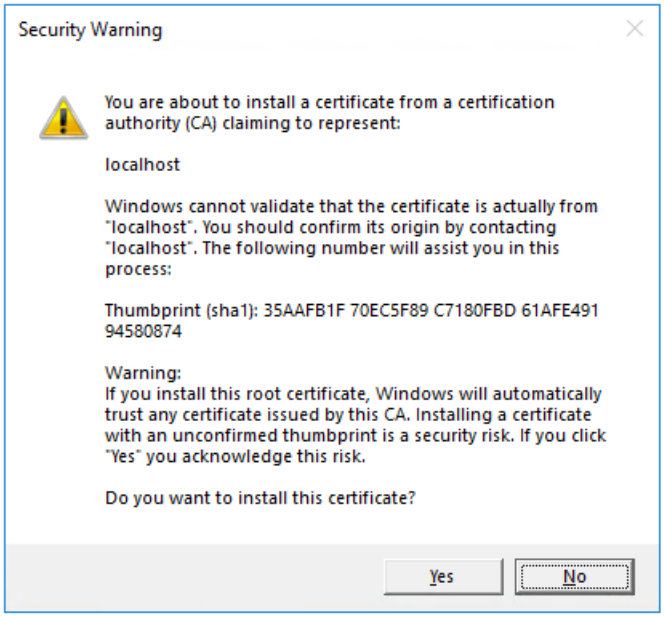
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog:



  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:

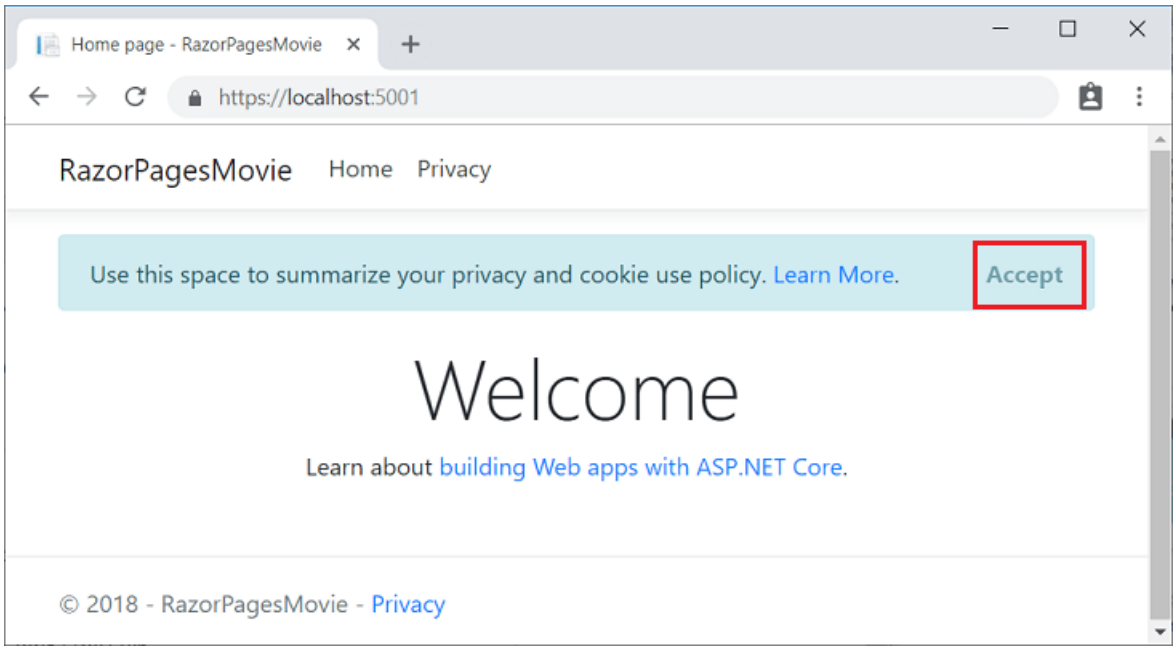


Select **Yes** if you agree to trust the development certificate.

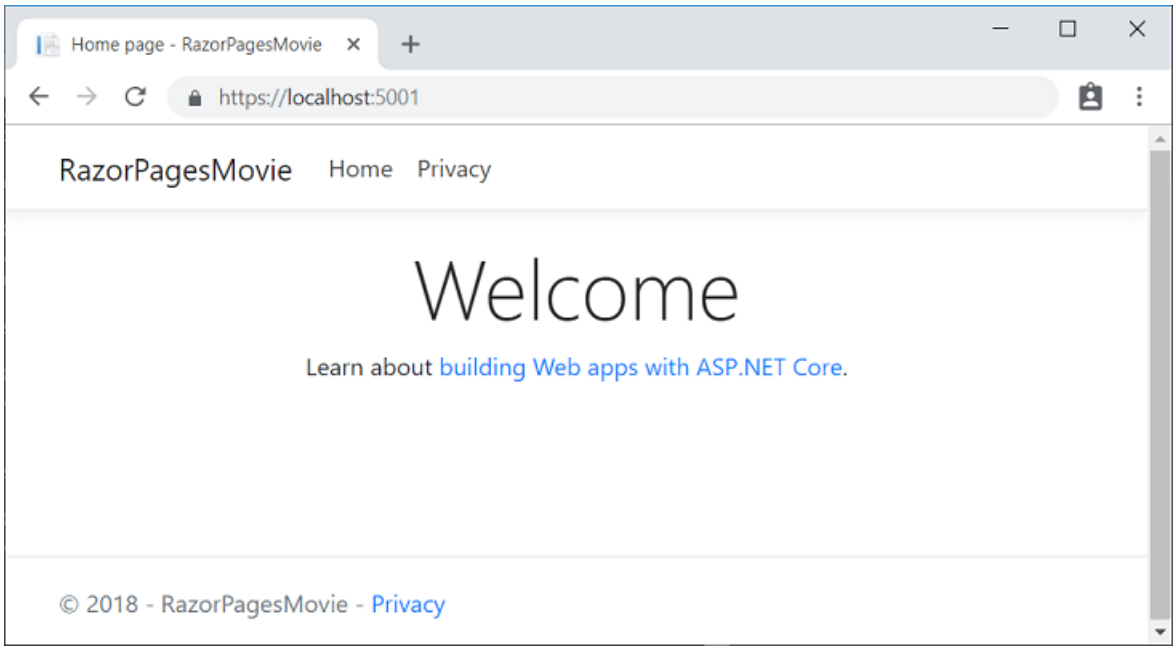
Visual Studio starts IIS Express and runs the app. The address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for the local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

- On the app's home page, select **Accept** to consent to tracking.

  This app doesn't track personal information, but the project template includes the consent feature in case you need it to comply with the European Union's General Data Protection Regulation (GDPR).



The following image shows the app after you give consent to tracking:



Examine the project files

Here's an overview of the main project folders and files that you'll work with in later tutorials.

Pages folder

Contains Razor pages and supporting files. Each Razor page is a pair of files:

- A *.cshtml* file that contains HTML markup with C# code using Razor syntax.
- A *.cshtml.cs* file that contains C# code that handles page events.

Supporting files have names that begin with an underscore. For example, the *_Layout.cshtml* file configures UI elements common to all pages. This file sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page. For more information, see Layout in ASP.NET Core.

wwwroot folder

Contains static files, such as HTML files, JavaScript files, and CSS files. For more information, see Static files in ASP.NET Core.

appSettings.json

Contains configuration data, such as connection strings. For more information, see Configuration in ASP.NET Core.

Program.cs

Contains the entry point for the program. For more information, see .NET Generic Host.

Startup.cs

Contains code that configures app behavior, such as whether it requires consent for cookies. For more information, see App startup in ASP.NET Core.

Additional resources

- Youtube version of this tutorial

Next steps

Advance to the next tutorial in the series:

ADD A MODEL

Part 2, add a model to a Razor Pages app in ASP.NET Core

9/22/2020 • 22 minutes to read • Edit Online

By Rick Anderson

In this section, classes are added for managing movies. The app's model classes use Entity Framework Core (EF Core) to work with the database. EF Core is an object-relational mapper (O/RM) that simplifies data access.

The model classes are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They define the properties of the data that are stored in the database.

View or download sample code (how to download).

View or download sample code (how to download).

Add a data model

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Right-click the **RazorPagesMovie** project > **Add** > **New Folder**. Name the folder *Models*.

Right click the *Models* folder. Select **Add** > **Class**. Name the class **Movie**.

Add the following properties to the `Movie` class:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains:

- The `ID` field is required by the database for the primary key.
- `[DataType(DataType.Date)]`: The DataType attribute specifies the type of the data (Date). With this attribute:
  - The user is not required to enter time information in the date field.
  - Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

Build the project to verify there are no compilation errors.

Scaffold the movie model

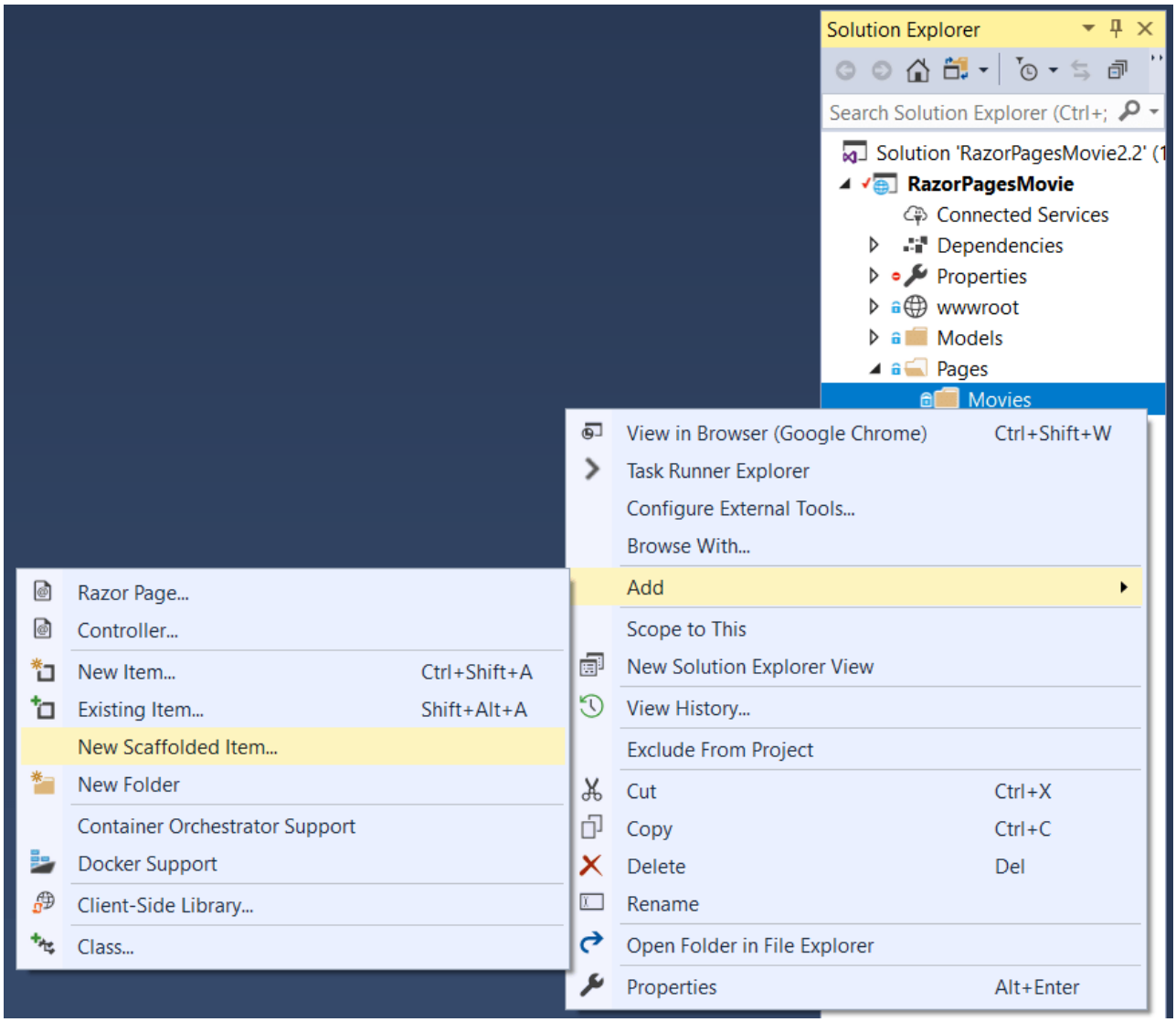
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

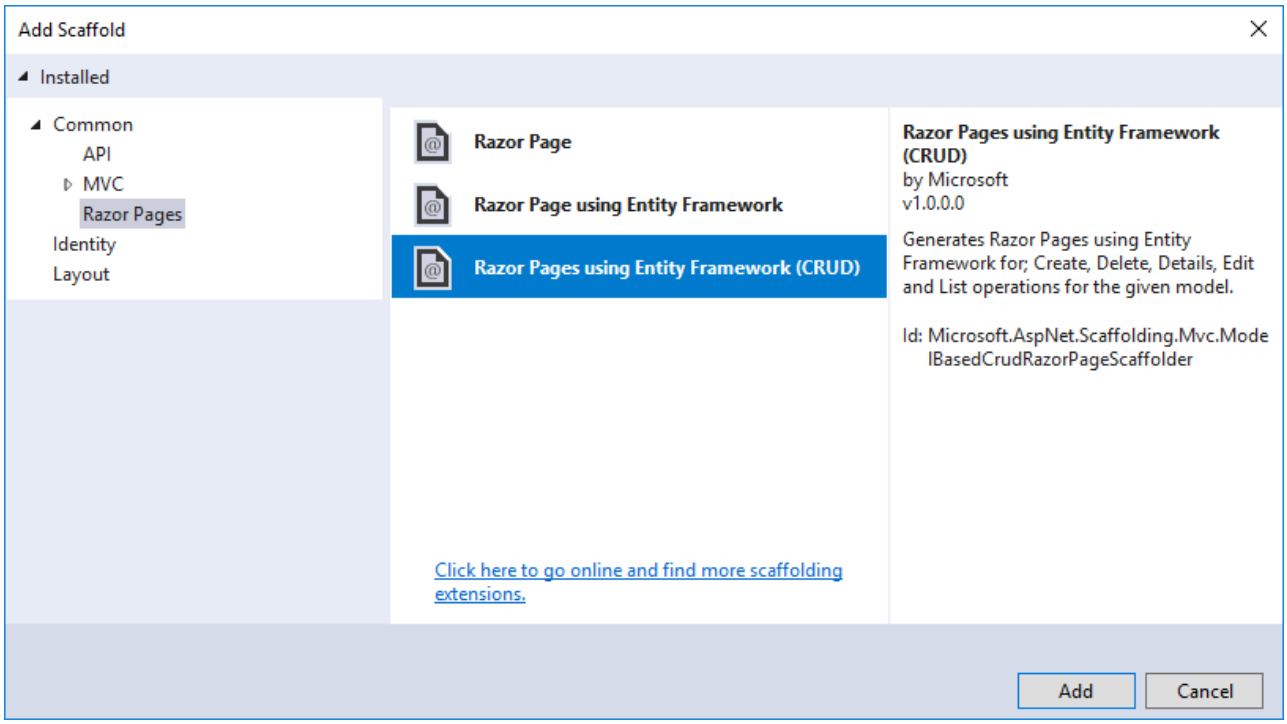
Create a *Pages/Movies* folder:

- Right click on the *Pages* folder > **Add** > **New Folder**.
- Name the folder *Movies*

Right click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.

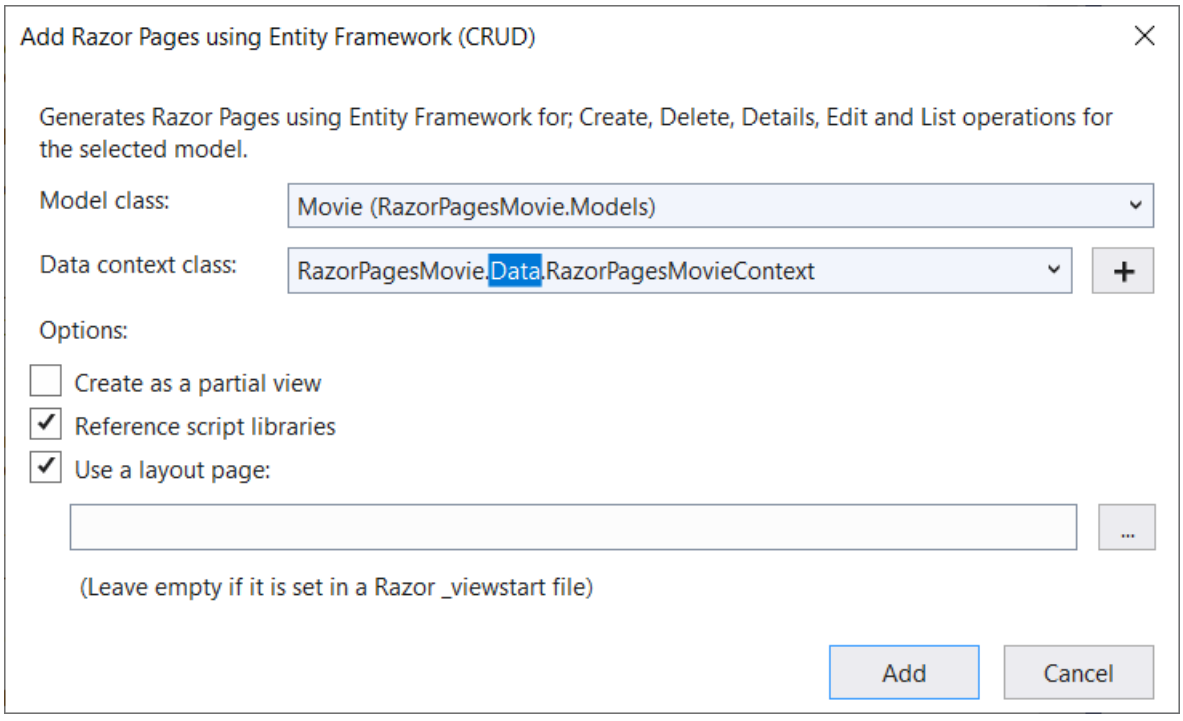


In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **Add**.



Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:

- In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.
- In the **Data context class** row, select the **+** (plus) sign and change the generated name from RazorPagesMovie.**Models**.RazorPagesMovieContext to RazorPagesMovie.**Data**.RazorPagesMovieContext. This change is not required. It creates the database context class with the correct namespace.
- Select **Add**.



The *appsettings.json* file is updated with the connection string used to connect to a local database.

Files created

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

The scaffold process creates and updates the following files:

- *Pages/Movies*: Create, Delete, Details, Edit, and Index.
- *Data/RazorPagesMovieContext.cs*

Updated

- *Startup.cs*

The created and updated files are explained in the next section.

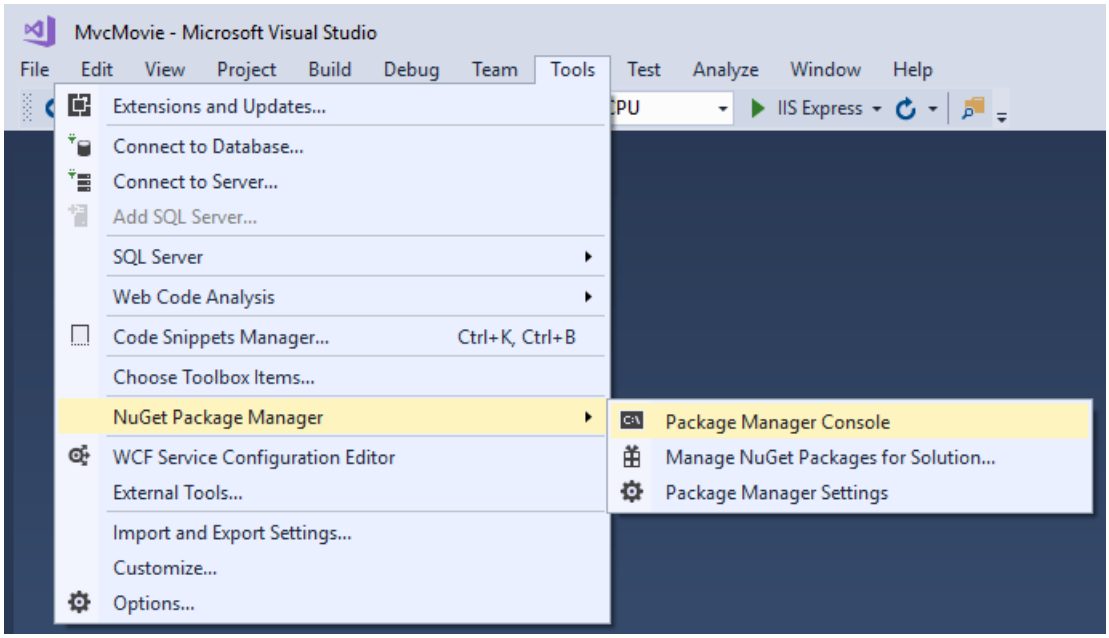
Initial migration

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In this section, the Package Manager Console (PMC) is used to:

- Add an initial migration.
- Update the database with the initial migration.

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration InitialCreate
Update-Database
```

The preceding commands generate the following warning: "No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'."

You can ignore that warning, it will be fixed in a later tutorial.

The migrations command generates code to create the initial database schema. The schema is based on the model specified in `DbContext`. The `InitialCreate` argument is used to name the migrations. Any name can be used, but by convention a name is selected that describes the migration.

The `update` command runs the `Up` method in migrations that have not been applied. In this case, `update` runs the `Up` method in *Migrations/<time-stamp>_InitialCreate.cs* file, which creates the database.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

The scaffolding tool automatically created a DB context and registered it with the dependency injection container.

Examine the `Startup.ConfigureServices` method. The highlighted line was added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

The `RazorPagesMovieContext` coordinates EF Core functionality (Create, Read, Update, Delete, etc.) for the `Movie` model. The data context (`RazorPagesMovieContext`) is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model.

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Test the app

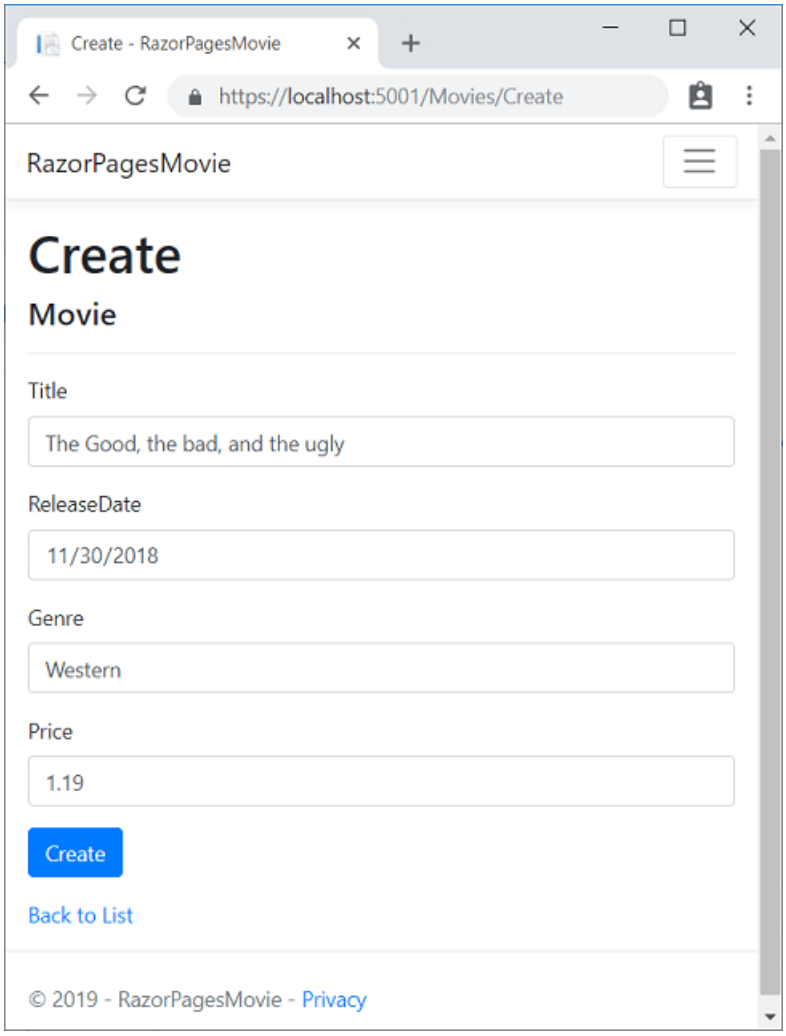
- Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

If you get the error:

```
SqlException: Cannot open database "RazorPagesMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

- Test the **Create** link.



> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** links.

The next tutorial explains the files created by scaffolding.

Additional resources

PREVIOUS: GET STARTED
NEXT: SCAFFOLDED RAZOR PAGES

In this section, classes are added for managing movies in a cross-platform SQLite database. Apps created from an ASP.NET Core template use a SQLite database. The app's model classes are used with Entity Framework Core (EF Core) (SQLite EF Core Database Provider) to work with the database. EF Core is an object-relational mapping (ORM) framework that simplifies data access.

The model classes are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They define the properties of the data that are stored in the database.

View or download sample code (how to download).

View or download sample code (how to download).

Add a data model

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Right-click the **RazorPagesMovie** project > **Add** > **New Folder**. Name the folder *Models*.

Right click the *Models* folder. Select **Add** > **Class**. Name the class **Movie**.

Add the following properties to the `Movie` class:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains:

- The `ID` field is required by the database for the primary key.
- `[DataType(DataType.Date)]`: The DataType attribute specifies the type of the data (Date). With this attribute:
  - The user is not required to enter time information in the date field.
  - Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

Build the project to verify there are no compilation errors.

Scaffold the movie model

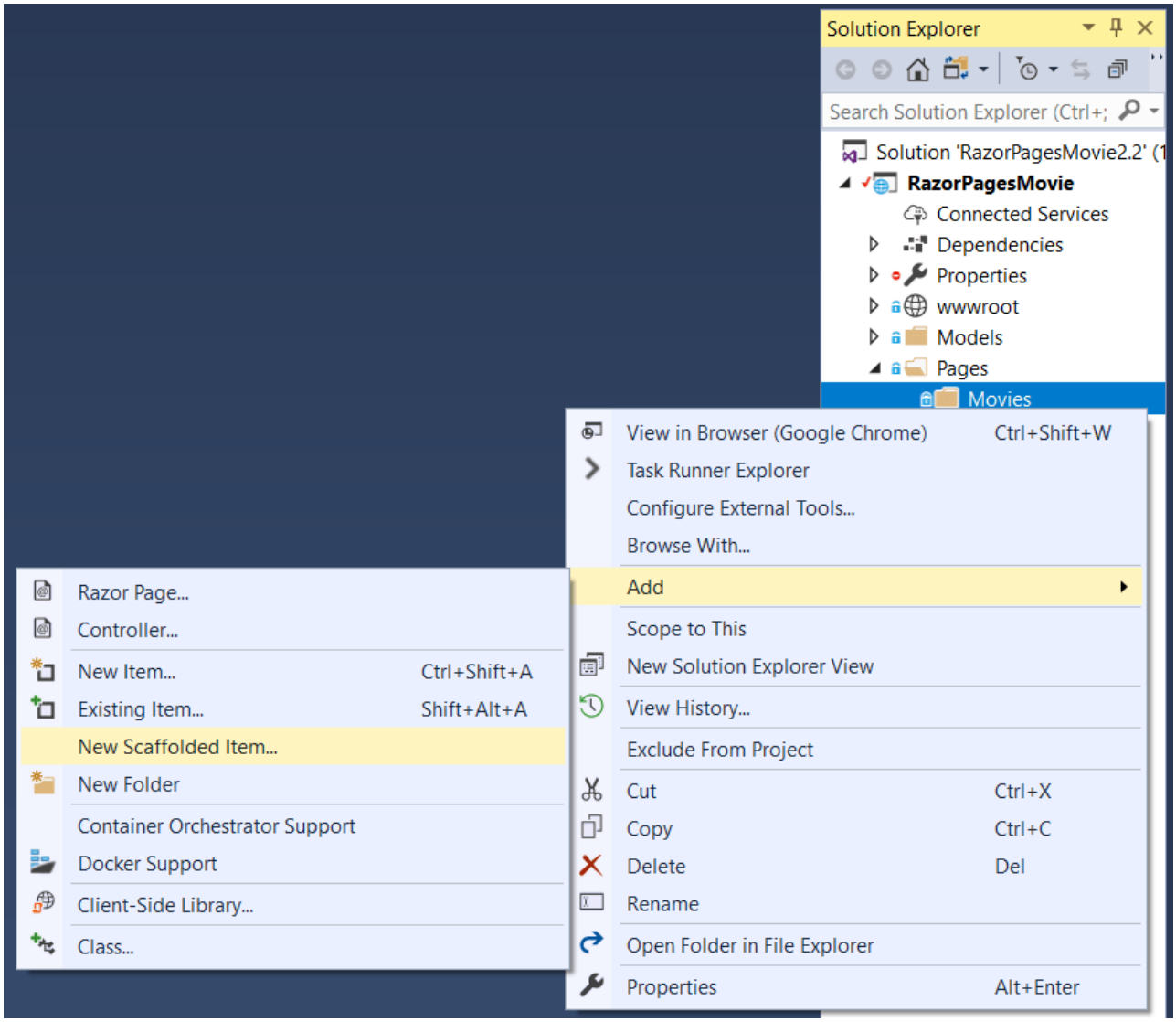
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

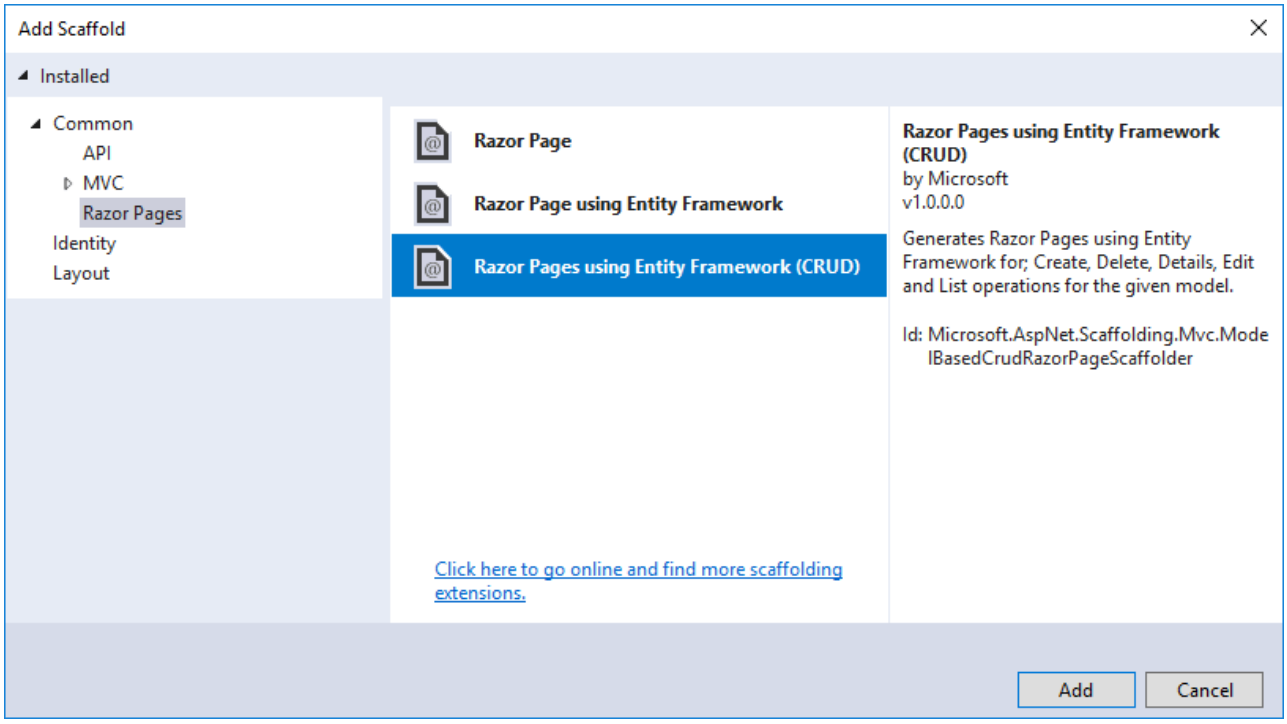
Create a *Pages/Movies* folder:

- Right click on the *Pages* folder > **Add** > **New Folder**.
- Name the folder *Movies*

Right click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.

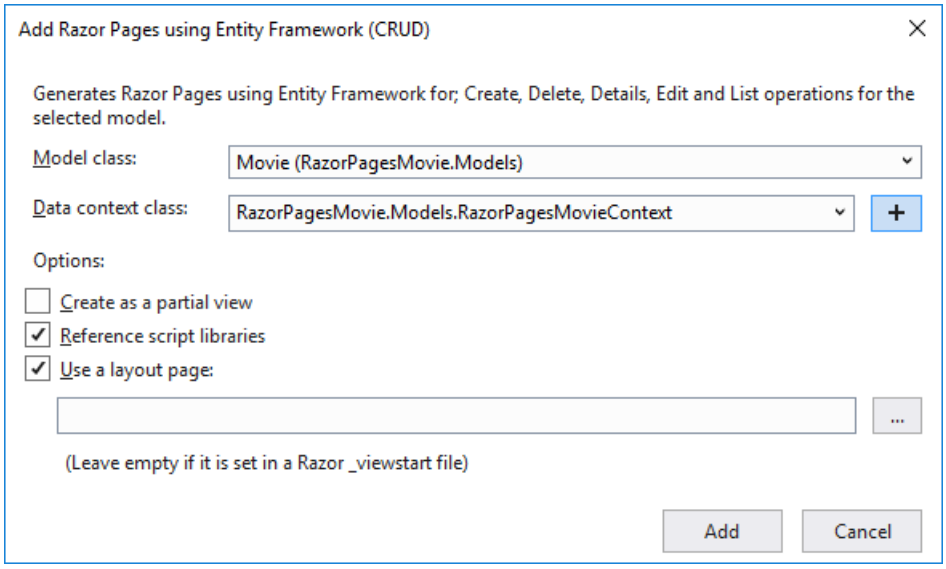


In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **Add**.



Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:

- In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.
- In the **Data context class** row, select the **+** (plus) sign and accept the generated name **RazorPagesMovie.Models.RazorPagesMovieContext**.
- Select **Add**.



The *appsettings.json* file is updated with the connection string used to connect to a local database.

The scaffold process creates and updates the following files:

Files created

- *Pages/Movies*: Create, Delete, Details, Edit, and Index.
- *Data/RazorPagesMovieContext.cs*

File updated

- *Startup.cs*

The created and updated files are explained in the next section.

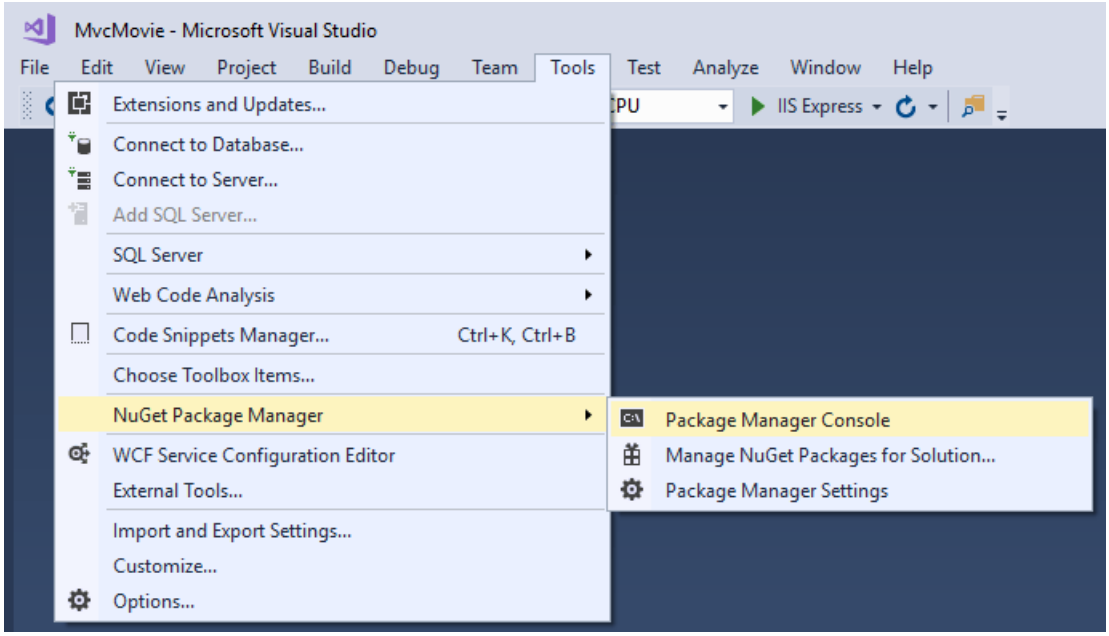
Initial migration

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In this section, the Package Manager Console (PMC) is used to:

- Add an initial migration.
- Update the database with the initial migration.

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration Initial
Update-Database
```

The `Add-Migration` command generates code to create the initial database schema. The schema is based on the model specified in the `DbContext` (In the *RazorPagesMovieContext.cs* file). The `InitialCreate` argument is used to name the migration. Any name can be used, but by convention a name that describes the migration is used. For more information, see Tutorial: Using the migrations feature - ASP.NET MVC with EF Core.

The `Update-Database` command runs the `Up` method in the *Migrations/<time-stamp>_InitialCreate.cs* file. The `Up` method creates the database.

> **NOTE**
>
> The preceding commands generate the following warning: "*No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.*" You can ignore that warning, it will be fixed in a later tutorial.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

The scaffolding tool automatically created a DB context and registered it with the dependency injection container.

Examine the `Startup.ConfigureServices` method. The highlighted line was added by the scaffolder:

```
// This method gets called by the runtime.
// Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is
        // needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

The `RazorPagesMovieContext` coordinates EF Core functionality (Create, Read, Update, Delete, etc.) for the `Movie` model. The data context (`RazorPagesMovieContext`) is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model.

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Test the app

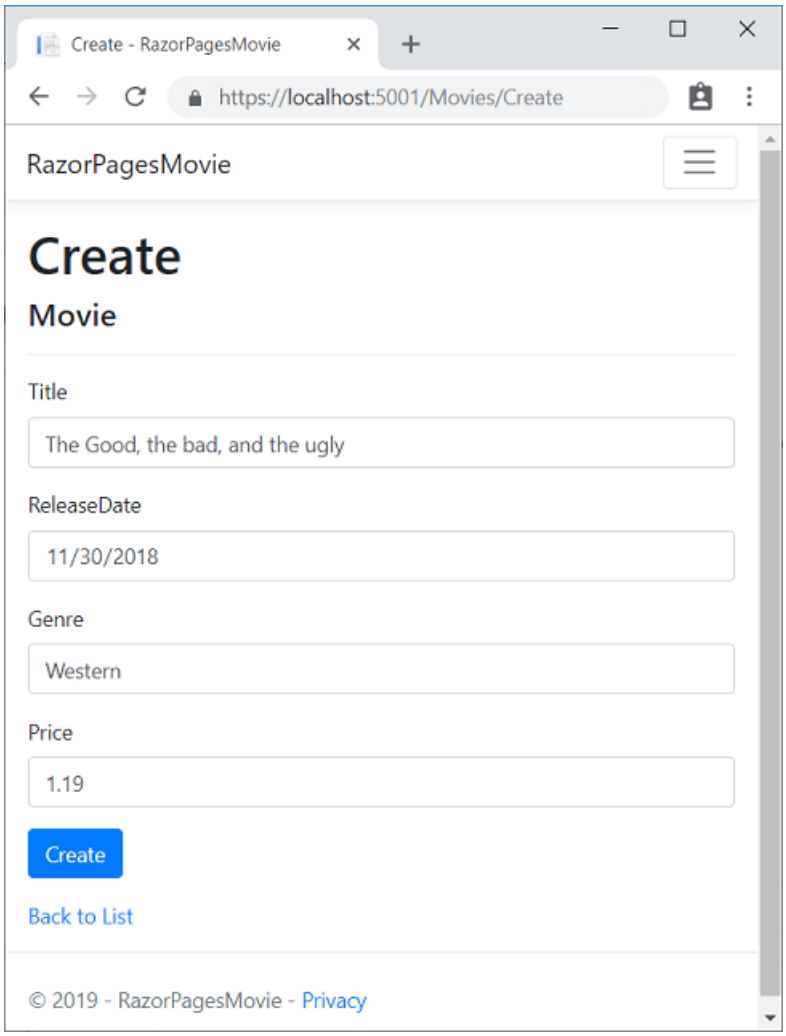
- Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

If you get the error:

```
SqlException: Cannot open database "RazorPagesMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

- Test the **Create** link.



> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** links.

The next tutorial explains the files created by scaffolding.

Additional resources

PREVIOUS: GET STARTED
NEXT: SCAFFOLDED RAZOR PAGES

Part 3, scaffolded Razor Pages in ASP.NET Core

9/22/2020 • 16 minutes to read • Edit Online

By Rick Anderson

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial.

View or download sample code (how to download).

View or download sample code (how to download).

The Create, Delete, Details, and Edit pages

Examine the *Pages/Movies/Index.cshtml.cs* Page Model:

```
// Unused usings removed.
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

        public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get;set; }

        public async Task OnGetAsync()
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor Pages are derived from `PageModel`. By convention, the `PageModel`-derived class is called `<PageName>Model`. The constructor uses dependency injection to add the `RazorPagesMovieContext` to the page. All the scaffolded pages follow this pattern. See Asynchronous code for more information on asynchronous programming with Entity Framework.

When a request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. `OnGetAsync` or `OnGet` is called to initialize the state of the page. In this case, `OnGetAsync` gets a list of movies and displays them.

When `OnGet` returns `void` or `OnGetAsync` returns `Task`, no return statement is used. When the return type is `IActionResult` or `Task<IActionResult>`, a return statement must be provided. For example, the *Pages/Movies/Create.cshtml.cs* `OnPostAsync` method:

```
    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Movie.Add(Movie);
        await _context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Examine the *Pages/Movies/Index.cshtml* Razor Page:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup, otherwise it transitions into C#.

The @page directive

The `@page` Razor directive makes the file an MVC action, which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` is an example of transitioning into Razor-specific markup. See Razor syntax for more information.

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movie[0].Title)
```

The `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` is `null` or empty. When the lambda expression is evaluated (for example, with `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

The @model directive

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel`-derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayFor` HTML Helpers on the page.

The layout page

Select the menu links (**RazorPagesMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Pages/Shared/_Layout.cshtml* file. Open the *Pages/Shared/_Layout.cshtml* file.

Layout templates allow the HTML container layout to be:

- Specified in one place.
- Applied in multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the page-specific views show up, *wrapped* in the layout page. For example, select the **Privacy** link and the *Pages/Privacy.cshtml* view is rendered inside the `RenderBody` method.

ViewData and layout

Consider the following markup from the *Pages/Movies/Index.cshtml* file:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

The preceding highlighted markup is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class contains a `ViewData` dictionary property that can be used to pass data to a View. Objects are added to the `ViewData` dictionary using a key/value pattern. In the preceding sample, the `"Title"` property is added to the `ViewData` dictionary.

The `"Title"` property is used in the *Pages/Shared/_Layout.cshtml* file. The following markup shows the first few lines of the *_Layout.cshtml* file.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>

    @*Markup removed for brevity.*@
```

The line `@*Markup removed for brevity.*@` is a Razor comment. Unlike HTML comments (`<!-- -->`), Razor comments are not sent to the client.

Update the layout

Change the `<title>` element in the *Pages/Shared/_Layout.cshtml* file to display **Movie** rather than **RazorPagesMovie**.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie</title>
```

Find the following anchor element in the *Pages/Shared/_Layout.cshtml* file.

```
<a class="navbar-brand" asp-area="" asp-page="/Index">RazorPagesMovie</a>
```

Replace the preceding element with the following markup:

```
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

The preceding anchor element is a Tag Helper. In this case, it's the Anchor Tag Helper. The `asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page. The `asp-area` attribute value is empty, so the area isn't used in the link. See Areas for more information.

Save your changes, and test the app by clicking on the **RpMovie** link. See the _Layout.cshtml file in GitHub if you have any problems.

Test the other links (**Home**, **RpMovie**, **Create**, **Edit**, and **Delete**). Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark.

> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

The `Layout` property is set in the *Pages/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The preceding markup sets the layout file to *Pages/Shared/_Layout.cshtml* for all Razor files under the *Pages* folder.

See Layout for more information.

The Create page model

Examine the *Pages/Movies/Create.cshtml.cs* page model:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

        public CreateModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize, so `Page` is returned. Later in the tutorial, an example of `OnGet` initializing state is shown. The `Page` method creates a `PageResult` object that renders the *Create.cshtml* page.

The `Movie` property uses the `[BindProperty]` attribute to opt-in to model binding. When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date. Client-side validation and model validation are discussed later in the tutorial.

If there are no model errors, the data is saved, and the browser is redirected to the Index page.

The Create Razor Page

Examine the *Pages/Movies/Create.cshtml* Razor Page file:

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

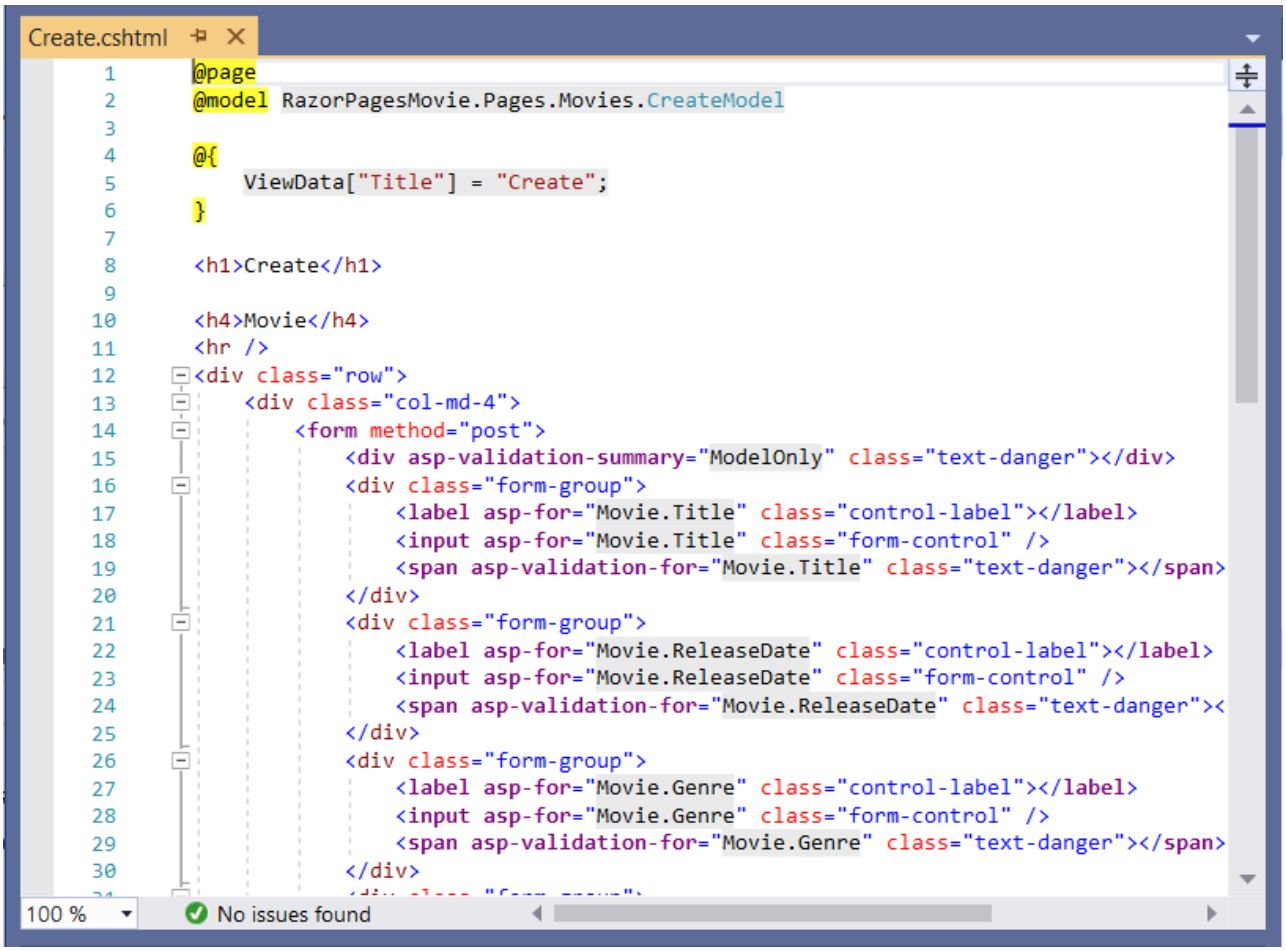
<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio displays the following tags in a distinctive bold font used for Tag Helpers:

- `<form method="post">`
- `<div asp-validation-summary="ModelOnly" class="text-danger"></div>`
- `<label asp-for="Movie.Title" class="control-label"></label>`
- `<input asp-for="Movie.Title" class="form-control" />`
- `<span asp-validation-for="Movie.Title" class="text-danger"></span>`



The `<form method="post">` element is a Form Tag Helper. The Form Tag Helper automatically includes an antiforgery token.

The scaffolding engine creates Razor markup for each field in the model (except the ID) similar to the following:

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

The Validation Tag Helpers (`<div asp-validation-summary` and `<span asp-validation-for`) display validation errors. Validation is covered in more detail later in this series.

The Label Tag Helper (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `for` attribute for the `Title` property.

The Input Tag Helper (`<input asp-for="Movie.Title" class="form-control">`) uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side.

For more information on Tag Helpers such as `<form method="post">`, see Tag Helpers in ASP.NET Core.

Additional resources

PREVIOUS: ADDING A MODEL
NEXT: DATABASE

By Rick Anderson

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial.

View or download sample.

The Create, Delete, Details, and Edit pages

Examine the *Pages/Movies/Index.cshtml.cs* Page Model:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public IndexModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get;set; }

        public async Task OnGetAsync()
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor Pages are derived from `PageModel`. By convention, the `PageModel`-derived class is called `<PageName>Model`. The constructor uses dependency injection to add the `RazorPagesMovieContext` to the page. All the scaffolded pages follow this pattern. See Asynchronous code for more information on asynchronous programming with Entity Framework.

When a request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. `OnGetAsync` or `OnGet` is called on a Razor Page to initialize the state for the page. In this case, `OnGetAsync` gets a list of movies and displays them.

When `OnGet` returns `void` or `OnGetAsync` returns `Task`, no return method is used. When the return type is `IActionResult` or `Task<IActionResult>`, a return statement must be provided. For example, the *Pages/Movies/Create.cshtml.cs* `OnPostAsync` method:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Examine the *Pages/Movies/Index.cshtml* Razor Page:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup, otherwise it transitions into C#.

The `@page` Razor directive makes the file into an MVC action, which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` is an example of transitioning into Razor-specific markup. See Razor syntax for more information.

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movie[0].Title)
```

The `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` are `null` or empty. When the lambda expression is evaluated (for example, with `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

The @model directive

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel`-derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayFor` HTML Helpers on the page.

The layout page

Select the menu links (**RazorPagesMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Pages/Shared/_Layout.cshtml* file. Open the *Pages/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the page-specific views you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the **Pages/Privacy.cshtml** view is rendered inside the `RenderBody` method.

ViewData and layout

Consider the following code from the *Pages/Movies/Index.cshtml* file:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

The preceding highlighted code is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class has a `ViewData` dictionary property that can be used to add data that you want to pass to a View. You add objects into the `ViewData` dictionary using a key/value pattern. In the preceding sample, the "Title" property is added to the `ViewData` dictionary.

The "Title" property is used in the *Pages/Shared/_Layout.cshtml* file. The following markup shows the first few lines of the *_Layout.cshtml* file.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>

    @*Markup removed for brevity.*@
```

The line `@*Markup removed for brevity.*@` is a Razor comment which doesn't appear in your layout file. Unlike

HTML comments (`<!-- -->`), Razor comments are not sent to the client.

Update the layout

Change the `<title>` element in the *Pages/Shared/_Layout.cshtml* file to display **Movie** rather than **RazorPagesMovie**.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie</title>
```

Find the following anchor element in the *Pages/Shared/_Layout.cshtml* file.

```
<a class="navbar-brand" asp-area="" asp-page="/Index">RazorPagesMovie</a>
```

Replace the preceding element with the following markup.

```
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

The preceding anchor element is a Tag Helper. In this case, it's the Anchor Tag Helper. The `asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page. The `asp-area` attribute value is empty, so the area isn't used in the link. See Areas for more information.

Save your changes, and test the app by clicking on the **RpMovie** link. See the _Layout.cshtml file in GitHub if you have any problems.

Test the other links (**Home**, **RpMovie**, **Create**, **Edit**, and **Delete**). Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark.

> **NOTE**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. This GitHub issue 4076 for instructions on adding decimal comma.

The `Layout` property is set in the *Pages/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The preceding markup sets the layout file to *Pages/Shared/_Layout.cshtml* for all Razor files under the *Pages* folder. See Layout for more information.

The Create page model

Examine the *Pages/Movies/Create.cshtml.cs* page model:

```
// Unused usings removed.
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public CreateModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize, so `Page` is returned. Later in the tutorial you see `OnGet` method initialize state. The `Page` method creates a `PageResult` object that renders the *Create.cshtml* page.

The `Movie` property uses the `[BindProperty]` attribute to opt-in to model binding. When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date. Client-side validation and model validation are discussed later in the tutorial.

If there are no model errors, the data is saved, and the browser is redirected to the Index page.

The Create Razor Page

Examine the *Pages/Movies/Create.cshtml* Razor Page file:

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

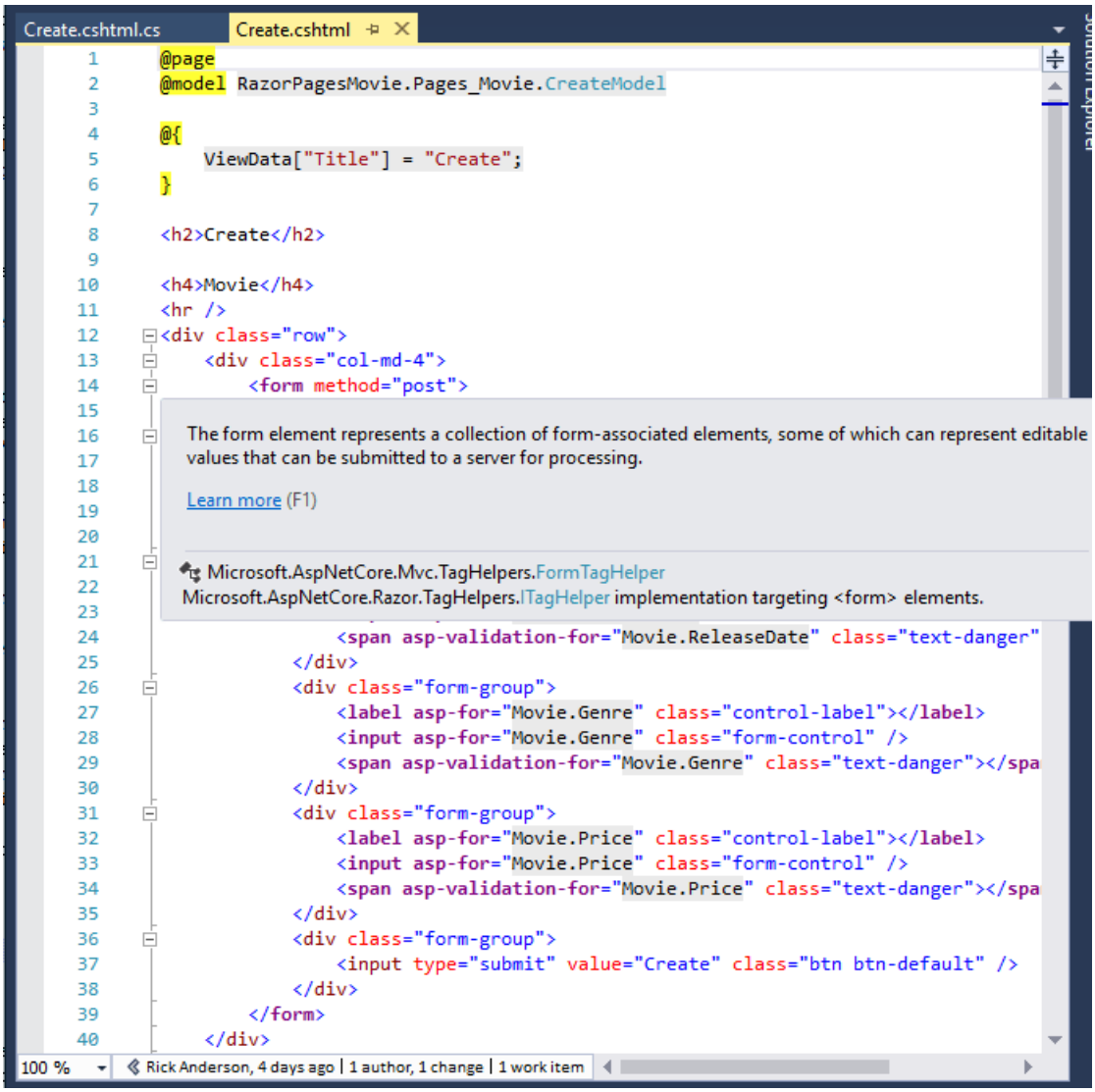
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio displays the `<form method="post">` tag in a distinctive bold font used for Tag Helpers:



The `<form method="post">` element is a Form Tag Helper. The Form Tag Helper automatically includes an antiforgery token.

The scaffolding engine creates Razor markup for each field in the model (except the ID) similar to the following:

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

The Validation Tag Helpers (`<div asp-validation-summary` and `<span asp-validation-for`) display validation errors. Validation is covered in more detail later in this series.

The Label Tag Helper (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `for` attribute for the `Title` property.

The Input Tag Helper (`<input asp-for="Movie.Title" class="form-control">`) uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side.

Additional resources

- YouTube version of this tutorial

PREVIOUS: ADDING A MODEL

NEXT: DATABASE

Part 4, with a database and ASP.NET Core

9/22/2020 • 12 minutes to read • Edit Online

By Rick Anderson and Joe Audette

View or download sample code (how to download).

View or download sample code (how to download).

The `RazorPagesMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in *Startup.cs*:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

The name value for the database (`Database={Database name}`) will be different for your generated code. The name value is arbitrary.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
bc;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

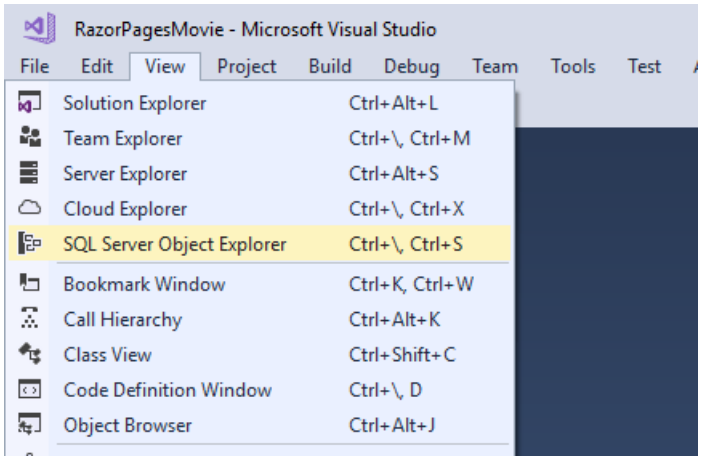
When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a real database server. See Configuration for more information.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

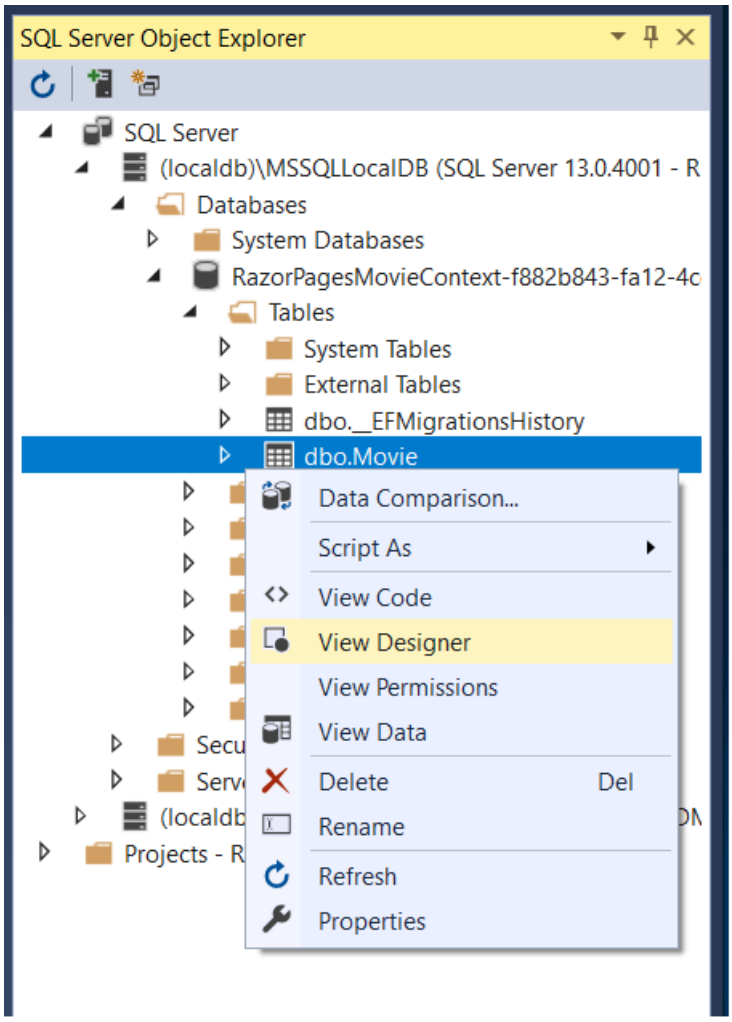
SQL Server Express LocalDB

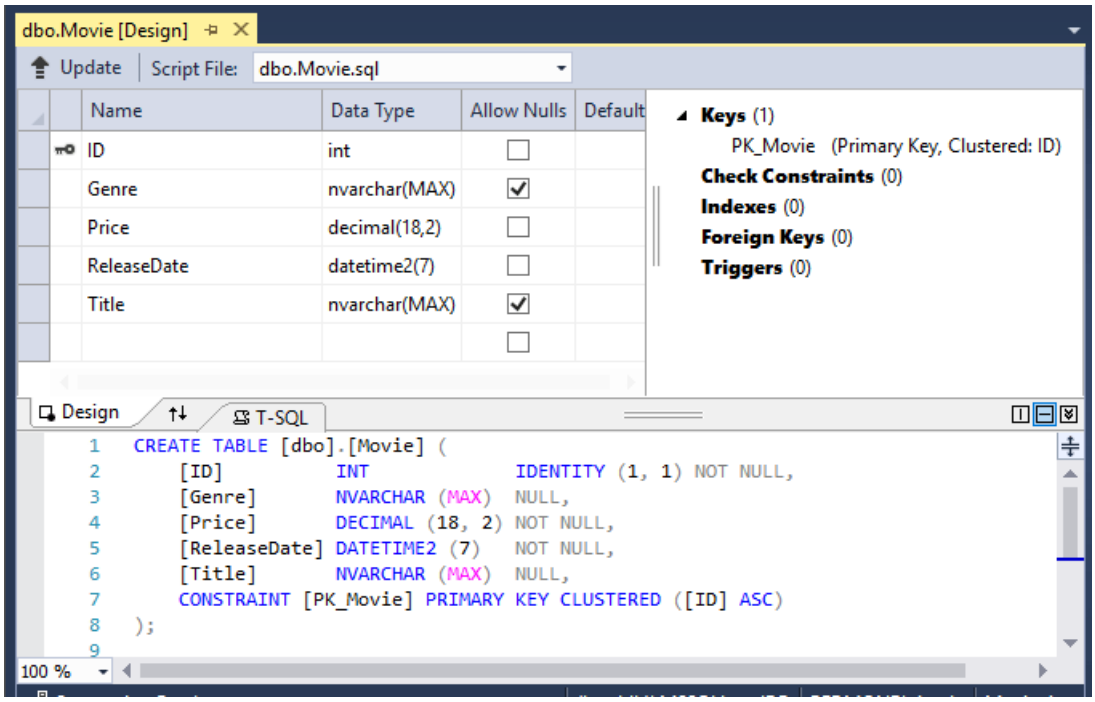
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:\Users\<user>\` directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



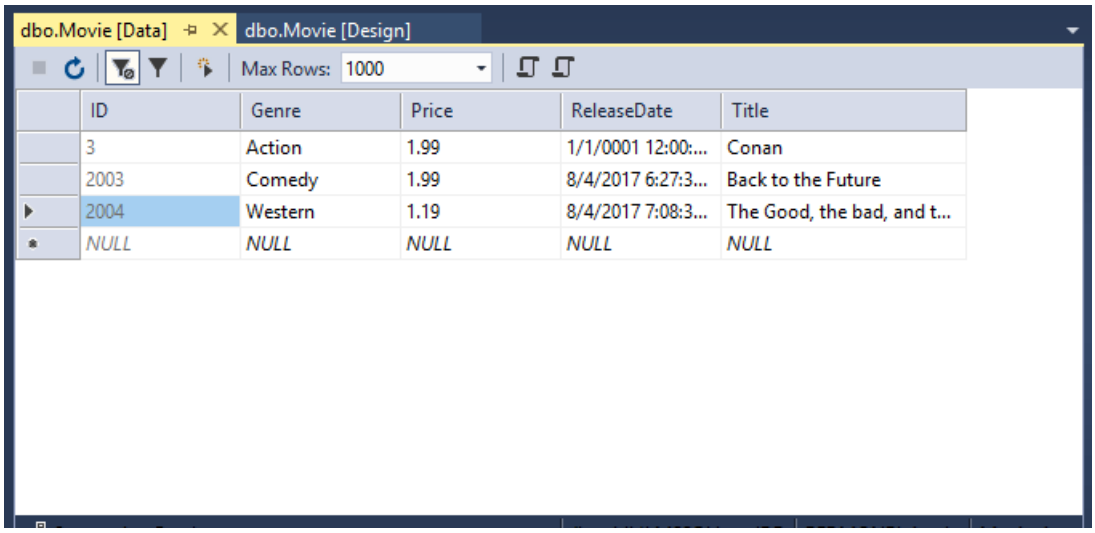
- Right click on the `Movie` table and select **View Designer**:





Note the key icon next to `ID`. By default, EF creates a property named `ID` for the primary key.

- Right click on the `Movie` table and select **View Data**:



Seed the database

Create a new class named `SeedData` in the *Models* folder with the following code:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new RazorPagesMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<RazorPagesMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

In *Program.cs*, modify the `Main` method to do the following:

- Get a DB context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method completes.

The following code shows the updated *Program.cs* file.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();

        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

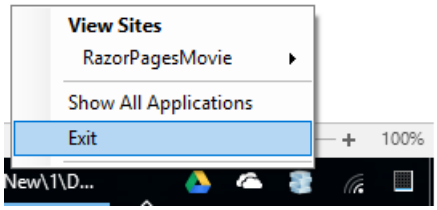
The following exception occurs when `Update-Database` has not been run:

```
SqlException: Cannot open database "RazorPagesMovieContext-" requested by the login. The login failed.
Login failed for user 'user name'.
```

Test the app

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:
  - Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**:



  - If you were running VS in non-debug mode, press F5 to run in debug mode.
  - If you were running VS in debug mode, stop the debugger and press F5.

The next tutorial will improve the presentation of the data.

Additional resources

PREVIOUS: SCAFFOLDED RAZOR PAGES

NEXT: UPDATING THE PAGES

View or download sample code (how to download).

View or download sample code (how to download).

The `RazorPagesMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in *Startup.cs*:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
// This method gets called by the runtime.
// Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is
        // needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

For more information on the methods used in `ConfigureServices`, see:

- EU General Data Protection Regulation (GDPR) support in ASP.NET Core for `CookiePolicyOptions`.
- SetCompatibilityVersion

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

The name value for the database (`Database={Database name}`) will be different for your generated code. The name value is arbitrary.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
1234;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

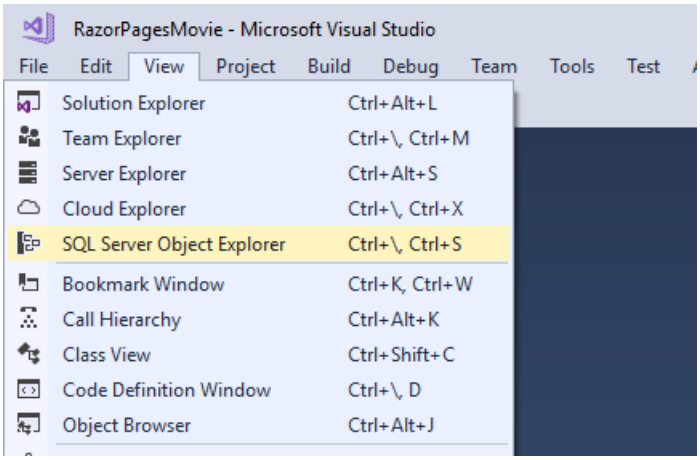
When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a real database server. See Configuration for more information.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

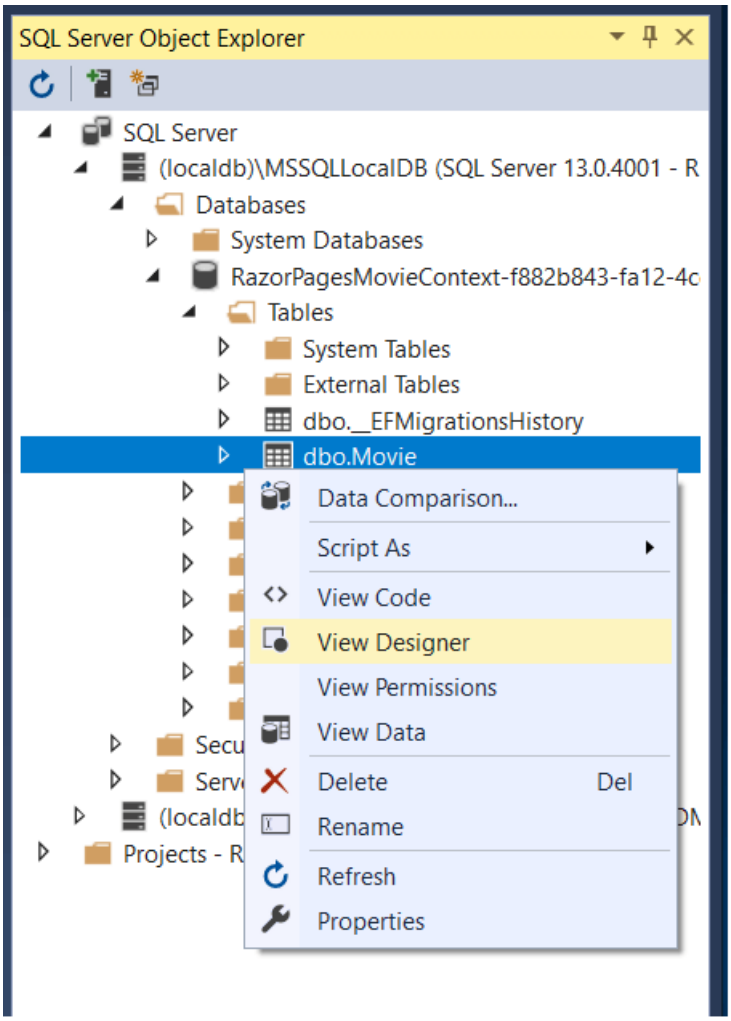
SQL Server Express LocalDB

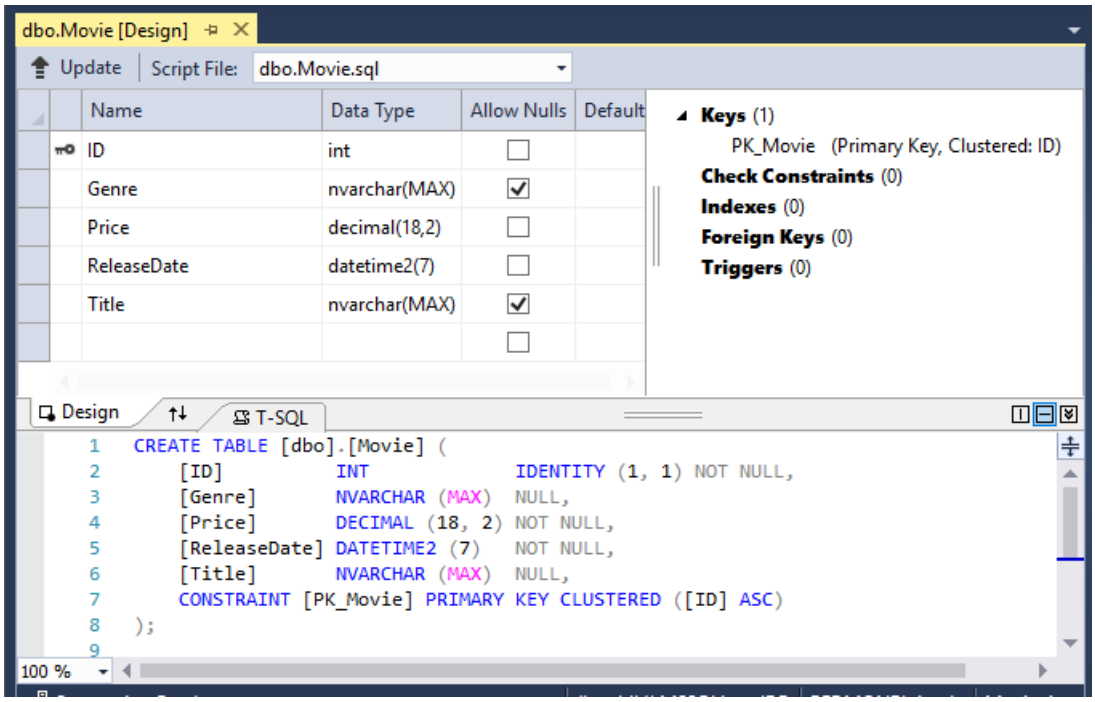
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:/Users/<user/>` directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



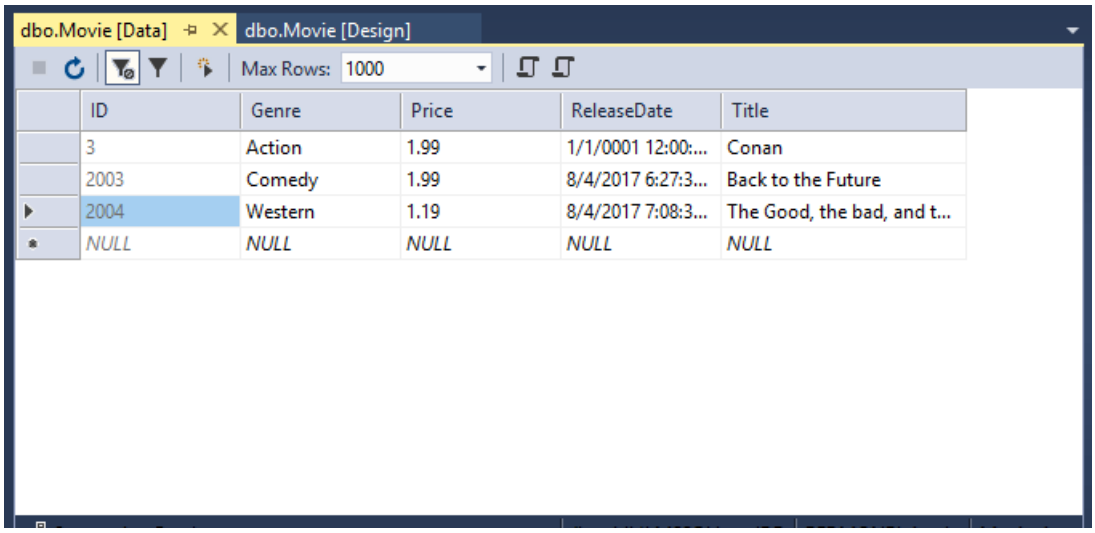
- Right click on the `Movie` table and select **View Designer**:





Note the key icon next to `ID`. By default, EF creates a property named `ID` for the primary key.

- Right click on the `Movie` table and select **View Data**:



Seed the database

Create a new class named `SeedData` in the *Models* folder with the following code:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new RazorPagesMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<RazorPagesMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

In *Program.cs*, modify the `Main` method to do the following:

- Get a DB context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method completes.

The following code shows the updated *Program.cs* file.

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context=services.
                        GetRequiredService<RazorPagesMovieContext>();
                    context.Database.Migrate();
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

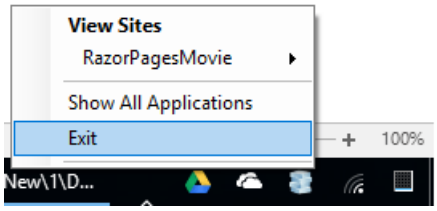
A production app would not call `Database.Migrate`. It's added to the preceding code to prevent the following exception when `Update-Database` has not been run:

SqlException: Cannot open database "RazorPagesMovieContext-21" requested by the login. The login failed. Login failed for user 'user name'.

Test the app

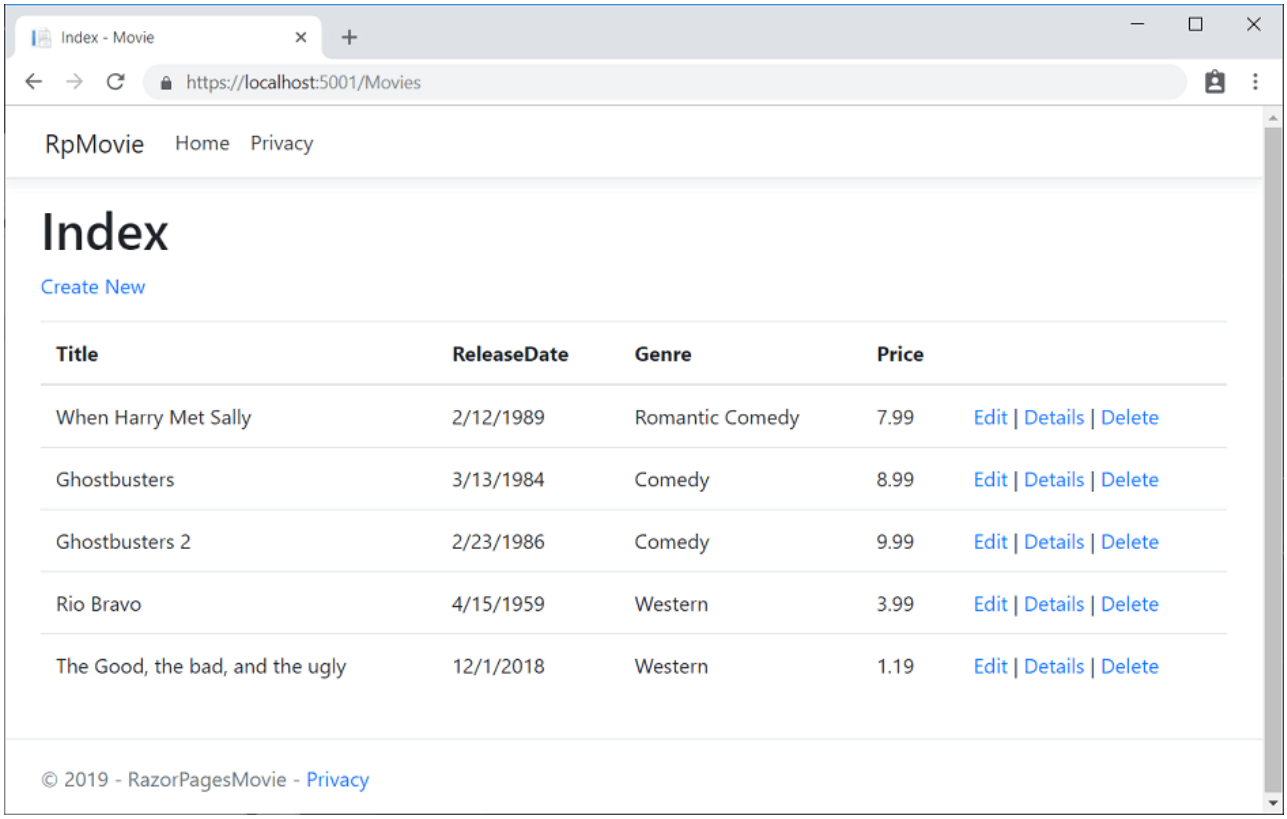
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:
  - Right-click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**:



  - If you were running VS in non-debug mode, press F5 to run in debug mode.
  - If you were running VS in debug mode, stop the debugger and press F5.

The app shows the seeded data:



The next tutorial will clean up the presentation of the data.

Additional resources

- YouTube version of this tutorial

PREVIOUS: SCAFFOLDED RAZOR PAGES

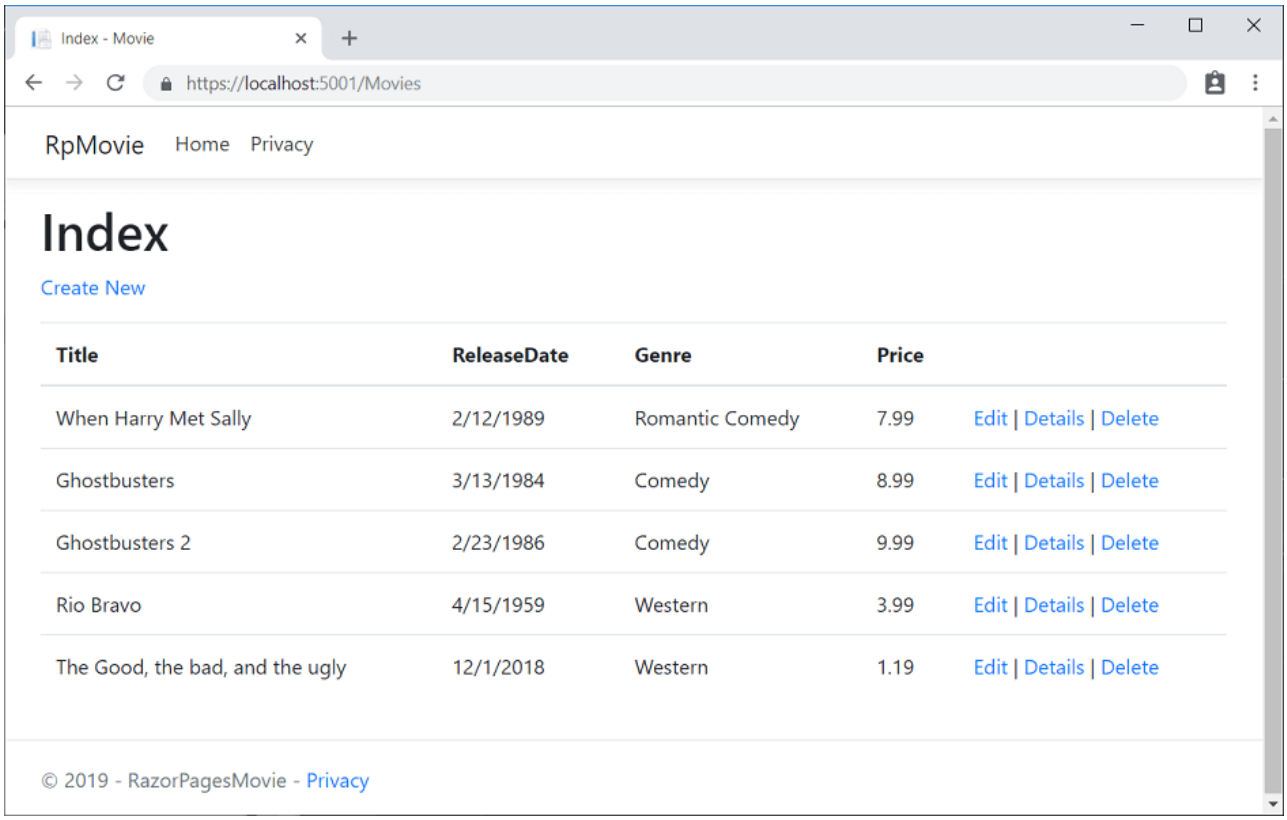
NEXT: UPDATING THE PAGES

Part 5, update the generated pages in an ASP.NET Core app

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

The scaffolded movie app has a good start, but the presentation isn't ideal. **ReleaseDate** should be **Release Date** (two words).



Update the generated code

Open the *Models/Movie.cs* file and add the highlighted lines shown in the following code:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

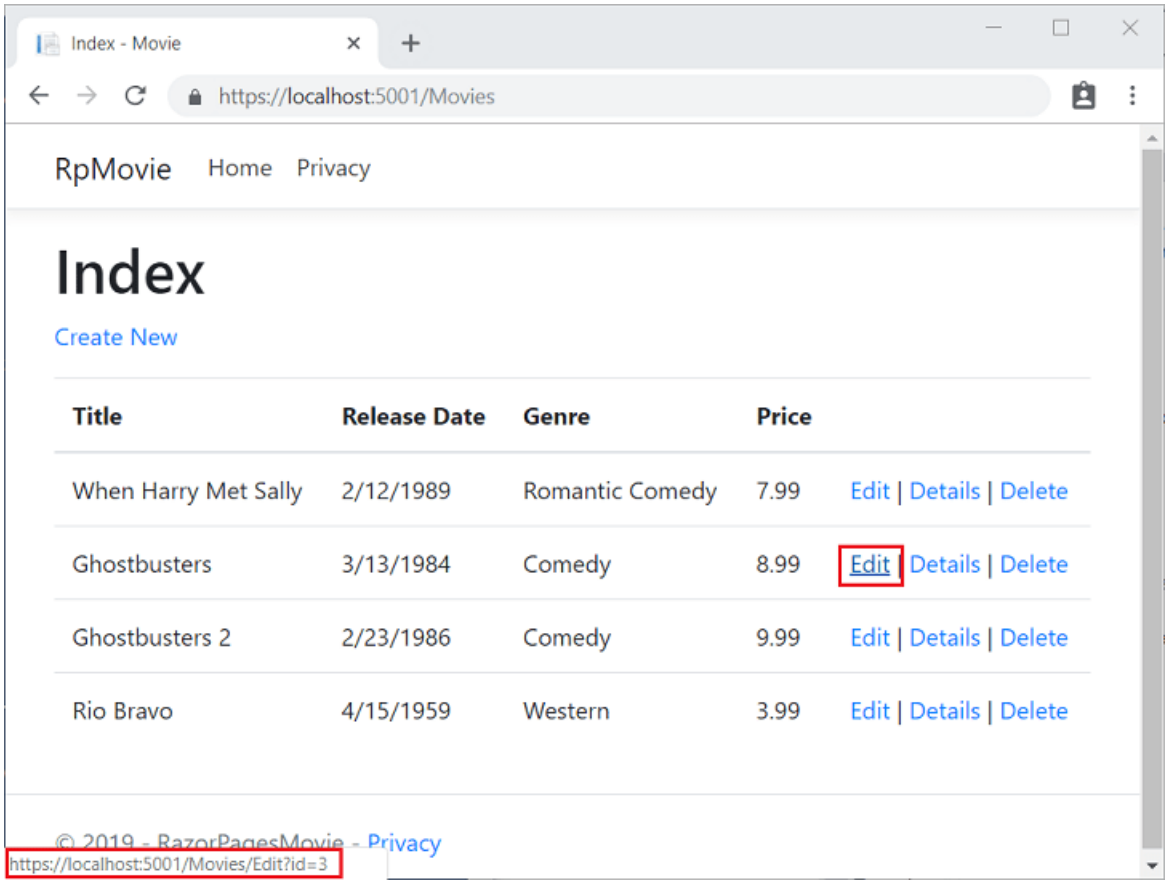
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

The `[Column(TypeName = "decimal(18, 2)")]` data annotation enables Entity Framework Core to correctly map `Price` to currency in the database. For more information, see Data Types.

DataAnnotations is covered in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

Browse to Pages/Movies and hover over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the *Pages/Movies/Index.cshtml* file.

```
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the preceding code, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route id (`asp-route-id`). See URL generation for Pages for more information.

Use **View Source** from your favorite browser to examine the generated markup. A portion of the generated HTML is shown below:

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically-generated links pass the movie ID with a query string (for example, the `?id=1` in `https://localhost:5001/Movies/Details?id=1`).

Add route template

Update the Edit, Details, and Delete Razor Pages to use the "{id:int}" route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`. Run the app and then view source. The generated HTML adds the ID to the path portion of the URL:

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the "{id:int}" route template that does **not** include the integer will return an HTTP 404 (not found) error. For example, `http://localhost:5000/Movies/Details` will return a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

To test the behavior of `@page "{id:int?}"`:

- Set the page directive in *Pages/Movies/Details.cshtml* to `@page "{id:int?}"`.
- Set a break point in `public async Task<IActionResult> OnGetAsync(int? id)` (in *Pages/Movies/Details.cshtml.cs*).
- Navigate to `https://localhost:5001/Movies/Details/`.

With the `@page "{id:int}"` directive, the break point is never hit. The routing engine returns HTTP 404. Using `@page "{id:int?}"`, the `OnGetAsync` method returns `NotFound` (HTTP 404).

Review concurrency exception handling

Review the `OnPostAsync` method in the *Pages/Movies/Edit.cshtml.cs* file:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.ID == id);
}
```

The previous code detects concurrency exceptions when the one client deletes the movie and the other client posts changes to the movie.

To test the `catch` block:

- Set a breakpoint on `catch (DbUpdateConcurrencyException)`
- Select **Edit** for a movie, make changes, but don't enter **Save**.
- In another browser window, select the **Delete** link for the same movie, and then delete the movie.
- In the previous browser window, post changes to the movie.

Production code may want to detect concurrency conflicts. See Handle concurrency conflicts for more information.

Posting and binding review

Examine the *Pages/Movies/Edit.cshtml.cs* file:

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Movie = await _context.Movie.FirstOrDefaultAsync(m => m.ID == id);

        if (Movie == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(Movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return RedirectToPage("./Index");
    }

    private bool MovieExists(int id)
    {
        return _context.Movie.Any(e => e.ID == id);
    }
```

When an HTTP GET request is made to the Movies/Edit page (for example, `http://localhost:5000/Movies/Edit/2`):

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the *Pages/Movies/Edit.cshtml* Razor Page. The *Pages/Movies/Edit.cshtml* file contains

the model directive (`@model RazorPagesMovie.Pages.Movies.EditModel`), which makes the movie model available on the page.

- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables Model binding.

```
[BindProperty]
public Movie Movie { get; set; }
```

- If there are errors in the model state (for example, `ReleaseDate` cannot be converted to a date), the form is redisplayed with the submitted values.
- If there are no model errors, the movie is saved.

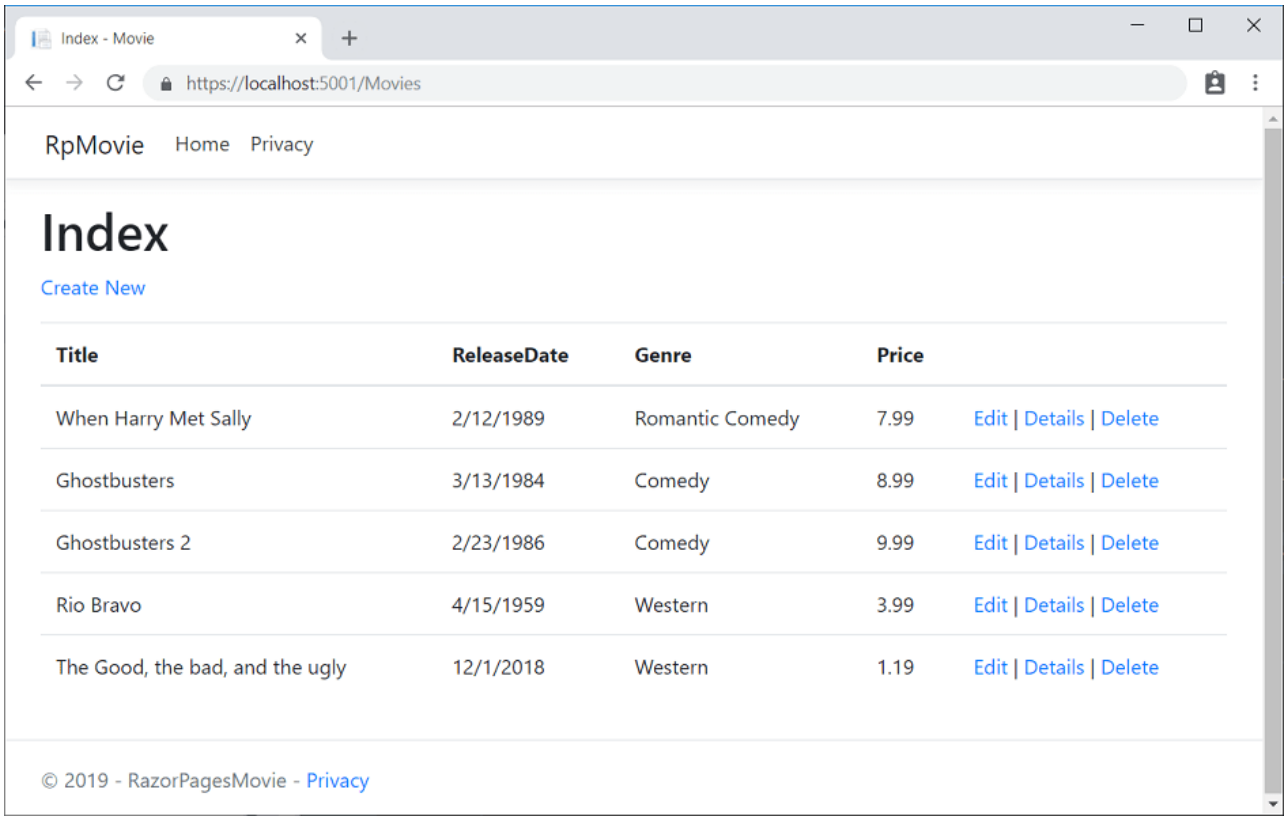
The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Additional resources

PREVIOUS: WORKING WITH A DATABASE

NEXT: ADD SEARCH

The scaffolded movie app has a good start, but the presentation isn't ideal. **ReleaseDate** should be **Release Date** (two words).



Update the generated code

Open the *Models/Movie.cs* file and add the highlighted lines shown in the following code:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

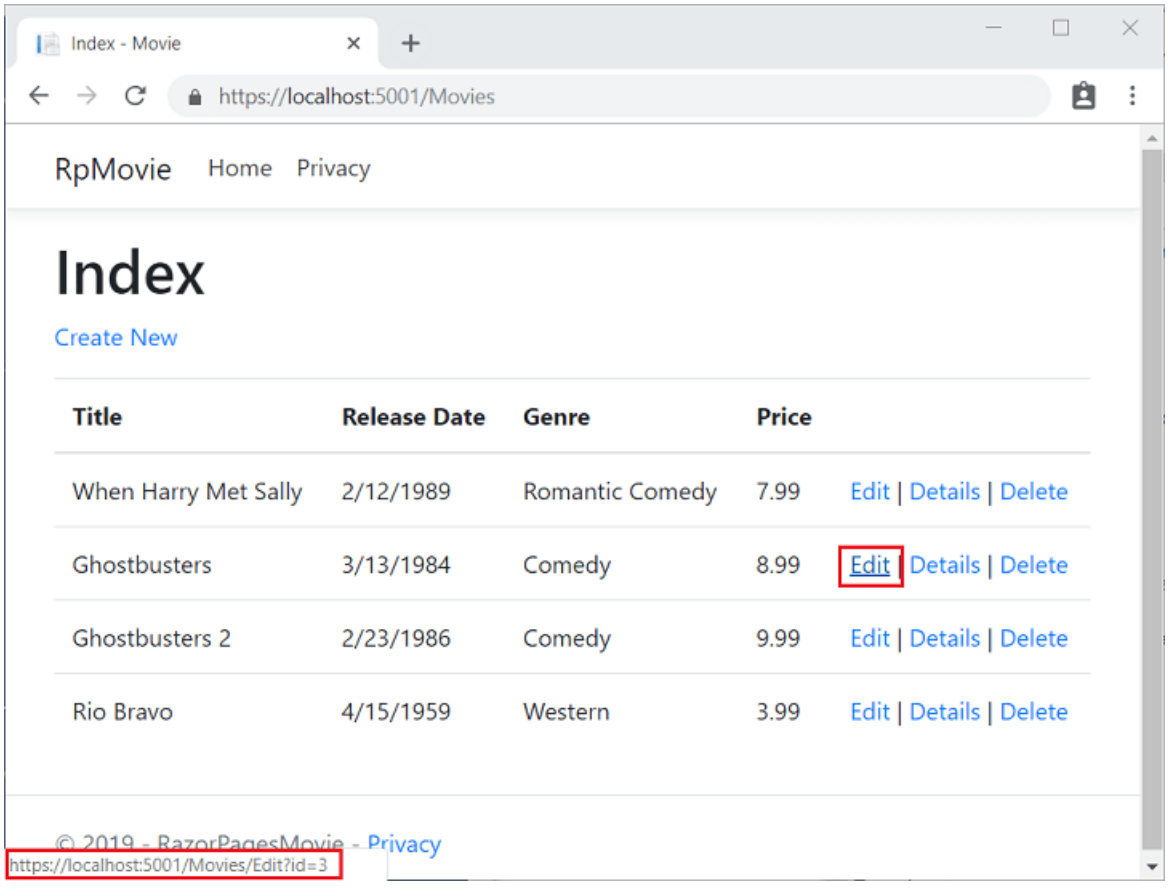
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

The `[Column(TypeName = "decimal(18, 2)")]` data annotation enables Entity Framework Core to correctly map `Price` to currency in the database. For more information, see Data Types.

DataAnnotations is covered in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

Browse to Pages/Movies and hover over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the *Pages/Movies/Index.cshtml* file.

```
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the preceding code, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route id (`asp-route-id`). See URL generation for Pages for more information.

Use **View Source** from your favorite browser to examine the generated markup. A portion of the generated HTML is shown below:

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically-generated links pass the movie ID with a query string (for example, the `?id=1` in `https://localhost:5001/Movies/Details?id=1`).

Update the Edit, Details, and Delete Razor Pages to use the "{id:int}" route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`. Run the app and then view source. The generated HTML adds the ID to the path portion of the URL:

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the "{id:int}" route template that does **not** include the integer will return an HTTP 404 (not found) error. For example, `http://localhost:5000/Movies/Details` will return a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

To test the behavior of `@page "{id:int?}"`:

- Set the page directive in *Pages/Movies/Details.cshtml* to `@page "{id:int?}"`.
- Set a break point in `public async Task<IActionResult> OnGetAsync(int? id)` (in *Pages/Movies/Details.cshtml.cs*).
- Navigate to `https://localhost:5001/Movies/Details/`.

With the `@page "{id:int}"` directive, the break point is never hit. The routing engine returns HTTP 404. Using `@page "{id:int?}"`, the `OnGetAsync` method returns `NotFound` (HTTP 404).

Review concurrency exception handling

Review the `OnPostAsync` method in the *Pages/Movies/Edit.cshtml.cs* file:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.ID == id);
}
```

The previous code detects concurrency exceptions when the one client deletes the movie and the other client posts changes to the movie.

To test the `catch` block:

- Set a breakpoint on `catch (DbUpdateConcurrencyException)`
- Select **Edit** for a movie, make changes, but don't enter **Save**.
- In another browser window, select the **Delete** link for the same movie, and then delete the movie.
- In the previous browser window, post changes to the movie.

Production code may want to detect concurrency conflicts. See Handle concurrency conflicts for more information.

Posting and binding review

Examine the *Pages/Movies/Edit.cshtml.cs* file:

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);

        if (Movie == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_context.Movie.Any(e => e.ID == Movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return RedirectToPage("./Index");
    }
}
```

When an HTTP GET request is made to the Movies/Edit page (for example, `http://localhost:5000/Movies/Edit/2`):

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the *Pages/Movies/Edit.cshtml* Razor Page. The *Pages/Movies/Edit.cshtml* file contains the model directive (`@model RazorPagesMovie.Pages.Movies.EditModel`), which makes the movie model available on the page.
- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables Model binding.

```
[BindProperty]
public Movie Movie { get; set; }
```

- If there are errors in the model state (for example, `ReleaseDate` cannot be converted to a date), the form is displayed with the submitted values.
- If there are no model errors, the movie is saved.

The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Search is added in the next tutorial.

Additional resources

- YouTube version of this tutorial

PREVIOUS: WORKING WITH A DATABASE

NEXT: ADD SEARCH

Part 6, add search to ASP.NET Core Razor Pages

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

View or download sample code (how to download).

View or download sample code (how to download).

In the following sections, searching movies by *genre* or *name* is added.

Add the following highlighted properties to *Pages/Movies/Index.cshtml.cs*:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    [BindProperty(SupportsGet = true)]
    public string SearchString { get; set; }
    // Requires using Microsoft.AspNetCore.Mvc.Rendering;
    public SelectList Genres { get; set; }
    [BindProperty(SupportsGet = true)]
    public string MovieGenre { get; set; }
```

- `SearchString`: contains the text users enter in the search text box. `SearchString` has the `[BindProperty]` attribute. `[BindProperty]` binds form values and query strings with the same name as the property. `(SupportsGet = true)` is required for binding on GET requests.
- `Genres`: contains the list of genres. `Genres` allows the user to select a genre from the list. `SelectList` requires `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`: contains the specific genre the user selects (for example, "Western").
- `Genres` and `MovieGenre` are used later in this tutorial.

WARNING

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the `[BindProperty]` attribute's `SupportsGet` property to `true`:

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Update the Index page's `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a LINQ query to select the movies:

```
// using System.Linq;
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

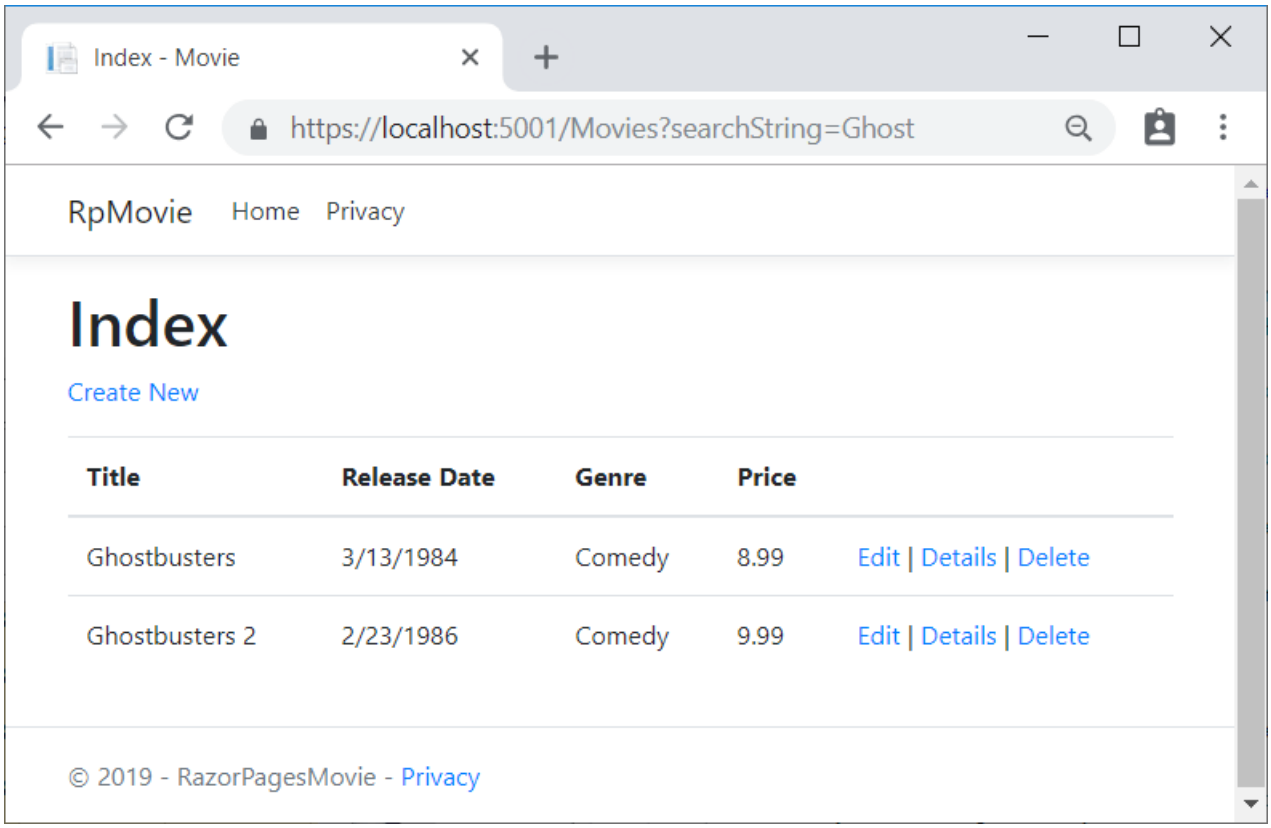
If the `SearchString` property is not null or empty, the movies query is modified to filter on the search string:

```
if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}
```

The `s => s.Title.Contains()` code is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the preceding code). LINQ queries are not executed when they're defined or when they're modified by calling a method (such as `Where`, `Contains` or `OrderBy`). Rather, query execution is deferred. That means the evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See Query Execution for more information.

> **NOTE**
>
> The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

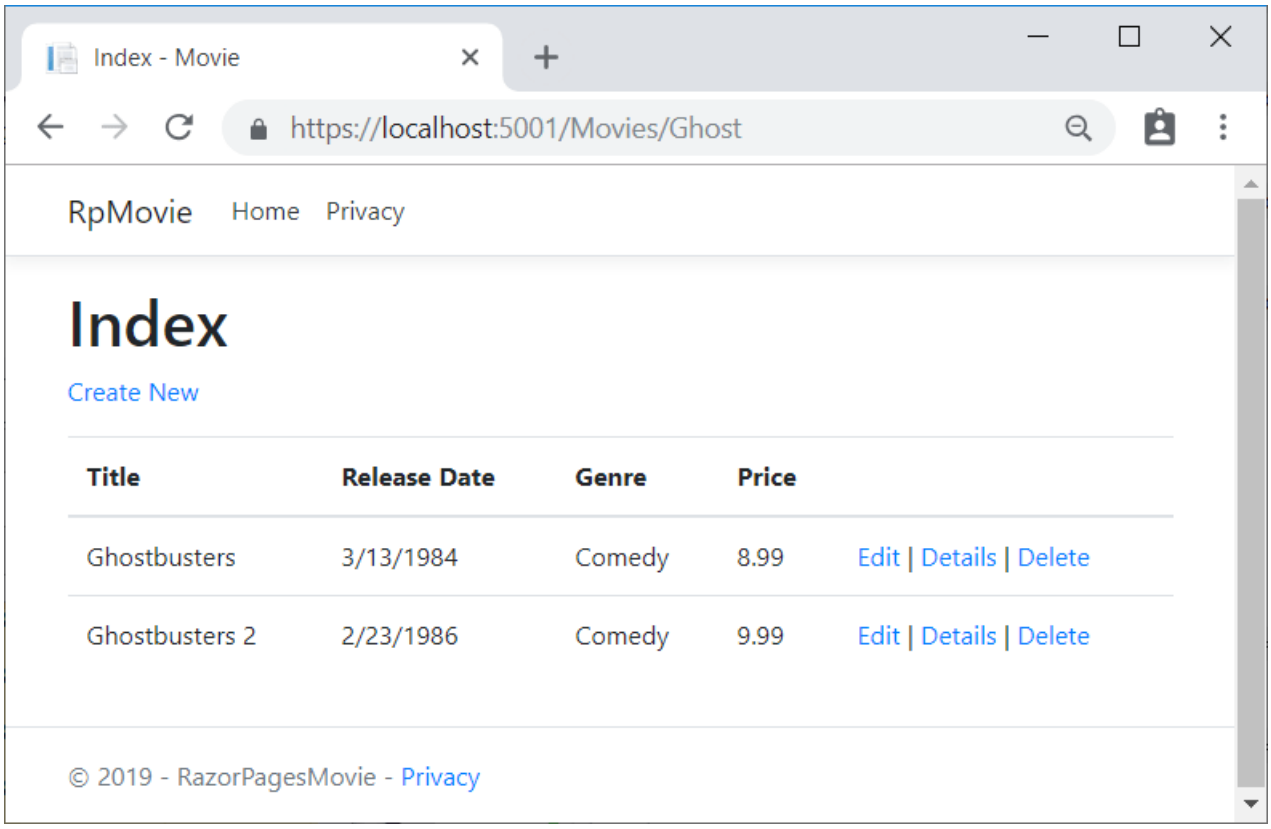
Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL (for example, `https://localhost:5001/Movies?searchString=Ghost`). The filtered movies are displayed.



If the following route template is added to the Index page, the search string can be passed as a URL segment (for example, `https://localhost:5001/Movies/Ghost`).

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



The ASP.NET Core runtime uses model binding to set the value of the `SearchString` property from the query string

(`?searchString=Ghost`) or route data (`https://localhost:5001/Movies/Ghost`). Model binding is not case sensitive.

However, you can't expect users to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the *Pages/Movies/Index.cshtml* file, and add the `<form>` markup highlighted in the following code:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

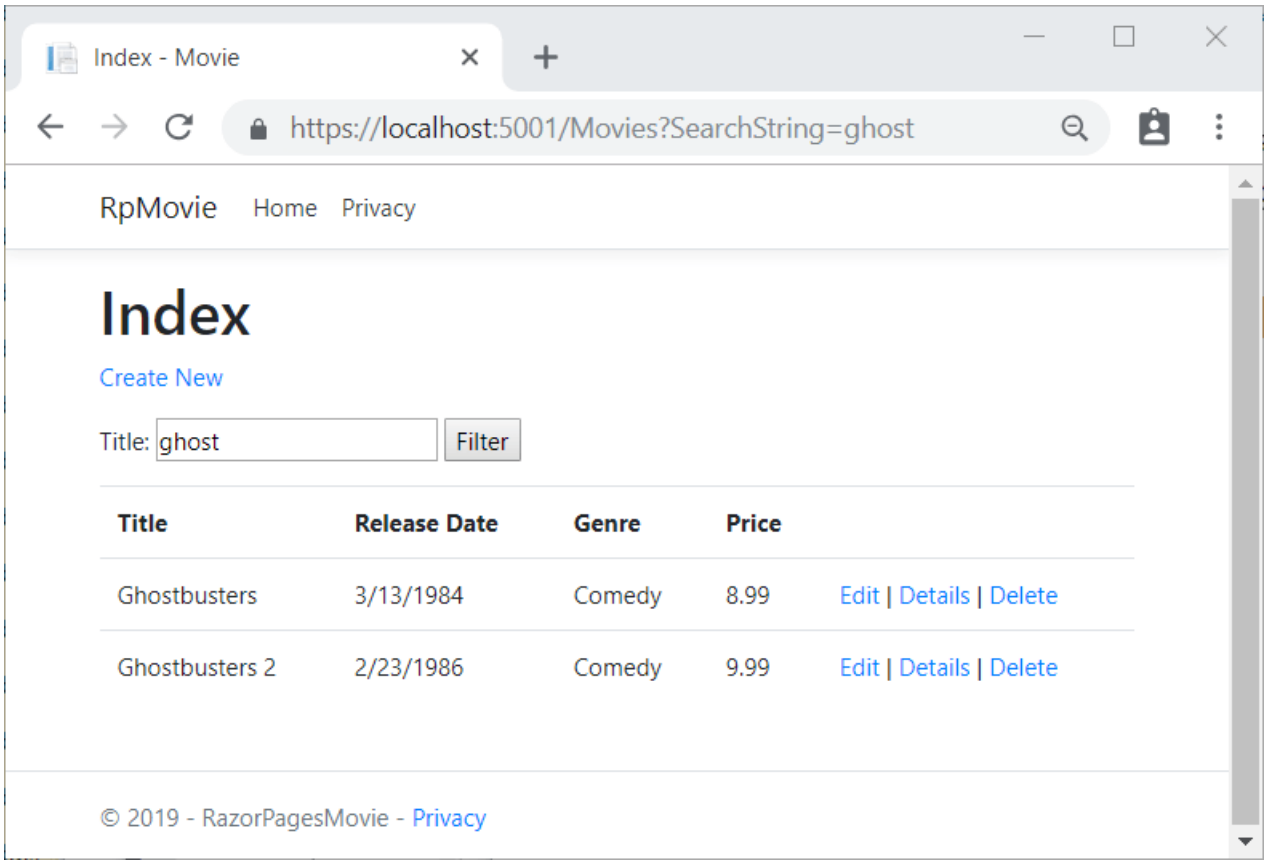
<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

The HTML `<form>` tag uses the following Tag Helpers:

- Form Tag Helper. When the form is submitted, the filter string is sent to the *Pages/Movies/Index* page via query string.
- Input Tag Helper

Save the changes and test the filter.



Search by genre

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

The following code is a LINQ query that retrieves all the genres from the database.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres.

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Add search by genre to the Razor Page

Update *Index.cshtml* as follows:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

Test the app by searching by genre, by movie title, and by both.

Additional resources

- YouTube version of this tutorial

PREVIOUS: UPDATING THE PAGES

NEXT: ADDING A NEW FIELD

View or download sample code (how to download).

View or download sample code (how to download).

In the following sections, searching movies by *genre* or *name* is added.

Add the following highlighted properties to *Pages/Movies/Index.cshtml.cs*:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    [BindProperty(SupportsGet = true)]
    public string SearchString { get; set; }
    // Requires using Microsoft.AspNetCore.Mvc.Rendering;
    public SelectList Genres { get; set; }
    [BindProperty(SupportsGet = true)]
    public string MovieGenre { get; set; }
```

- `SearchString`: contains the text users enter in the search text box. `SearchString` has the `[BindProperty]` attribute. `[BindProperty]` binds form values and query strings with the same name as the property. `(SupportsGet = true)` is required for binding on GET requests.
- `Genres`: contains the list of genres. `Genres` allows the user to select a genre from the list. `SelectList` requires `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`: contains the specific genre the user selects (for example, "Western").
- `Genres` and `MovieGenre` are used later in this tutorial.

WARNING

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the `[BindProperty]` attribute's `SupportsGet` property to `true`:

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Update the Index page's `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a LINQ query to select the movies:

```
// using System.Linq;
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

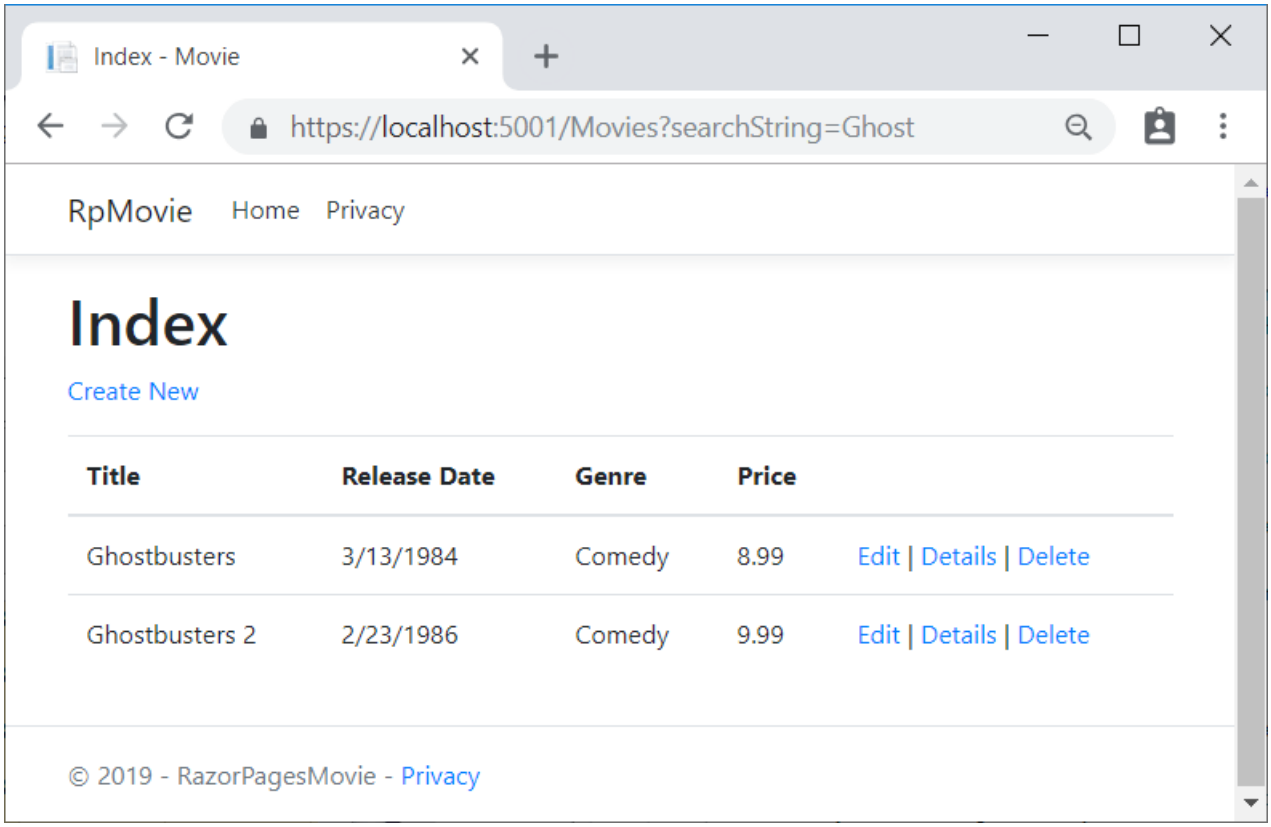
If the `SearchString` property is not null or empty, the movies query is modified to filter on the search string:

```
if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}
```

The `s => s.Title.Contains()` code is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the preceding code). LINQ queries are not executed when they're defined or when they're modified by calling a method (such as `Where`, `Contains` or `OrderBy`). Rather, query execution is deferred. That means the evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See Query Execution for more information.

Note: The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

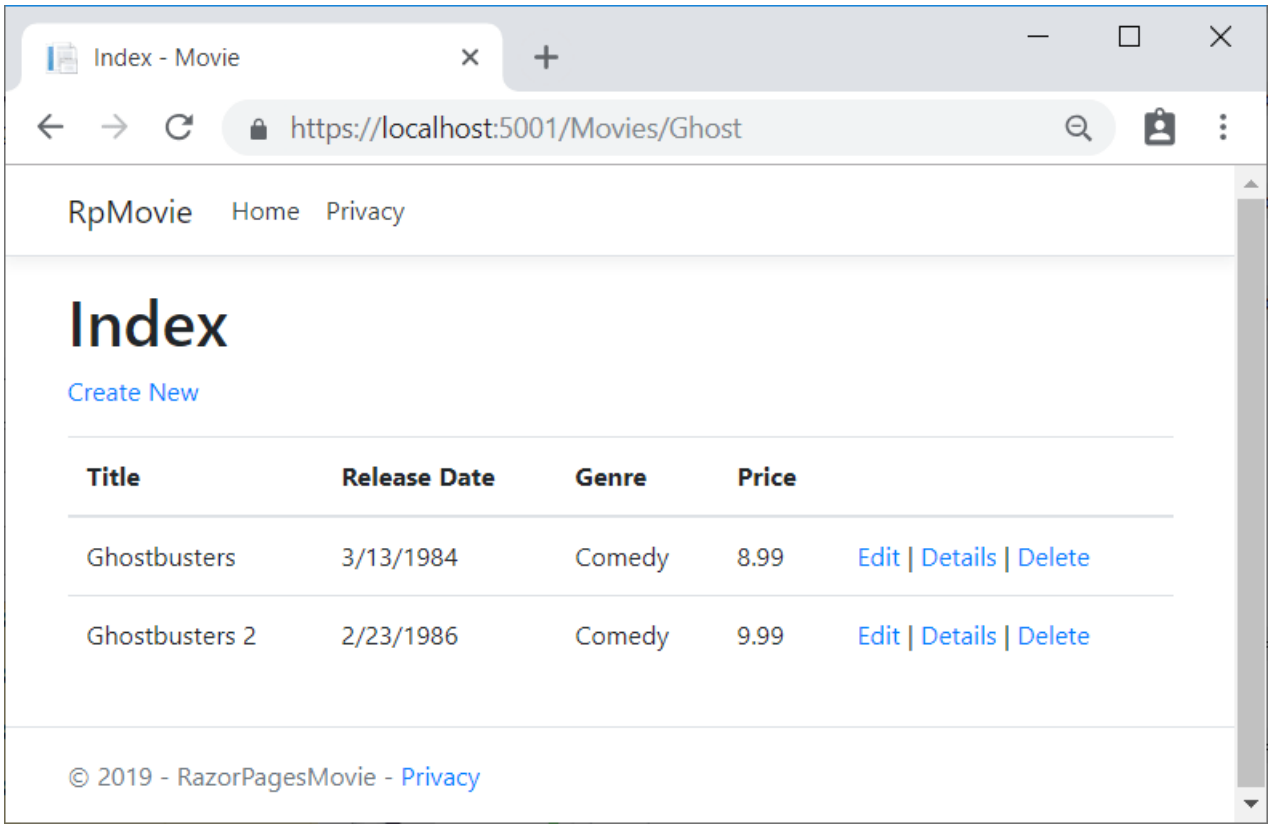
Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL (for example, `https://localhost:5001/Movies?searchString=Ghost`). The filtered movies are displayed.



If the following route template is added to the Index page, the search string can be passed as a URL segment (for example, `https://localhost:5001/Movies/Ghost`).

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



The ASP.NET Core runtime uses model binding to set the value of the `SearchString` property from the query string (`?searchString=Ghost`) or route data (`https://localhost:5001/Movies/Ghost`). Model binding is not case sensitive.

However, you can't expect users to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the *Pages/Movies/Index.cshtml* file, and add the `<form>` markup highlighted in the following code:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

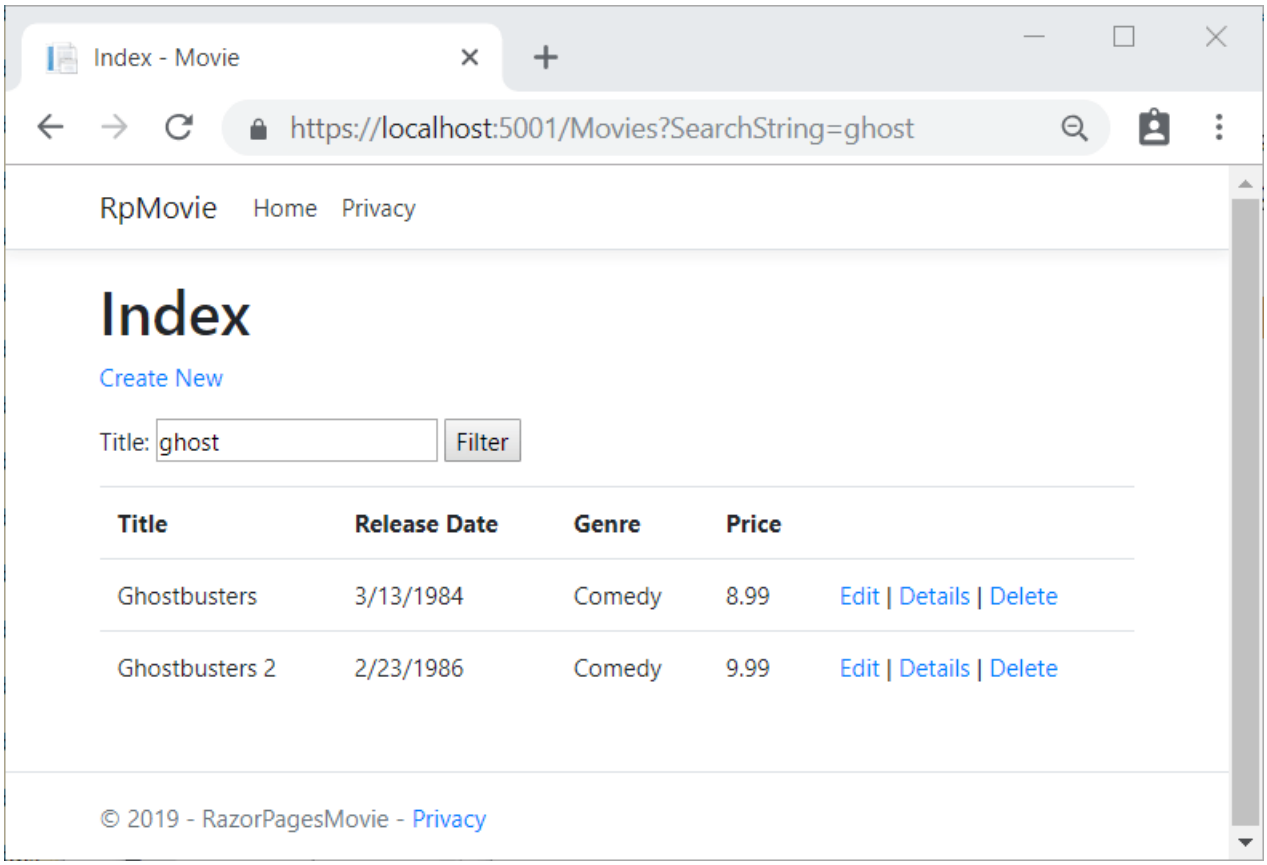
<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

The HTML `<form>` tag uses the following Tag Helpers:

- Form Tag Helper. When the form is submitted, the filter string is sent to the *Pages/Movies/Index* page via query string.
- Input Tag Helper

Save the changes and test the filter.



Search by genre

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

The following code is a LINQ query that retrieves all the genres from the database.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres.

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Add search by genre to the Razor Page

Update *Index.cshtml* as follows:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

Test the app by searching by genre, by movie title, and by both. The preceding code uses the Select Tag Helper and Option Tag Helper.

Additional resources

- YouTube version of this tutorial

PREVIOUS: UPDATING THE PAGES

NEXT: ADDING A NEW FIELD

Part 7, add a new field to a Razor Page in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

View or download sample code (how to download).

View or download sample code (how to download).

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field schema change to the database.

When using EF Code First to automatically create a database, Code First:

- Adds an `__EFMigrationsHistory` table to the database to track whether the schema of the database is in sync with the model classes it was generated from.
- If the model classes aren't in sync with the DB, EF throws an exception.

Automatic verification of schema/model in sync makes it easier to find inconsistent database/code issues.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app.

Edit *Pages/Movies/Index.cshtml*, and add a `Rating` field:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
```

```
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the following pages:

- Add the `Rating` field to the Delete and Details pages.

- Update Create.cshtml with a `Rating` field.
- Add the `Rating` field to the Edit Page.

The app won't work until the DB is updated to include the new field. Running the app without updating the database throws a `SqlException`:

```
SqlException: Invalid column name 'Rating'.
```

The `SqlException` exception is caused by the updated Movie model class being different than the schema of the Movie table of the database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle; it allows you to quickly evolve the model and database schema together. The downside is that you lose existing data in the database. Don't use this approach on a production database! Dropping the DB on schema changes and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie` block.

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Price = 7.99M,
        Rating = "R"
    },
```

See the completed SeedData.cs file.

Build the solution.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add a migration for the rating field

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**. In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` DB schema.
- Create code to migrate the DB schema to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

The `Update-Database` command tells the framework to apply the schema changes to the database and to preserve existing data.

If you delete all the records in the DB, the initializer will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from Sql Server Object Explorer (SSOX).

Another option is to delete the database and use migrations to re-create the database. To delete the database in SSOX:

- Select the database in SSOX.
- Right click on the database, and select *Delete*.
- Check **Close existing connections**.
- Select **OK**.
- In the PMC, update the database:

```
Update-Database
```

Run the app and verify you can create/edit/display movies with a `Rating` field. If the database isn't seeded, set a break point in the `SeedData.Initialize` method.

Additional resources

- YouTube version of this tutorial

PREVIOUS: ADDING SEARCH

NEXT: ADDING VALIDATION

View or download sample code (how to download).

View or download sample code (how to download).

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field schema change to the database.

When using EF Code First to automatically create a database, Code First:

- Adds a table to the database to track whether the schema of the database is in sync with the model classes it was generated from.
- If the model classes aren't in sync with the DB, EF throws an exception.

Automatic verification of schema/model in sync makes it easier to find inconsistent database/code issues.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app.

Edit *Pages/Movies/Index.cshtml*, and add a `Rating` field:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
```

```
        {
        <tr><td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Rating)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
        }
    </tbody>
</table>
```

Update the following pages:

- Add the `Rating` field to the Delete and Details pages.
- Update Create.cshtml with a `Rating` field.
- Add the `Rating` field to the Edit Page.

The app won't work until the DB is updated to include the new field. If run now, the app throws a `SqlException`:

```
SqlException: Invalid column name 'Rating'.
```

This error is caused by the updated Movie model class being different than the schema of the Movie table of the database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle; it allows you to quickly evolve the model and database schema together. The downside is that you lose existing data in the database. Don't use this approach on a production database! Dropping the DB on schema changes and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie` block.

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Price = 7.99M,
        Rating = "R"
    },
```

See the completed SeedData.cs file.

Build the solution.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add a migration for the rating field

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` DB schema.
- Create code to migrate the DB schema to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

The `Update-Database` command tells the framework to apply the schema changes to the database.

If you delete all the records in the DB, the initializer will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from Sql Server Object Explorer (SSOX).

Another option is to delete the database and use migrations to re-create the database. To delete the database in SSOX:

- Select the database in SSOX.
- Right click on the database, and select *Delete*.
- Check **Close existing connections**.
- Select **OK**.
- In the PMC, update the database:

  ```
  Update-Database
  ```

Run the app and verify you can create/edit/display movies with a `Rating` field. If the database isn't seeded, set a break point in the `SeedData.Initialize` method.

Additional resources

- YouTube version of this tutorial

PREVIOUS: ADDING SEARCH

NEXT: ADDING VALIDATION

Part 8, add validation to an ASP.NET Core Razor Page

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

In this section, validation logic is added to the `Movie` model. The validation rules are enforced any time a user creates or edits a movie.

Validation

A key tenet of software development is called DRY ("**D**on't **R**epeat **Y**ourself"). Razor Pages encourages development where functionality is specified once, and it's reflected throughout the app. DRY can help:

- Reduce the amount of code in an app.
- Make the code less error prone, and easier to test and maintain.

The validation support provided by Razor Pages and Entity Framework is a good example of the DRY principle. Validation rules are declaratively specified in one place (in the model class), and the rules are enforced everywhere in the app.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

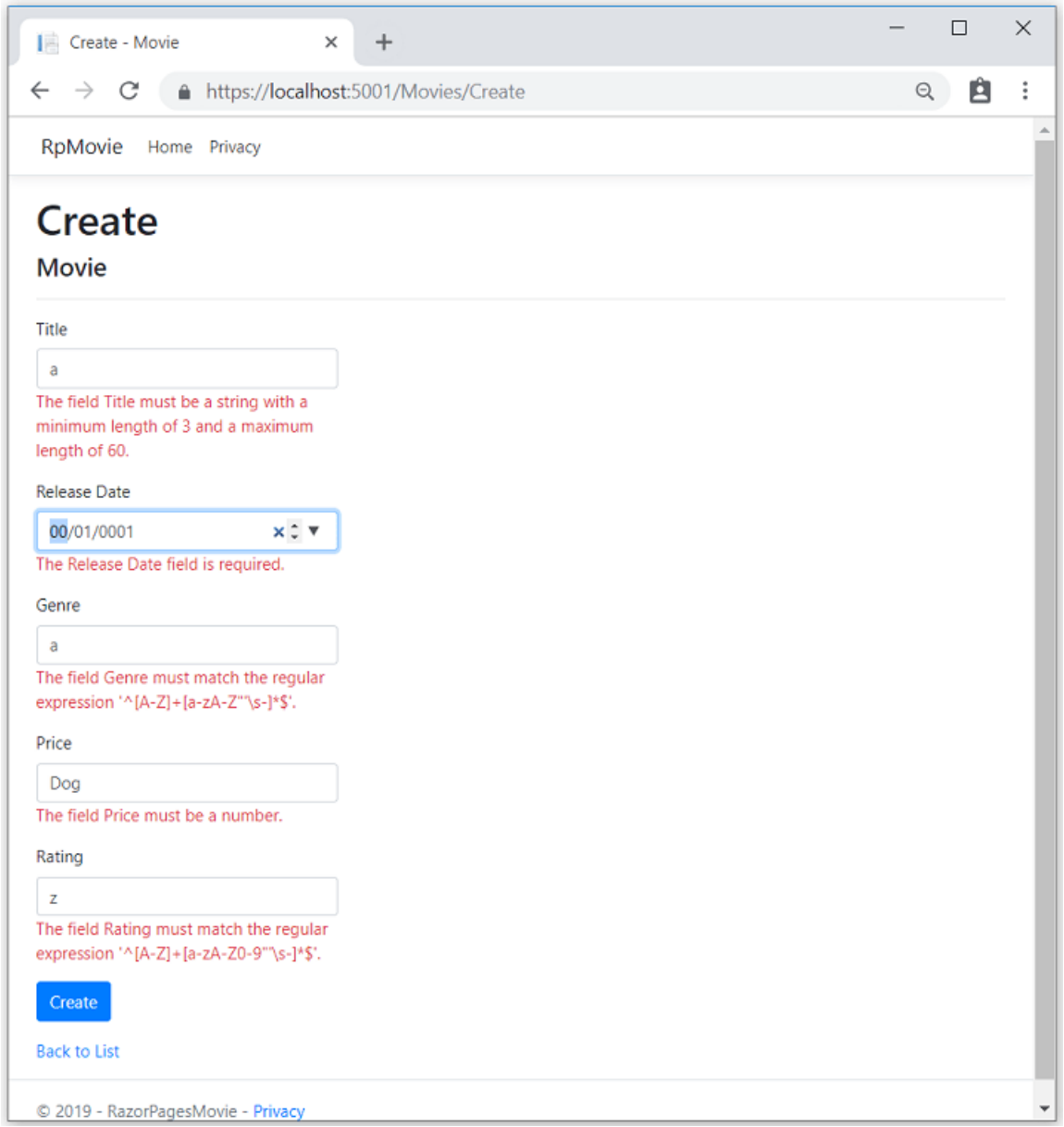
- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White space, numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI in Razor Pages

Run the app and navigate to Pages/Movies.

Select the **Create New** link. Complete the form with some invalid values. When jQuery client-side validation detects the error, it displays an error message.



> **NOTE**
> You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered a validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (when a user has JavaScript disabled).

A significant benefit is that **no** code changes were necessary in the Create or Edit pages. Once DataAnnotations were applied to the model, the validation UI was enabled. The Razor Pages created in this tutorial automatically picked up the validation rules (using validation attributes on the properties of the `Movie` model class). Test validation using the Edit page, the same validation is applied.

The form data isn't posted to the server until there are no client-side validation errors. Verify form data isn't posted by one or more of the following approaches:

- Put a break point in the `OnPostAsync` method. Submit the form (select **Create** or **Save**). The break point is never hit.
- Use the Fiddler tool.

- Use the browser developer tools to monitor network traffic.

Server-side validation

When JavaScript is disabled in the browser, submitting the form with errors will post to the server.

Optional, test server-side validation:

- Disable JavaScript in the browser. You can disable JavaScript using browser's developer tools. If you can't disable JavaScript in the browser, try another browser.
- Set a break point in the `OnPostAsync` method of the Create or Edit page.
- Submit a form with invalid data.
- Verify the model state is invalid:

```
if (!ModelState.IsValid)
{
    return Page();
}
```

Alternatively, you can Disable client-side validation on the server.

The following code shows a portion of the *Create.cshtml* page scaffolded earlier in the tutorial. It's used by the Create and Edit pages to display the initial form and to redisplay the form in the event of an error.

```
<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Movie.Title" class="control-label"></label>
        <input asp-for="Movie.Title" class="form-control" />
        <span asp-validation-for="Movie.Title" class="text-danger"></span>
    </div>
```

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side. The Validation Tag Helper displays validation errors. See Validation for more information.

The Create and Edit pages have no validation rules in them. The validation rules and the error strings are specified only in the `Movie` class. These validation rules are automatically applied to Razor Pages that edit the `Movie` model.

When validation logic needs to change, it's done only in the model. Validation is applied consistently throughout the application (validation logic is defined in one place). Validation in one place helps keep the code clean, and makes it easier to maintain and update.

Using DataType Attributes

Examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. The `DataType` attribute is applied to the `ReleaseDate` and `Price` properties.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supplies attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email). Use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type. `DataType` attributes are not validation attributes. In the sample application, only the date is displayed, without time.

The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`. A date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers consume. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should be applied when the value is displayed for editing. You might not want that behavior for some fields. For example, in currency values, you probably don't want the currency symbol in the edit UI.

The `DisplayFormat` attribute can be used by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable the ASP.NET Core framework to choose the right field template to render the data. The `DisplayFormat`, if used by itself, uses the string template.

Note: jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client-side validation error, even when the date is in the specified range:

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

Get started with Razor Pages and EF Core shows advanced EF Core operations with Razor Pages.

Apply migrations

The DataAnnotations applied to the class changes the schema. For example, the DataAnnotations applied to the `Title` field:

```
[StringLength(60, MinimumLength = 3)]
[Required]
public string Title { get; set; }
```

- Limits the characters to 60.
- Doesn't allow a `null` value.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

The `Movie` table currently has the following schema:

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (MAX)  NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (MAX)  NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (MAX)  NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

The preceding schema changes don't cause EF to throw an exception. However, create a migration so the schema is consistent with the model.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**. In the PMC, enter the following commands:

```
Add-Migration New_DataAnnotations
Update-Database
```

`Update-Database` runs the `Up` methods of the `New_DataAnnotations` class. Examine the `Up` method:

```
public partial class New_DataAnnotations : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "Title",
            table: "Movie",
            maxLength: 60,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);

        migrationBuilder.AlterColumn<string>(
            name: "Rating",
            table: "Movie",
            maxLength: 5,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);

        migrationBuilder.AlterColumn<string>(
            name: "Genre",
            table: "Movie",
            maxLength: 30,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);
    }
```

The updated `Movie` table has the following schema:

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (60)   NOT NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (30)   NOT NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (5)    NOT NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core app in Azure with SQL Database.

Thanks for completing this introduction to Razor Pages. Get started with Razor Pages and EF Core is an excellent follow up to this tutorial.

Additional resources

- Tag Helpers in forms in ASP.NET Core
- Globalization and localization in ASP.NET Core
- Tag Helpers in ASP.NET Core
- Author Tag Helpers in ASP.NET Core
- YouTube version of this tutorial

PREVIOUS: ADDING A NEW FIELD

Create a web app with ASP.NET Core MVC

9/22/2020 • 2 minutes to read • Edit Online

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point.

The tutorial series includes the following:

1. Get started
2. Add a controller
3. Add a view
4. Add a model
5. Work with SQL Server LocalDB
6. Controller methods and views
7. Add search
8. Add a new field
9. Add validation
10. Examine the Details and Delete methods

Get started with ASP.NET Core MVC

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point.

This tutorial teaches the basics of building an ASP.NET Core MVC web app.

The app manages a database of movie titles. You learn how to:

- Create a web app.
- Add and scaffold a model.
- Work with a database.
- Add search and validation.

At the end, you have an app that can manage and display movie data.

View or download sample code (how to download).

Prerequisites

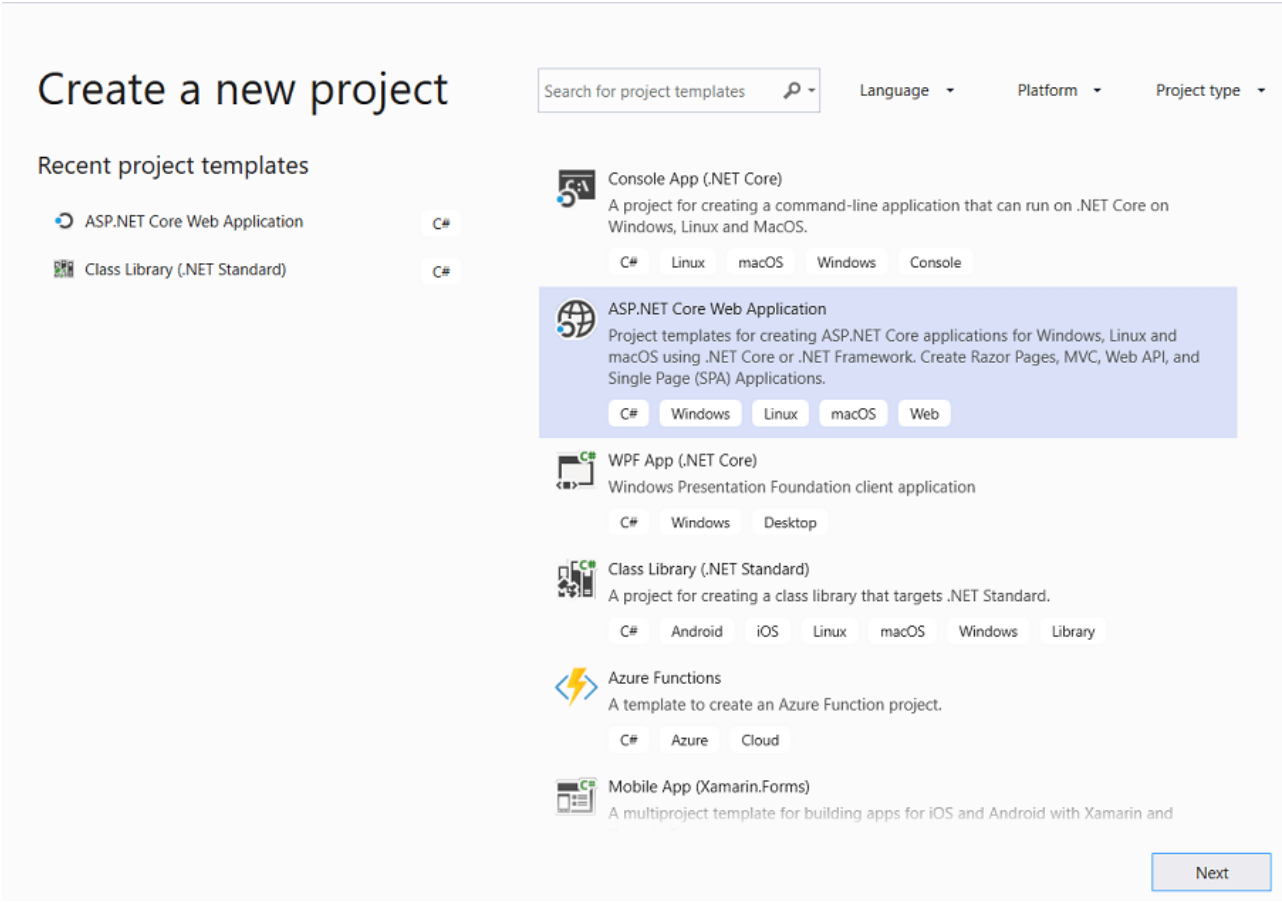
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

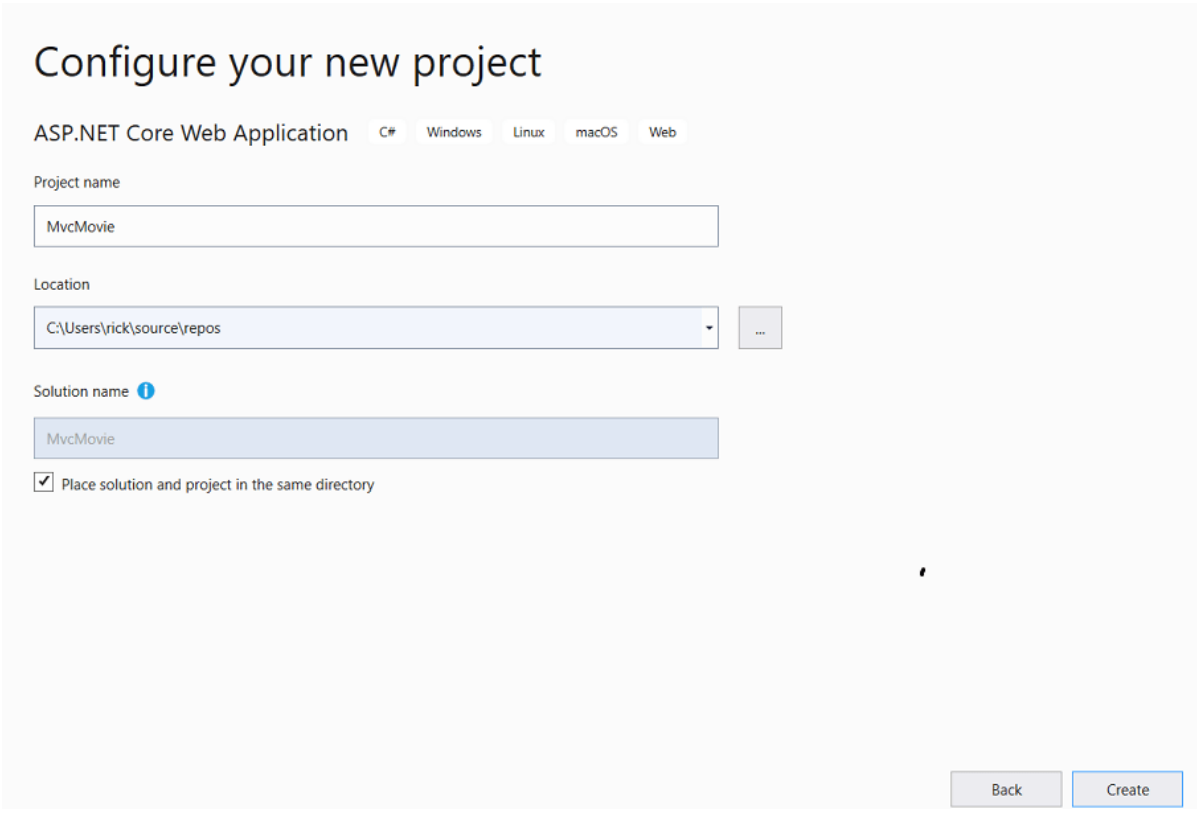
Create a web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

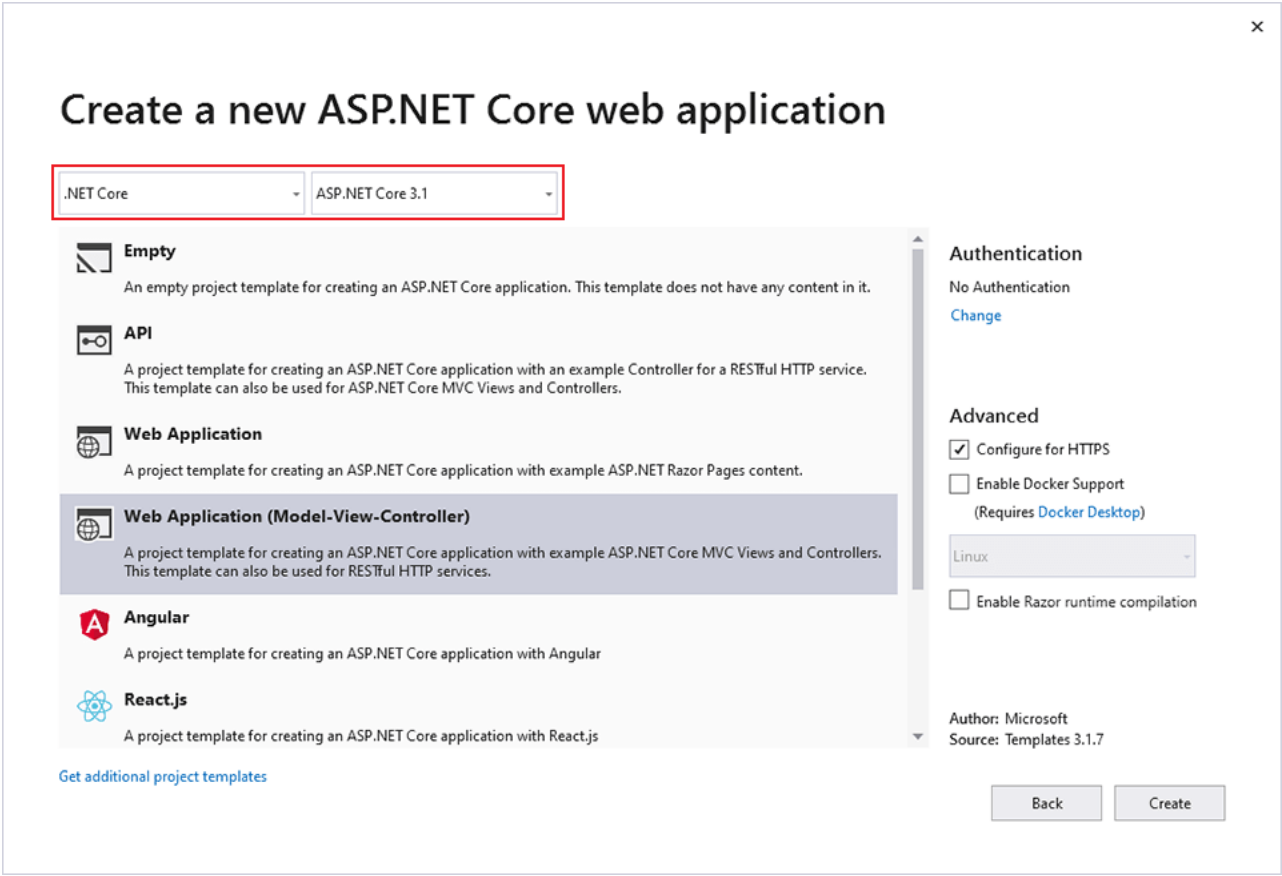
- From the Visual Studio select **Create a new project**.
- Select **ASP.NET Core Web Application** and then select **Next**.



- Name the project **MvcMovie** and select **Create**. It's important to name the project **MvcMovie** so when you copy code, the namespace will match.



- Select **Web Application(Model-View-Controller)**, and then select **Create**.



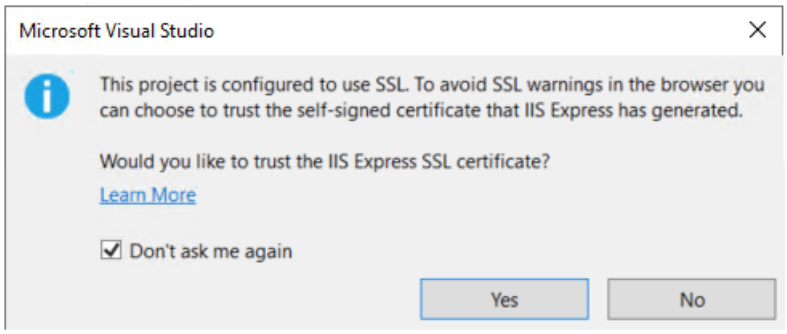
Visual Studio used the default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a basic starter project.

Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

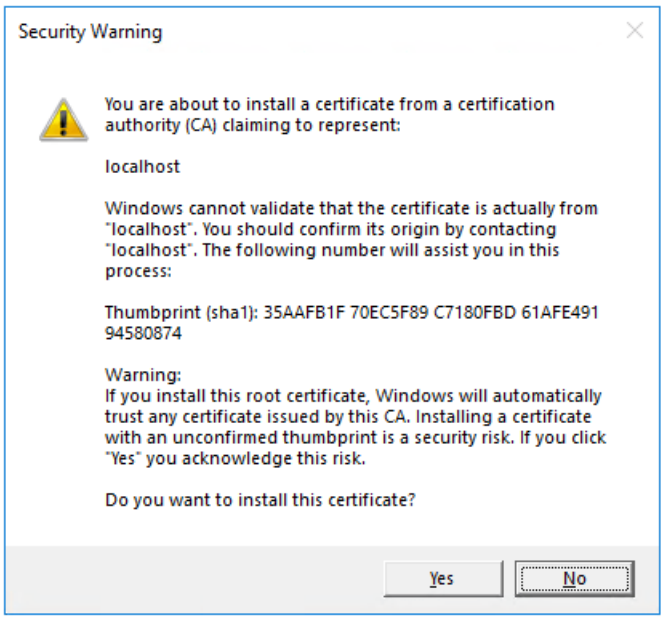
Select **Ctrl-F5** to run the app in non-debug mode.

Visual Studio displays the following dialog:



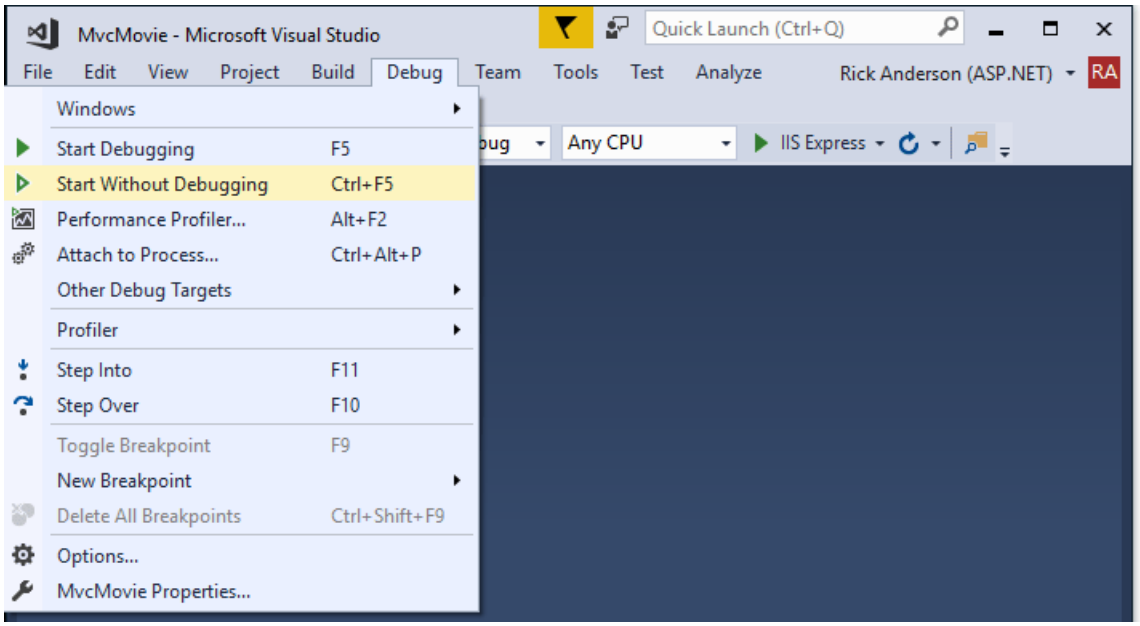
Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:

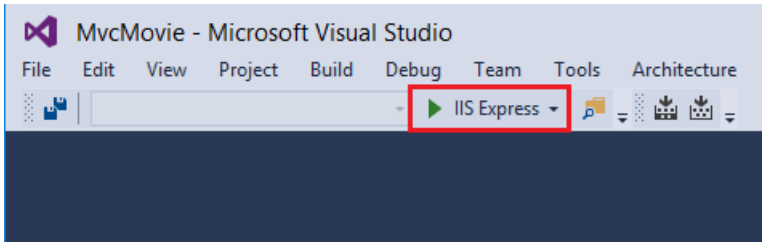


Select **Yes** if you agree to trust the development certificate.

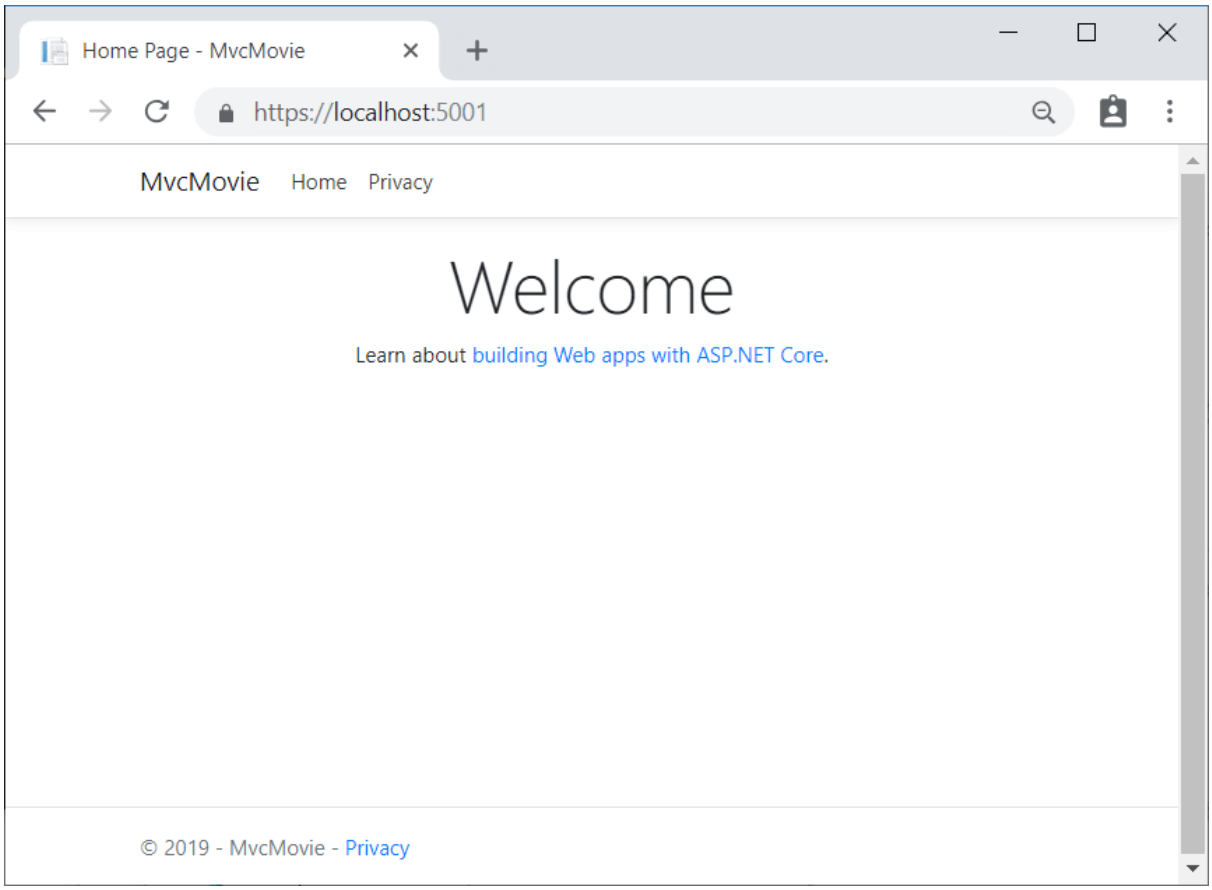
- Visual Studio starts IIS Express and runs the app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for your local computer. When Visual Studio creates a web project, a random port is used for the web server.
- Launching the app with Ctrl+F5 (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
- You can launch the app in debug or non-debug mode from the **Debug** menu item:



- You can debug the app by selecting the **IIS Express** button



The following image shows the app:



- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next part of this tutorial, you learn about MVC and start writing some code.

NEXT

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point.

This tutorial teaches the basics of building an ASP.NET Core MVC web app.

The app manages a database of movie titles. You learn how to:

- Create a web app.
- Add and scaffold a model.
- Work with a database.
- Add search and validation.

At the end, you have an app that can manage and display movie data.

View or download sample code (how to download).

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

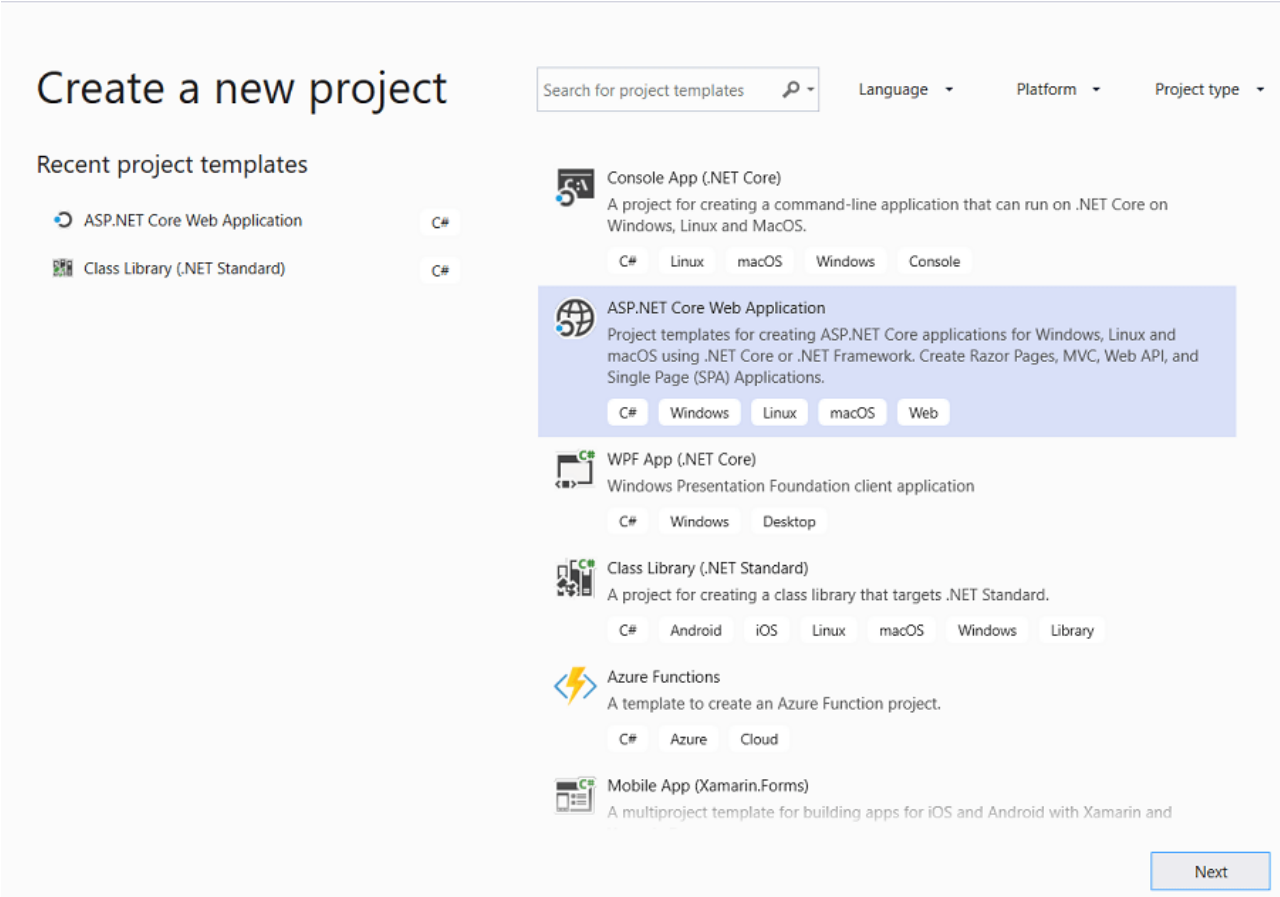
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

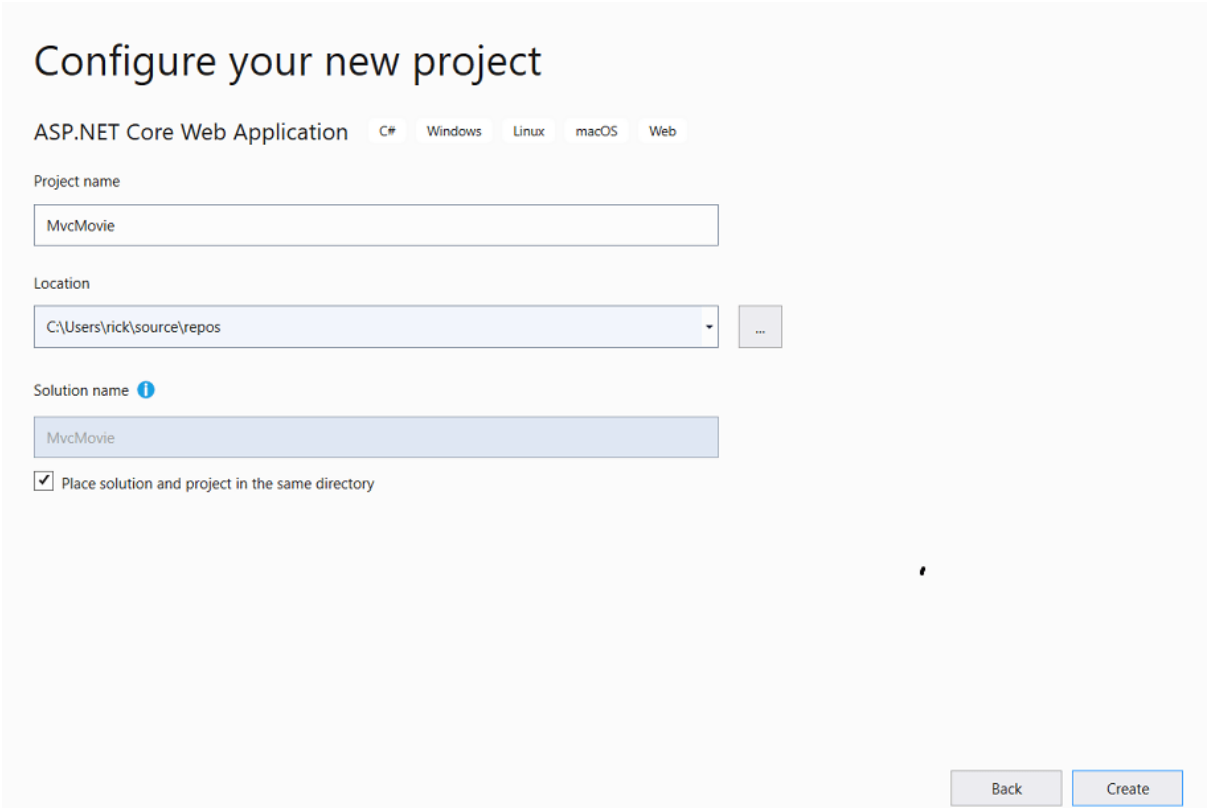
Create a web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

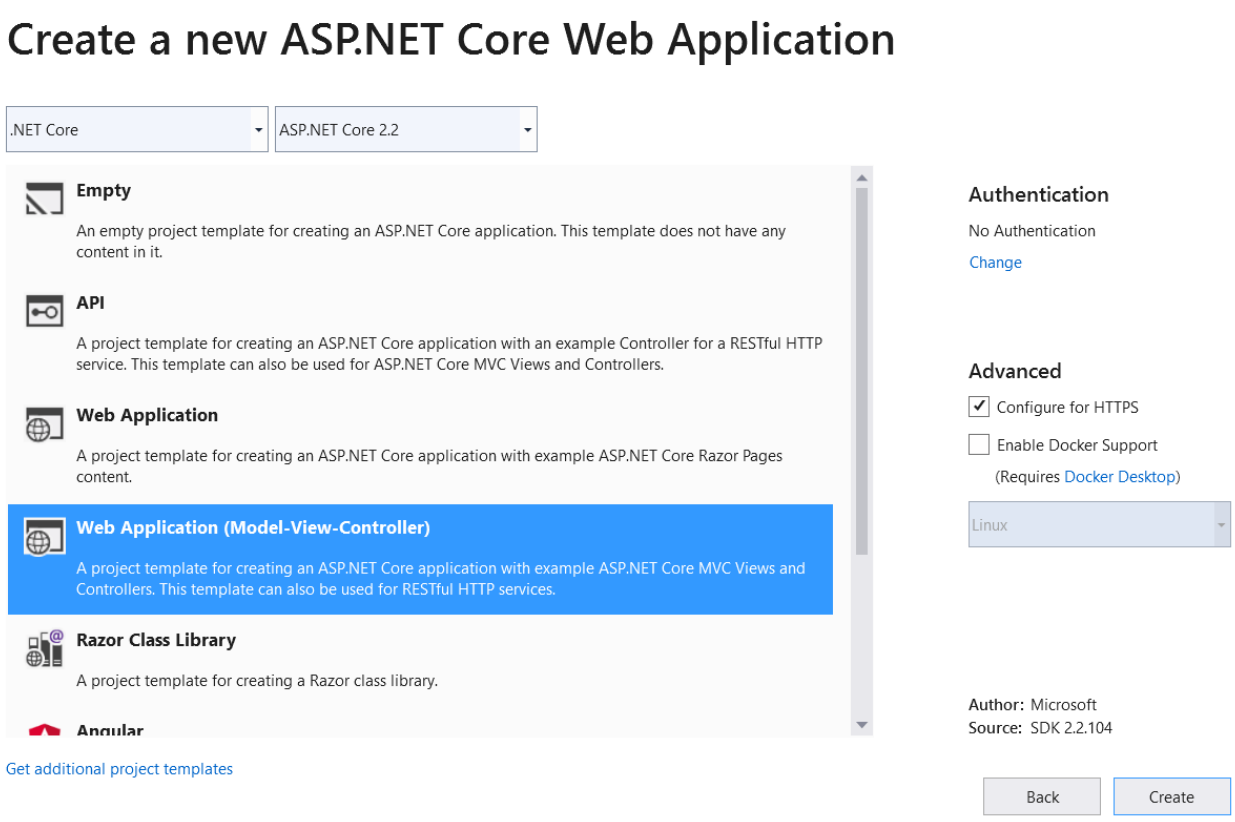
- From the Visual Studio select **Create a new project**.
- Select **ASP.NET Core Web Application** and then select **Next**.



- Name the project **MvcMovie** and select **Create**. It's important to name the project **MvcMovie** so when you copy code, the namespace will match.



- Select **Web Application(Model-View-Controller)**, and then select **Create**.



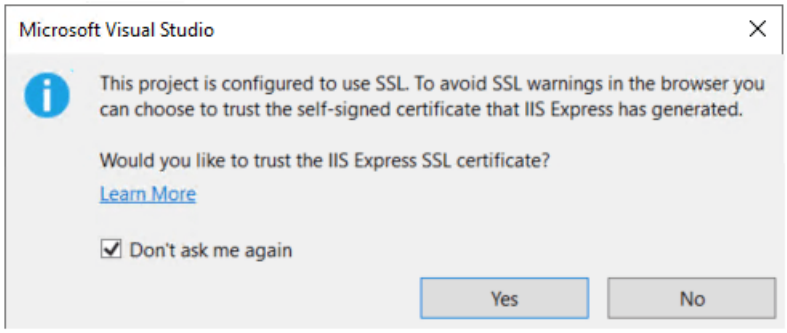
Visual Studio used the default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a basic starter project, and it's a good place to start.

Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

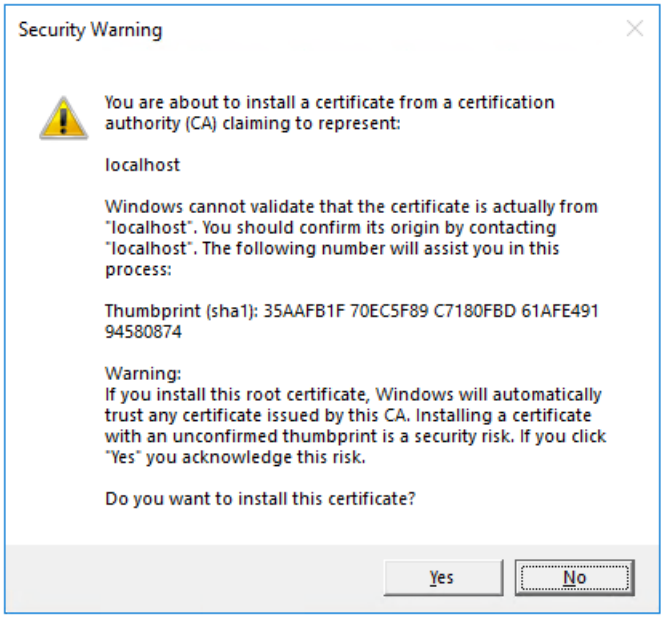
Select **Ctrl-F5** to run the app in non-debug mode.

Visual Studio displays the following dialog:



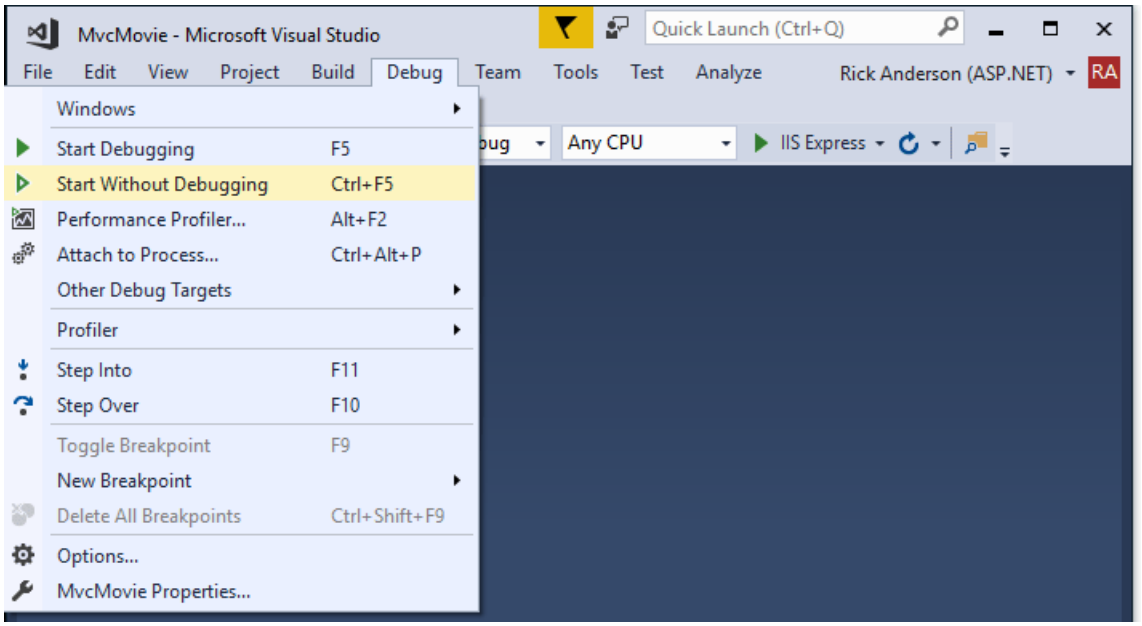
Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:

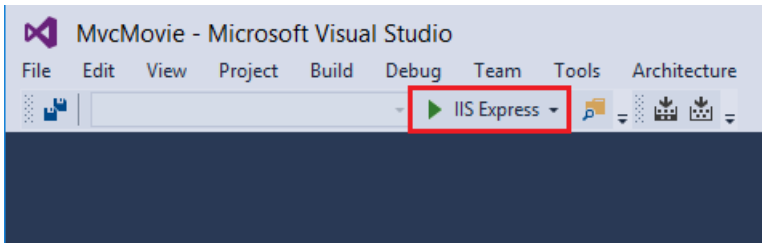


Select **Yes** if you agree to trust the development certificate.

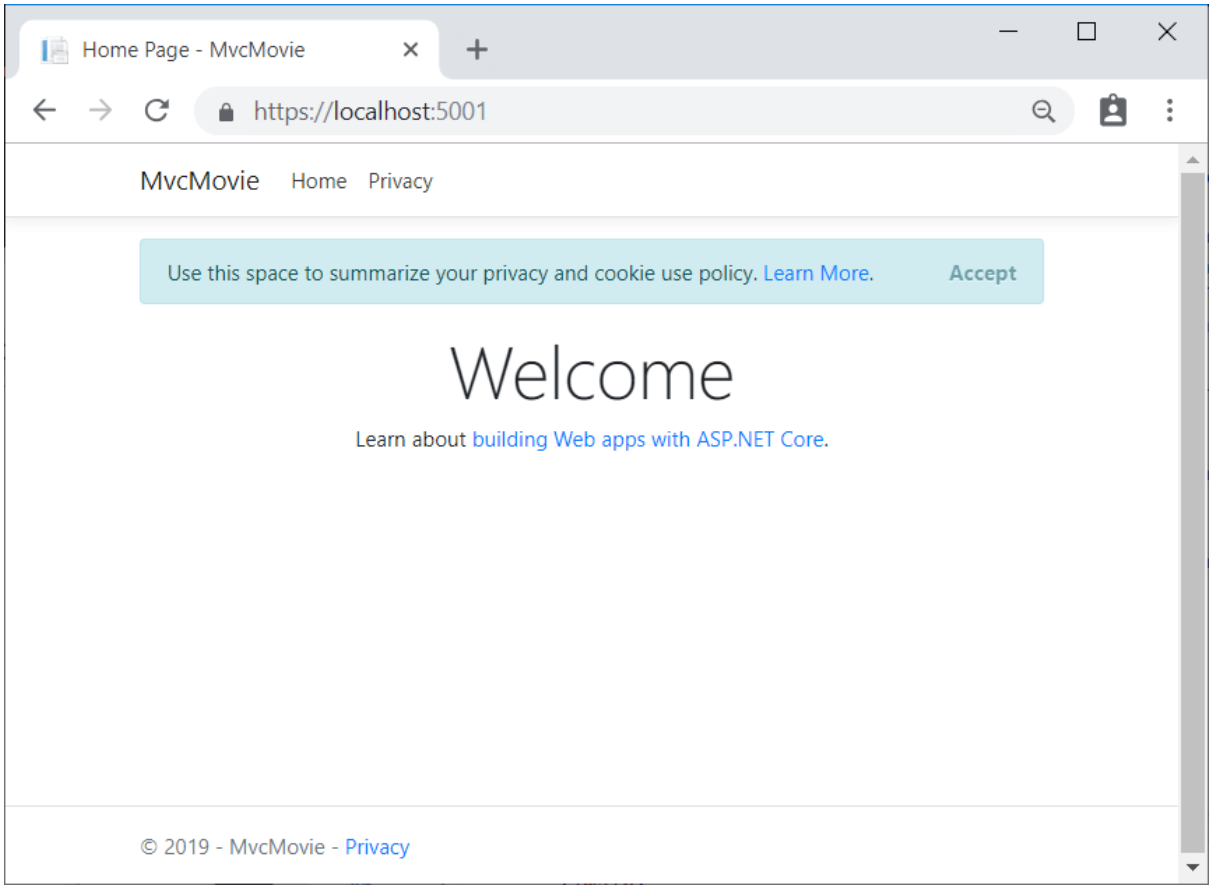
- Visual Studio starts IIS Express and runs the app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for your local computer. When Visual Studio creates a web project, a random port is used for the web server.
- Launching the app with Ctrl+F5 (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
- You can launch the app in debug or non-debug mode from the **Debug** menu item:



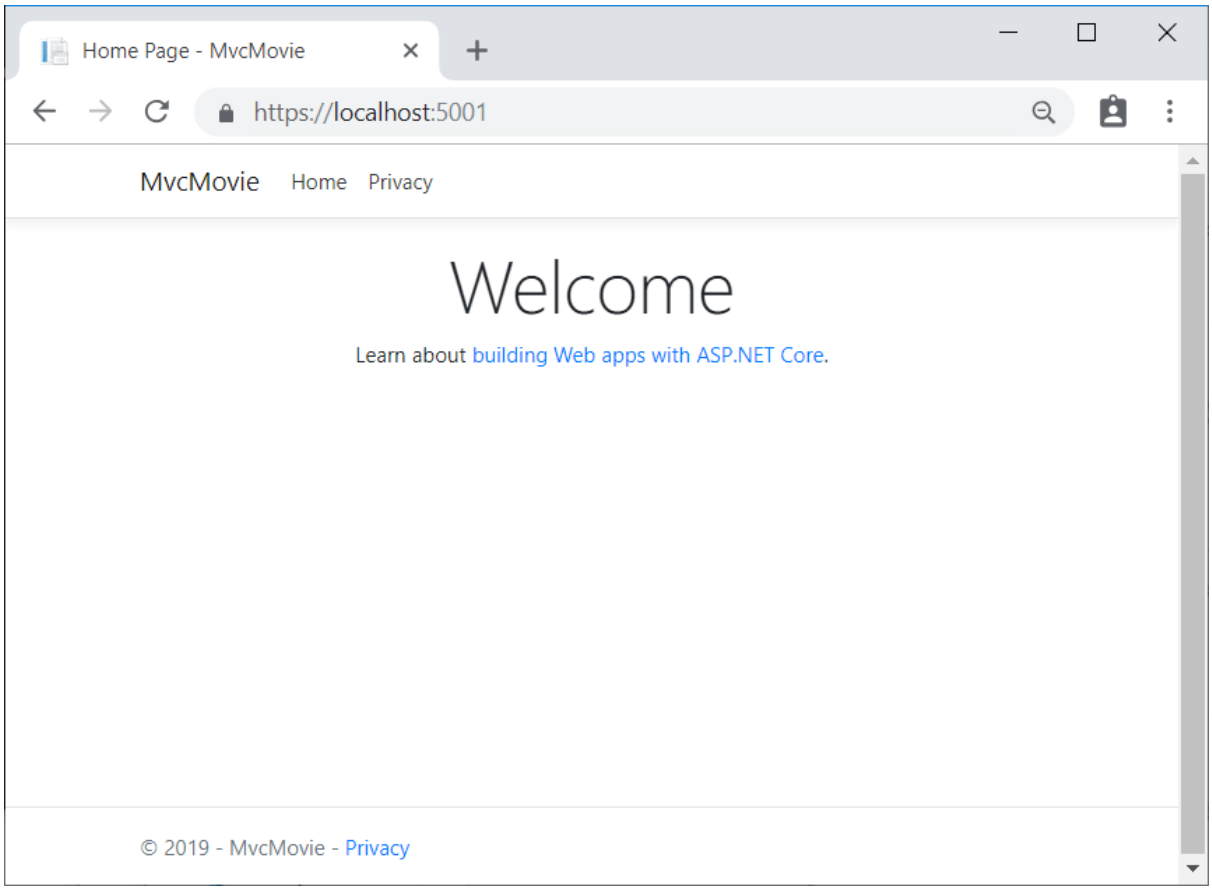
- You can debug the app by selecting the **IIS Express** button



- Select **Accept** to consent to tracking. This app doesn't track personal information. The template generated code includes assets to help meet General Data Protection Regulation (GDPR).



The following image shows the app after accepting tracking:



- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next part of this tutorial, you learn about MVC and start writing some code.

NEXT

Part 2, add a controller to an ASP.NET Core MVC app

9/22/2020 • 12 minutes to read • Edit Online

By Rick Anderson

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps. MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that handle browser requests. They retrieve model data and call view templates that return a response. In an MVC app, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database. For example, `https://localhost:5001/Home/Privacy` has route data of `Home` (the controller) and `Privacy` (the action method to call on the home controller). `https://localhost:5001/Movies/Edit/5` is a request to edit the movie with ID=5 using the movie controller. Route data is explained later in the tutorial.

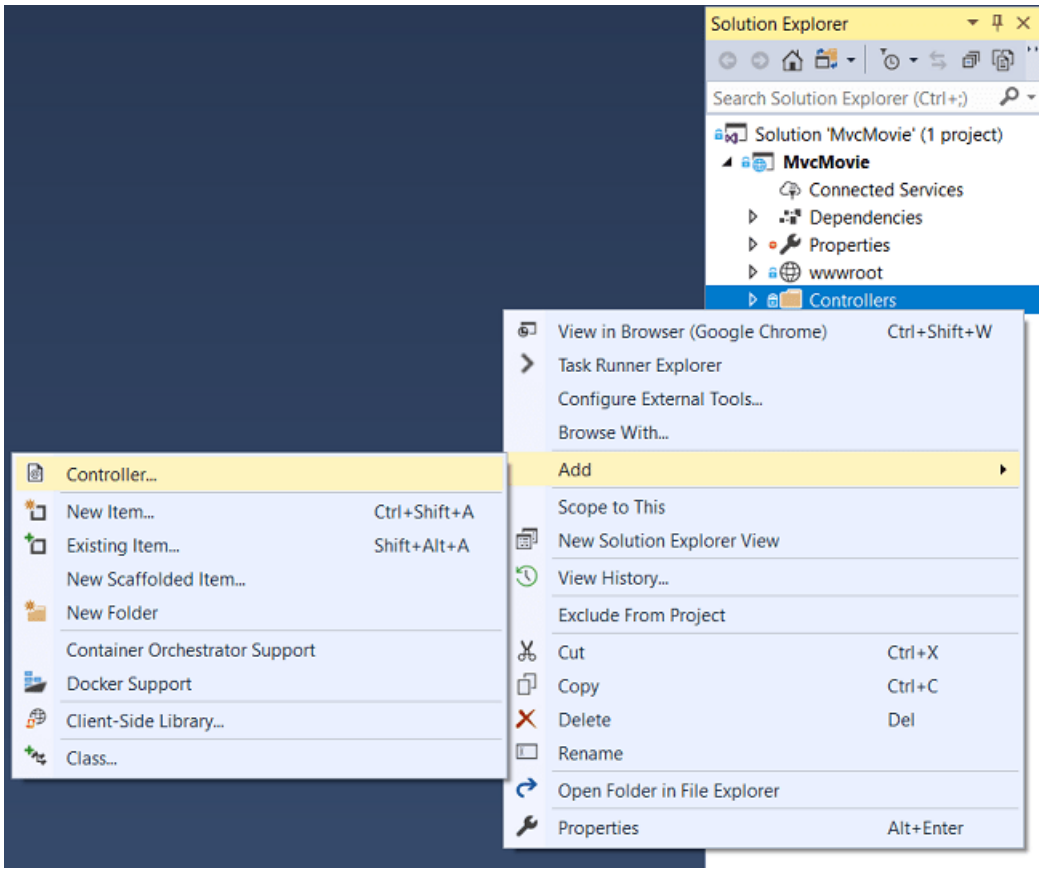
The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

We cover these concepts in this tutorial series and show you how to use them to build a movie app. The MVC project contains folders for the *Controllers* and *Views*.

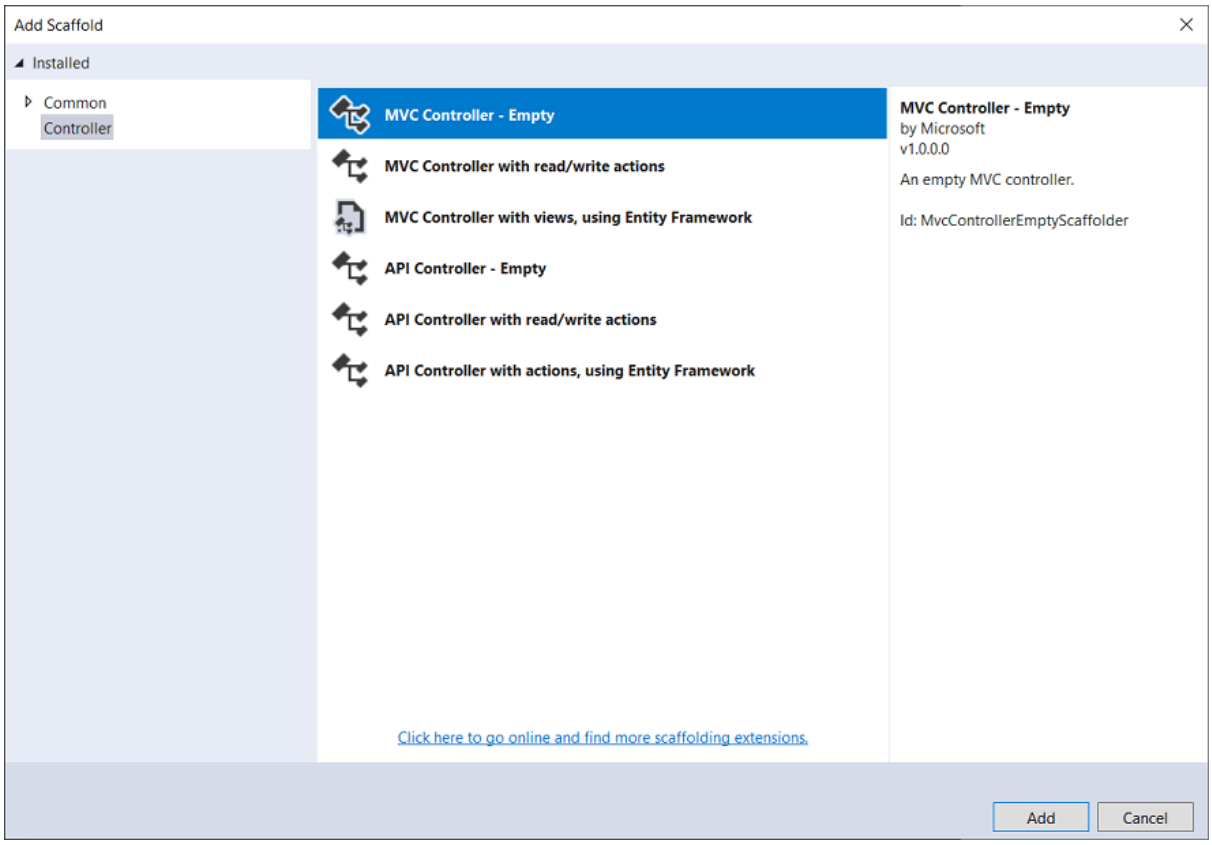
Add a controller

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click **Controllers** > **Add** > **Controller**



- In the **Add Scaffold** dialog box, select **Controller Class - Empty**



- In the **Add Empty MVC Controller dialog**, enter **HelloWorldController** and select **ADD**.

Replace the contents of *Controllers/HelloWorldController.cs* with the following:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

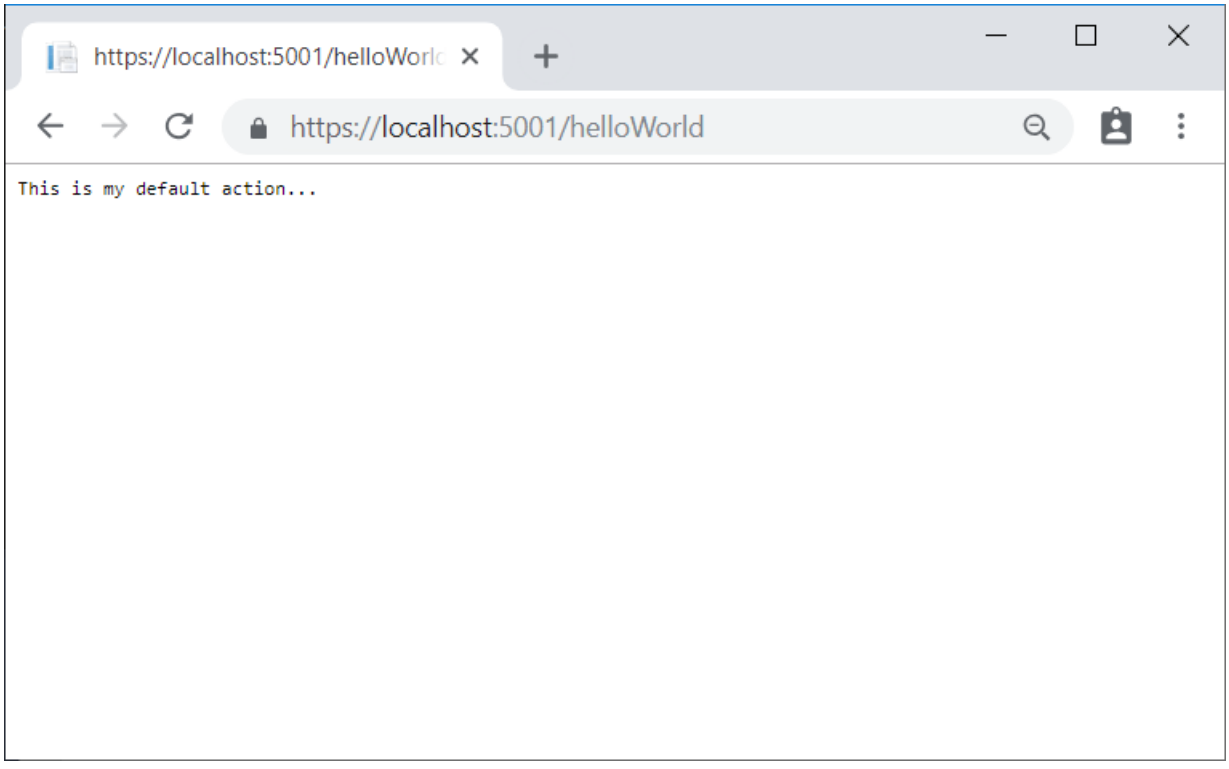
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`, and combines the protocol used: `HTTPS`, the network location of the web server (including the TCP port): `localhost:5001` and the target URI `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL. The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial the scaffolding engine is used to generate `HTTP POST` methods which update data.

Run the app in non-debug mode and append "HelloWorld" to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default URL routing logic used by MVC uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

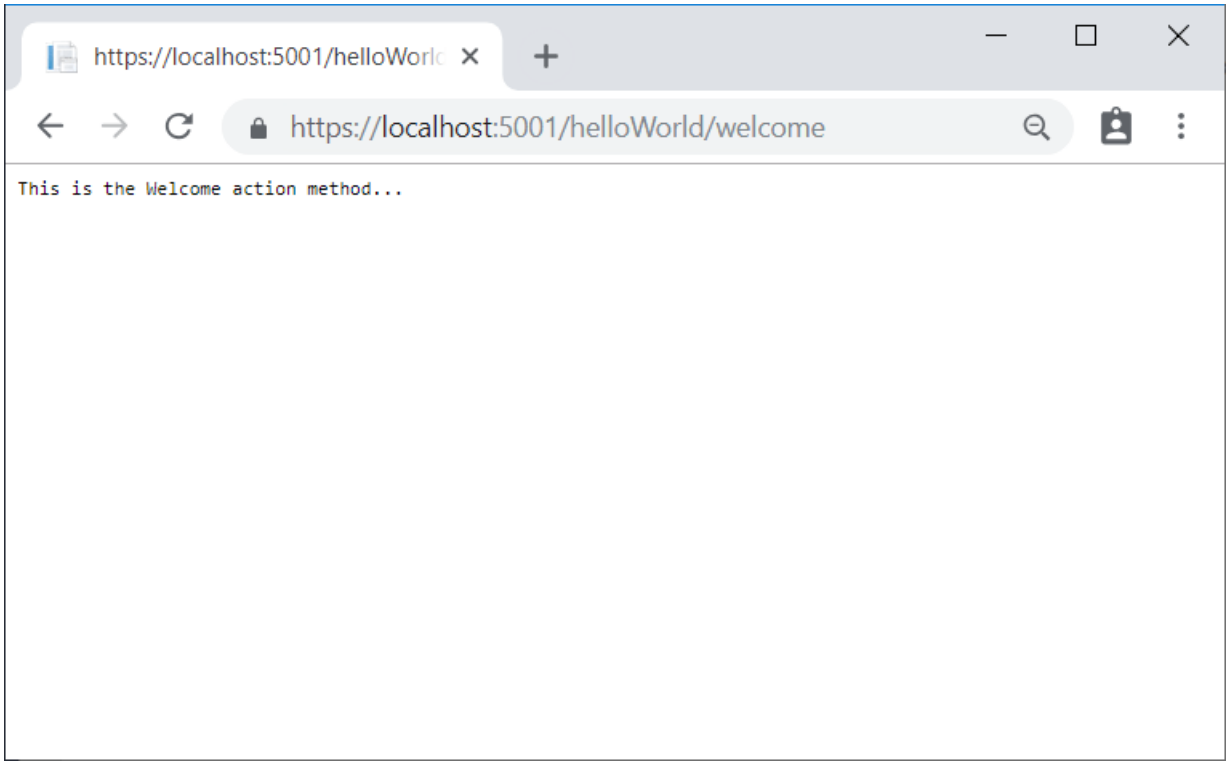
The routing format is set in the `Configure` method in *Startup.cs* file.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above.

The first URL segment determines the controller class to run. So `localhost:{PORT}/HelloWorld` maps to the **HelloWorld**Controller class. The second part of the URL segment determines the action method on the class. So `localhost:{PORT}/HelloWorld/Index` would cause the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:{PORT}/HelloWorld` and the `Index` method was called by default. That's because `Index` is the default method that will be called on a controller if a method name isn't explicitly specified. The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to `https://localhost:{PORT}/HelloWorld/Welcome`. The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`. Change the `Welcome` method to include two parameters as shown in the following code.

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

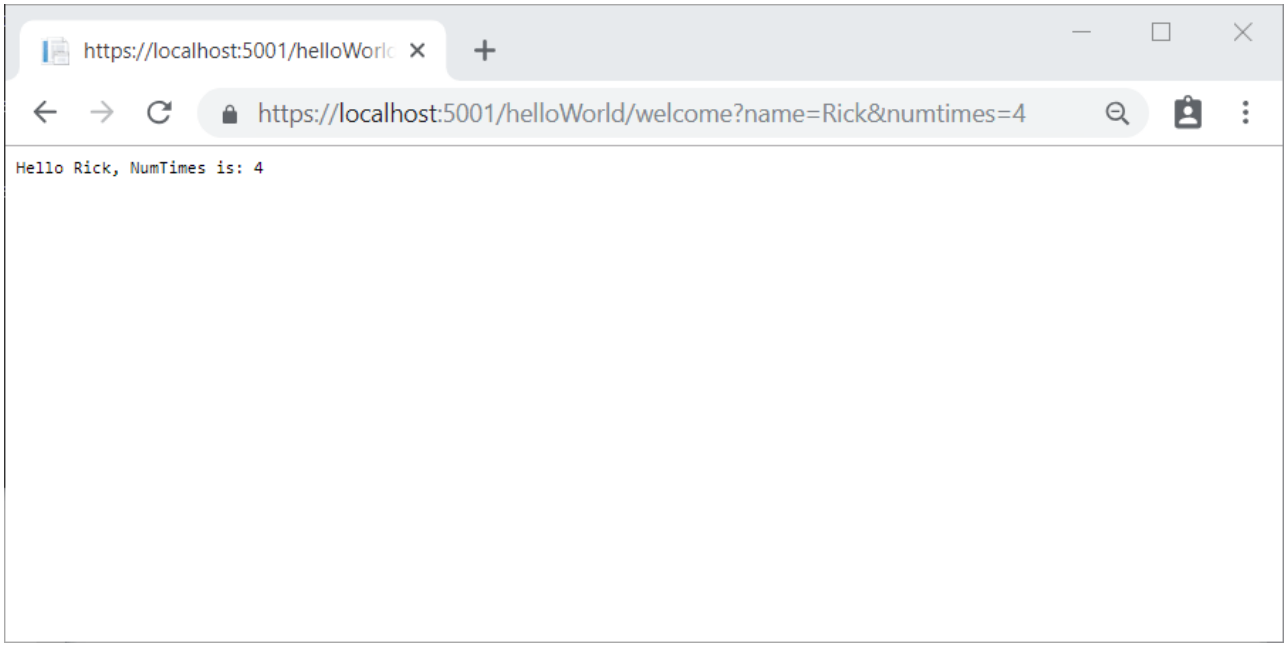
The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript).
- Uses Interpolated Strings in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

(Replace `{PORT}` with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method. See Model Binding for more information.



In the image above, the URL segment (`Parameters`) isn't used, the `name` and `numTimes` parameters are passed in the query string. The `?` (question mark) in the above URL is a separator, and the query string follows. The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick`

This time the third URL segment matched the route parameter `id`. The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method. The trailing `?` (in `id?`) indicates the `id` parameter is optional.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

In these examples the controller has been doing the "VC" portion of MVC - that is, the **V**iew and the **C**ontroller work. The controller is returning HTML directly. Generally you don't want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead you typically use a separate Razor view template file to generate the HTML response. You do that in the next tutorial.

PREVIOUS NEXT

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps. MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that handle browser requests. They retrieve model data and call view templates that return a response. In an MVC app, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database. For example, `https://localhost:5001/Home/About` has route data of `Home` (the controller) and `About` (the action method to call on the home controller). `https://localhost:5001/Movies/Edit/5` is a request to edit the movie with ID=5 using the movie controller. Route data is explained later in the tutorial.

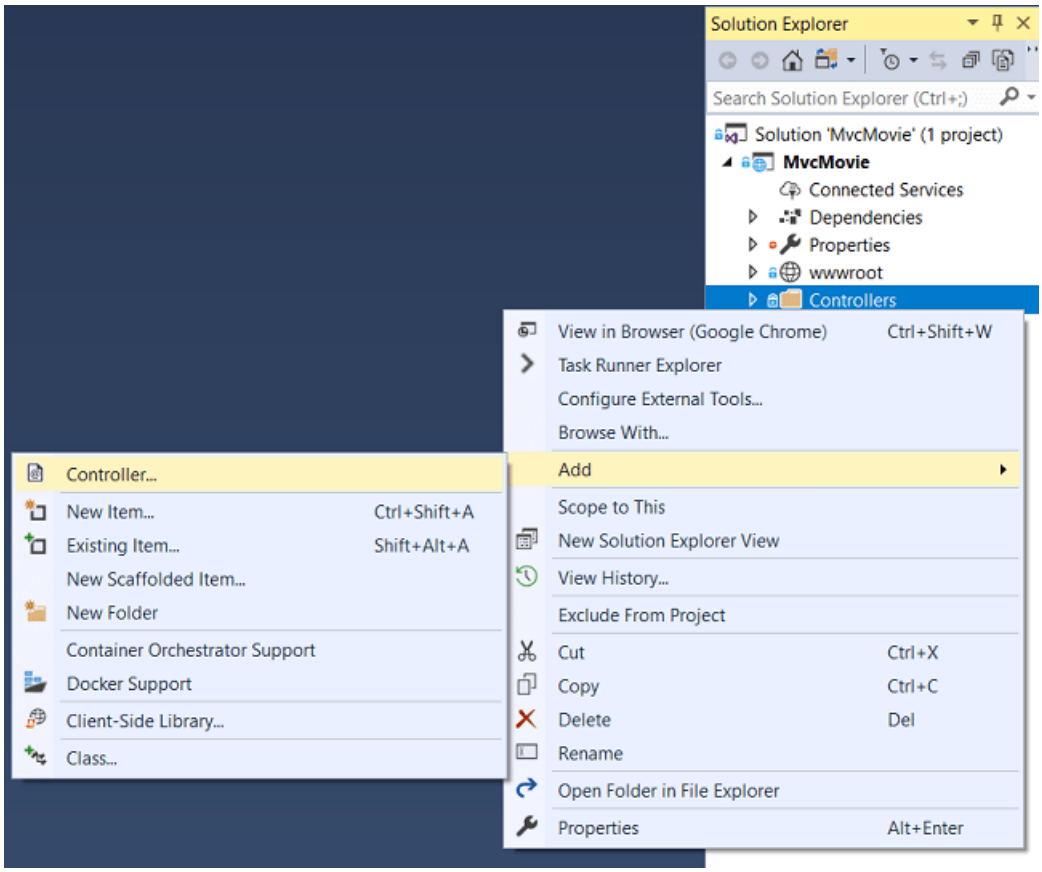
The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

We cover these concepts in this tutorial series and show you how to use them to build a movie app. The MVC project contains folders for the *Controllers* and *Views*.

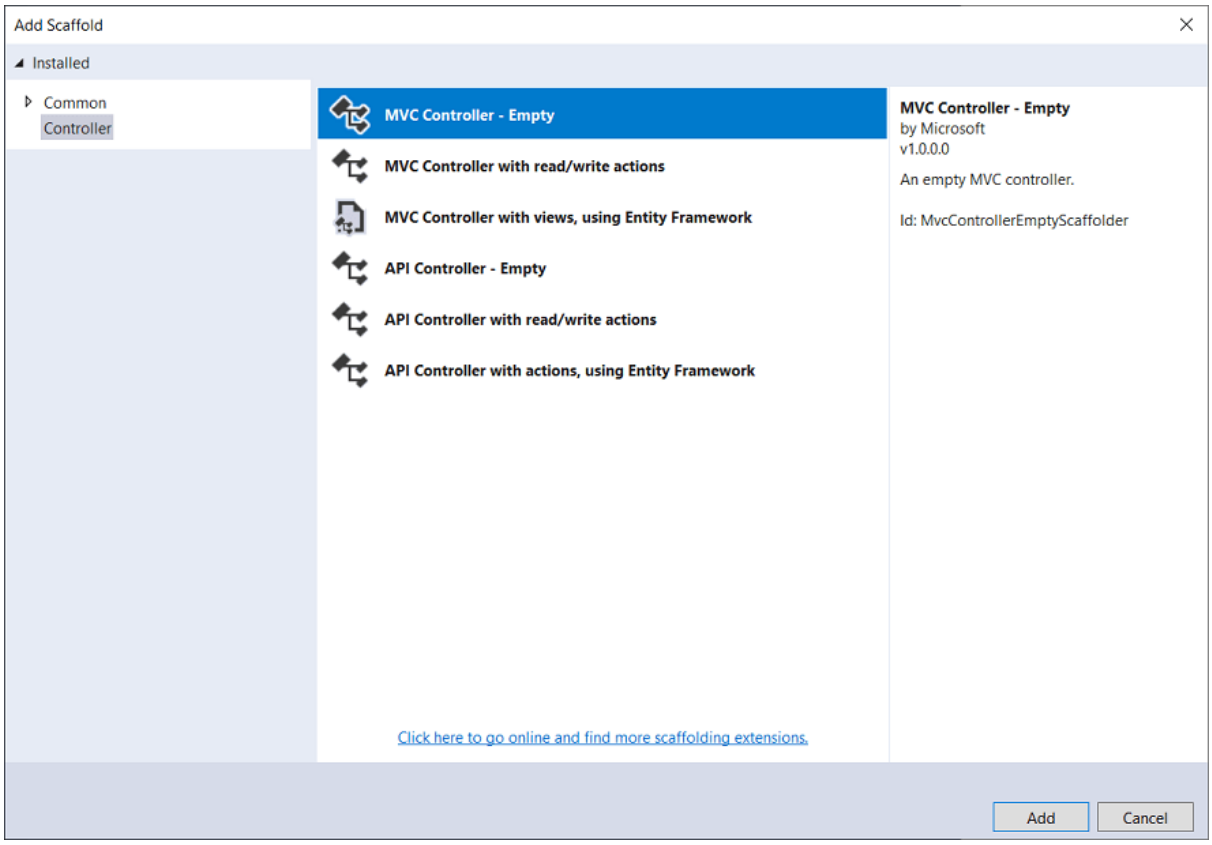
Add a controller

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click **Controllers** > **Add** > **Controller**



- In the **Add Scaffold** dialog box, select **MVC Controller - Empty**



- In the **Add Empty MVC Controller dialog**, enter **HelloWorldController** and select **ADD**.

Replace the contents of *Controllers/HelloWorldController.cs* with the following:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

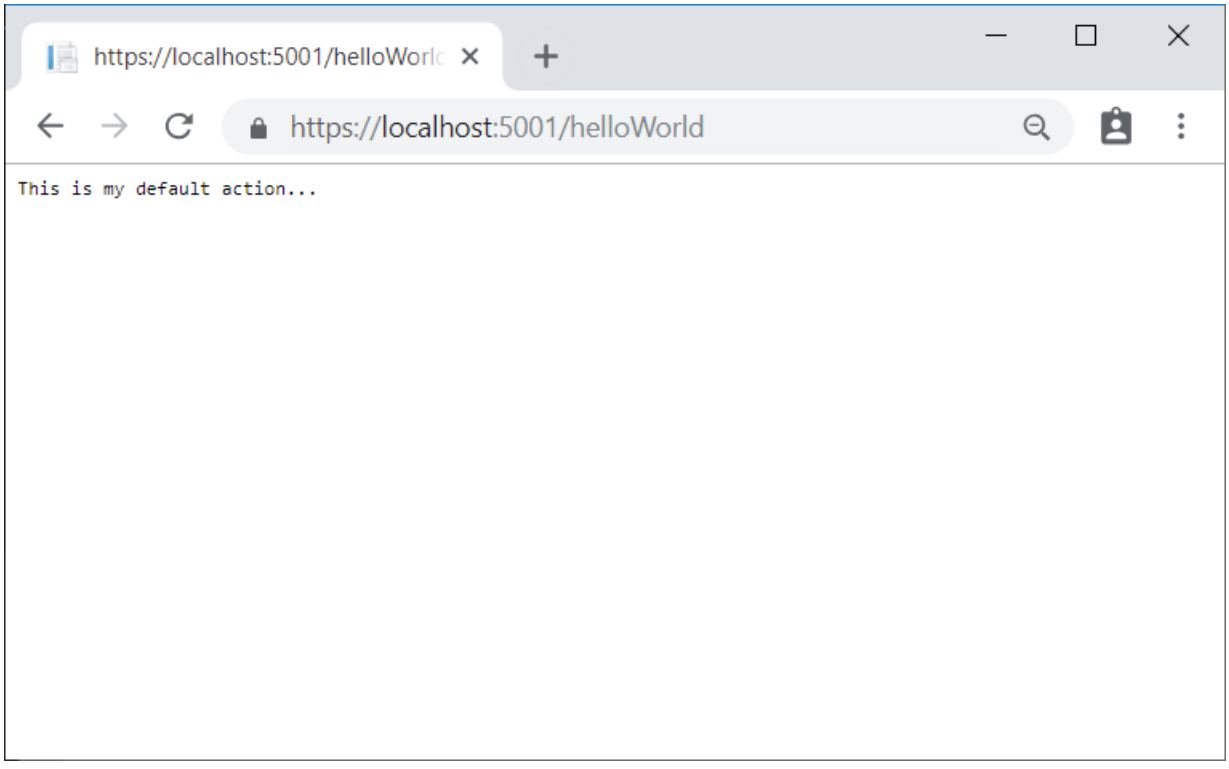
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`, and combines the protocol used: `HTTPS`, the network location of the web server (including the TCP port): `localhost:5001` and the target URI `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL. The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial the scaffolding engine is used to generate `HTTP POST` methods which update data.

Run the app in non-debug mode and append "HelloWorld" to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default URL routing logic used by MVC uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

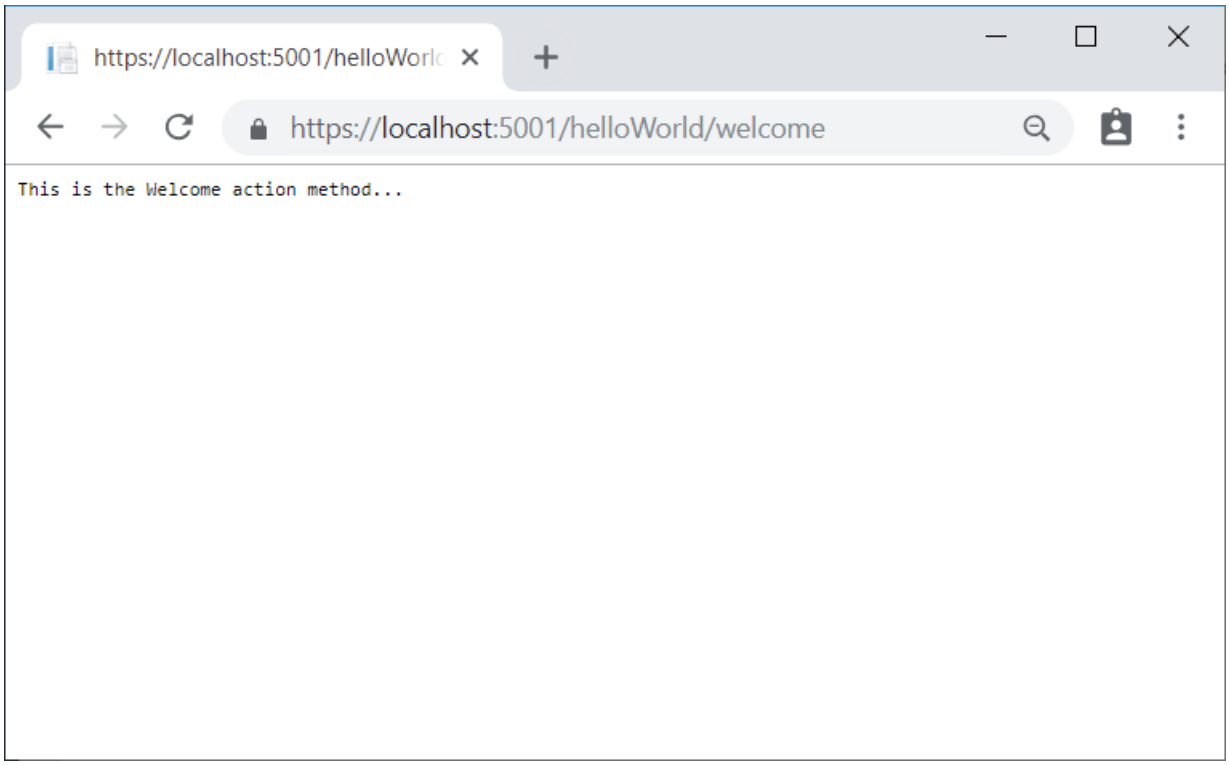
The routing format is set in the `Configure` method in *Startup.cs* file.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above.

The first URL segment determines the controller class to run. So `localhost:{PORT}/HelloWorld` maps to the `HelloWorldController` class. The second part of the URL segment determines the action method on the class. So `localhost:{PORT}/HelloWorld/Index` would cause the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:{PORT}/HelloWorld` and the `Index` method was called by default. This is because `Index` is the default method that will be called on a controller if a method name isn't explicitly specified. The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to `https://localhost:{PORT}/HelloWorld/Welcome`. The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`. Change the `Welcome` method to include two parameters as shown in the following code.

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

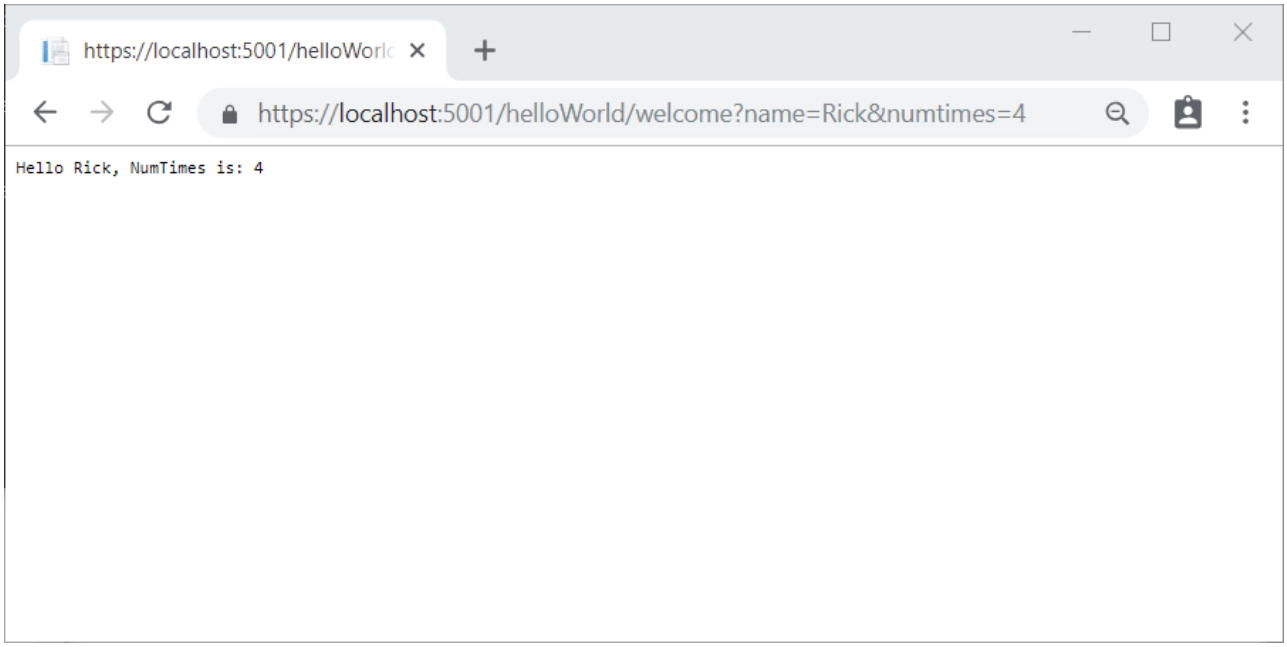
The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript).
- Uses Interpolated Strings in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

(Replace `{PORT}` with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method. See Model Binding for more information.



In the image above, the URL segment (`Parameters`) isn't used, the `name` and `numTimes` parameters are passed in the query string. The `?` (question mark) in the above URL is a separator, and the query string follows. The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick`

This time the third URL segment matched the route parameter `id`. The `Welcome` method contains a parameter `id` that matched the URL template in the `MapRoute` method. The trailing `?` (in `id?`) indicates the `id` parameter is optional.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

In these examples the controller has been doing the "VC" portion of MVC - that is, the view and controller work. The controller is returning HTML directly. Generally you don't want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead you typically use a separate Razor view template file to help generate the HTML response. You do that in the next tutorial.

PREVIOUS NEXT

Part 3, add a view to an ASP.NET Core MVC app

9/22/2020 • 16 minutes to read • Edit Online

By Rick Anderson

In this section you modify the `HelloWorldController` class to use Razor view files to cleanly encapsulate the process of generating HTML responses to a client.

You create a view template file using Razor. Razor-based view templates have a *.cshtml* file extension. They provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message that's hard-coded in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

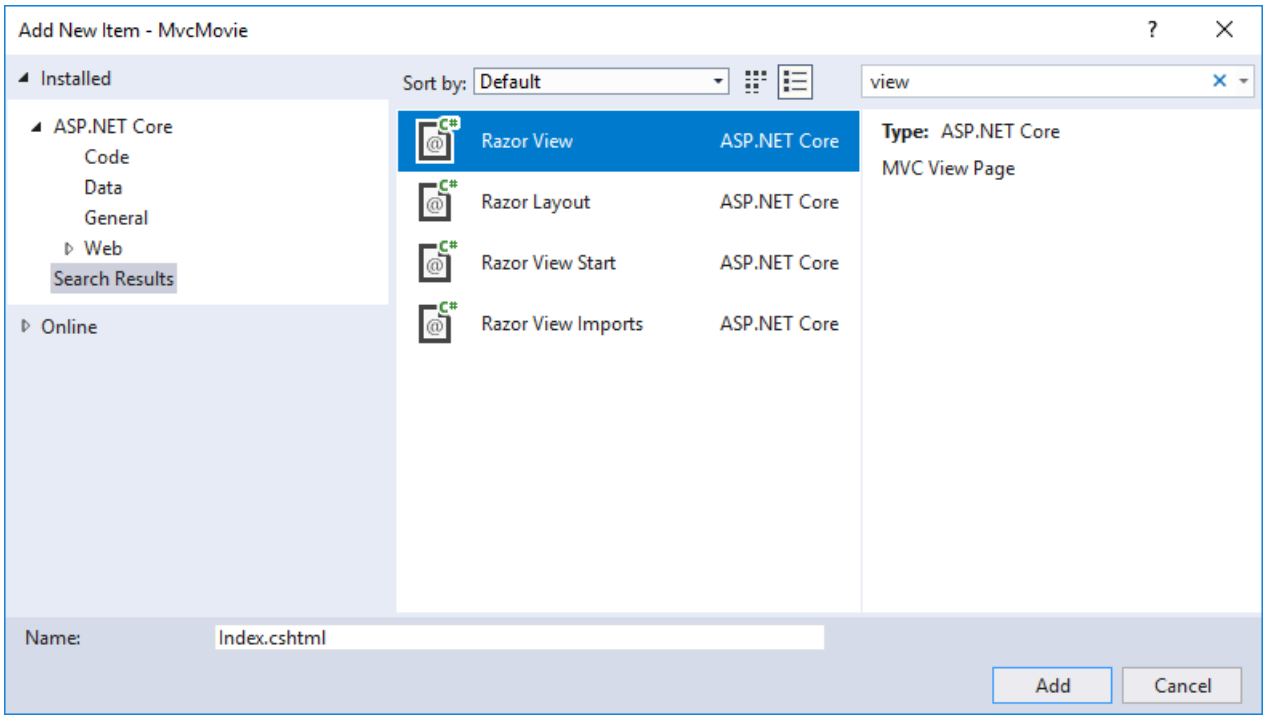
```
public IActionResult Index()
{
    return View();
}
```

The preceding code calls the controller's View method. It uses a view template to generate an HTML response. Controller methods (also known as *action methods*), such as the `Index` method above, generally return an IActionResult (or a class derived from ActionResult), not a type like `string`.

Add a view

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Right click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.
- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item - MvcMovie** dialog
  - In the search box in the upper-right, enter *view*
  - Select **Razor View**
  - Keep the **Name** box value, *Index.cshtml*.
  - Select **Add**



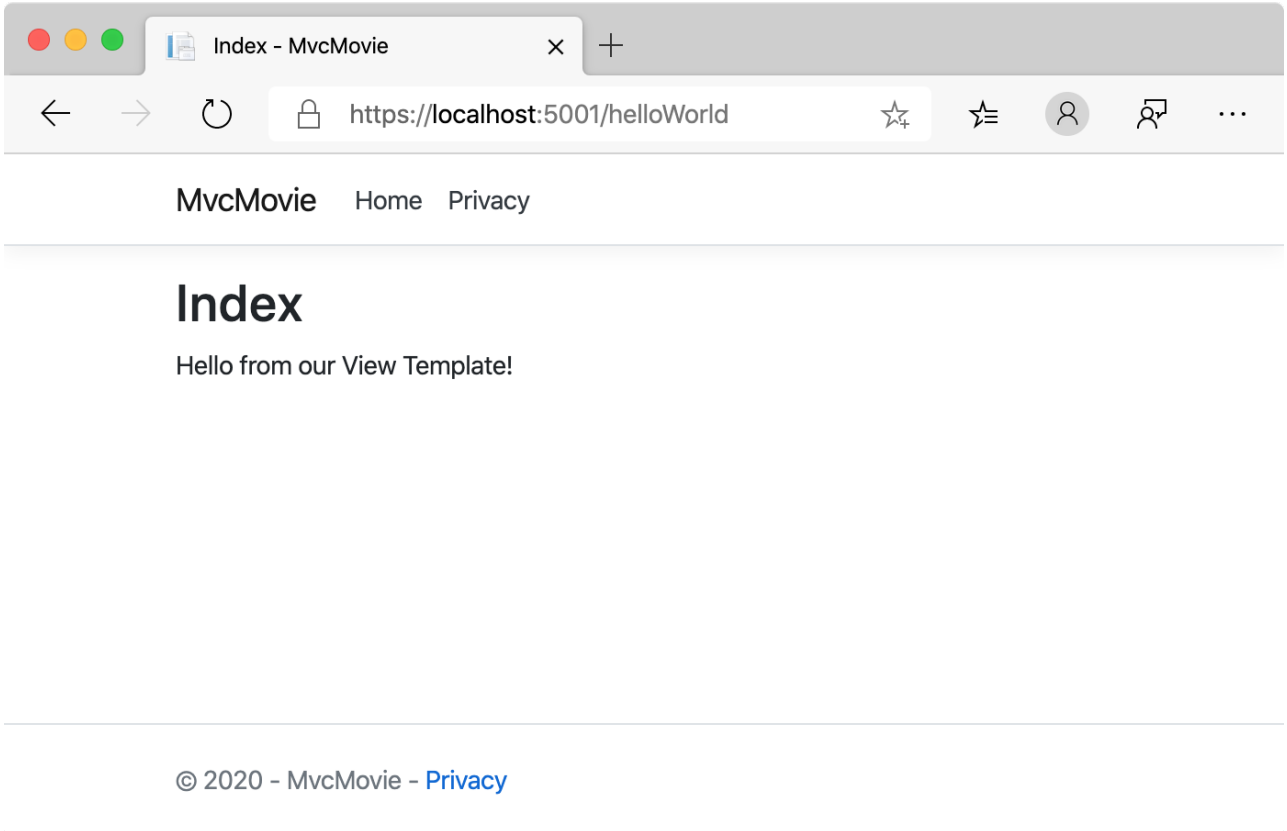
Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://localhost:{PORT}/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much; it ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because a view template file name wasn't specified, MVC defaulted to using the default view file. The default view file has the same name as the method (`Index`), so the view template in */Views/HelloWorld/Index.cshtml* is used. The image below shows the string "Hello from our View Template!" hard-coded in the view.



Change views and layout pages

Select the menu links (**MvcMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Views/Shared/_Layout.cshtml* file. Open the *Views/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the **Views/Home/Privacy.cshtml** view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

Replace the content of the *Views/Shared/_Layout.cshtml* file with the following markup. The changes are highlighted:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow
mb-3">
            <div class="container">
                <a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2020 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @RenderSection("Scripts", required: false)
</body>
</html>
```

The preceding markup made the following changes:

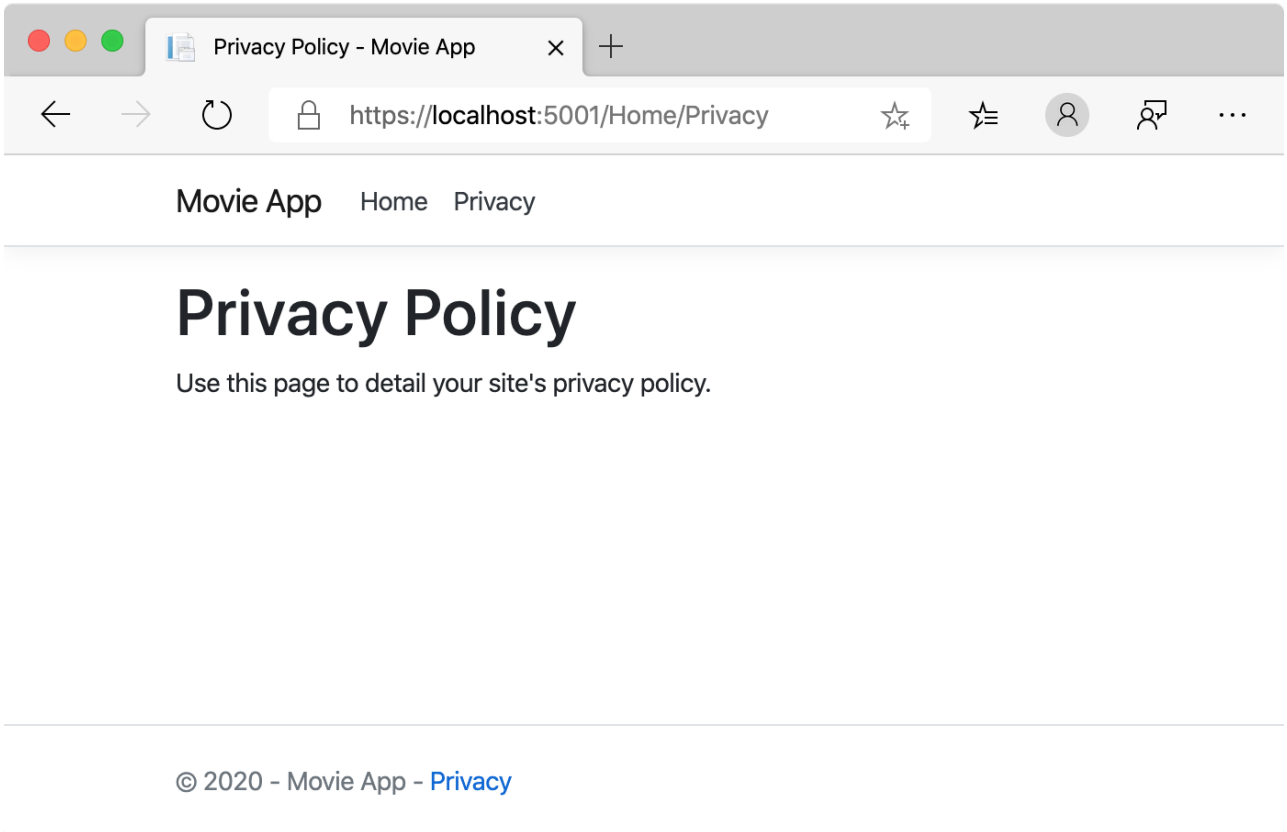
- 3 occurrences of `MvcMovie` to `Movie App`.
- The anchor element `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

In the preceding markup, the `asp-area=""` anchor Tag Helper attribute and attribute value was omitted because this app is not using Areas.

Note: The `Movies` controller has not been implemented. At this point, the `Movie App` link is not functional.

Save your changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy** -

Movie App instead of **Privacy Policy - Mvc Movie**:



Select the **Home** link and notice that the title and anchor text also display **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the *Views/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Change the title and `<h2>` element of the *Views/HelloWorld/Index.cshtml* view file:

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

The title and `<h2>` element are slightly different so you can see which bit of code changes the display.

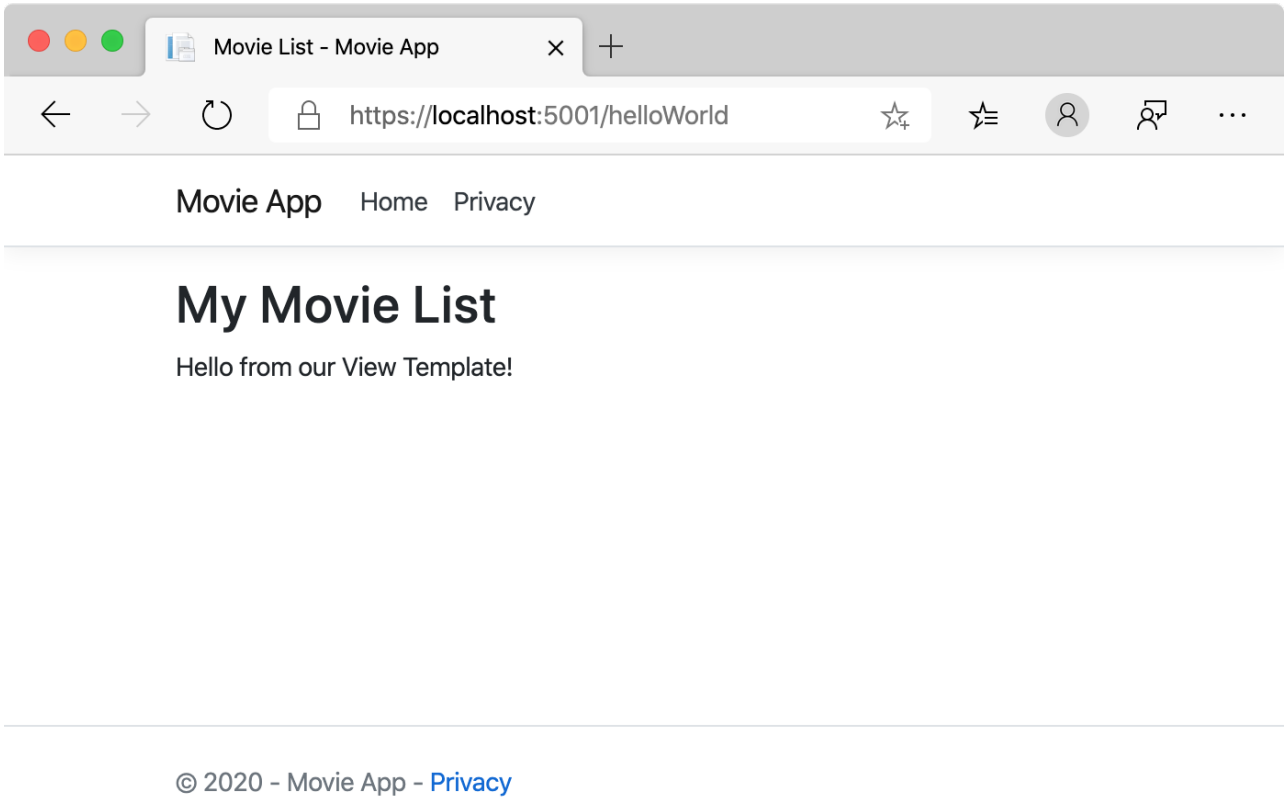
`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

Save the change and navigate to `https://localhost:{PORT}/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing

cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with `ViewData["Title"]` we set in the *Index.cshtml* view template and the additional "- Movie App" added in the layout file.

The content in the *Index.cshtml* view template is merged with the *Views/Shared/_Layout.cshtml* view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app. To learn more, see Layout.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code clean, testable, and maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and a `ID` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate bits of data must be passed from the controller to the view in order to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary that the view template can then access.

In *HelloWorldController.cs*, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means any type can be used; the `ViewData` object

has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete *HelloWorldController.cs* file looks like this:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named *Views/HelloWorld/Welcome.cshtml*.

You'll create a loop in the *Welcome.cshtml* view template that displays "Hello" `NumTimes` . Replace the contents of *Views/HelloWorld/Welcome.cshtml* with the following:

```
@{
    ViewData["Title"] = "Welcome";
}

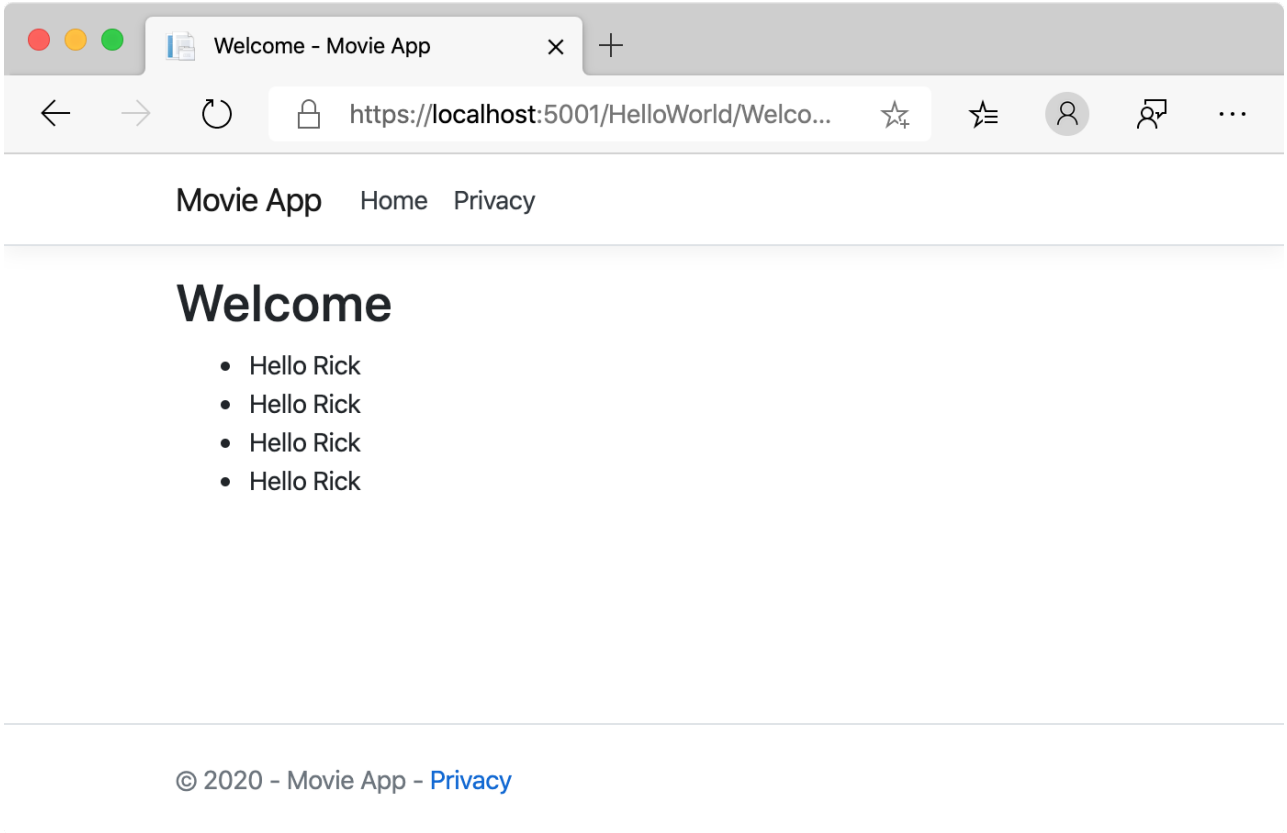
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder . The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the sample above, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See When to use ViewBag, ViewData, or TempData for more information.

In the next tutorial, a database of movies is created.

PREVIOUS NEXT

In this section you modify the `HelloWorldController` class to use Razor view files to cleanly encapsulate the process of generating HTML responses to a client.

You create a view template file using Razor. Razor-based view templates have a *.cshtml* file extension. They provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message that's hard-coded in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

```
public IActionResult Index()
{
    return View();
}
```

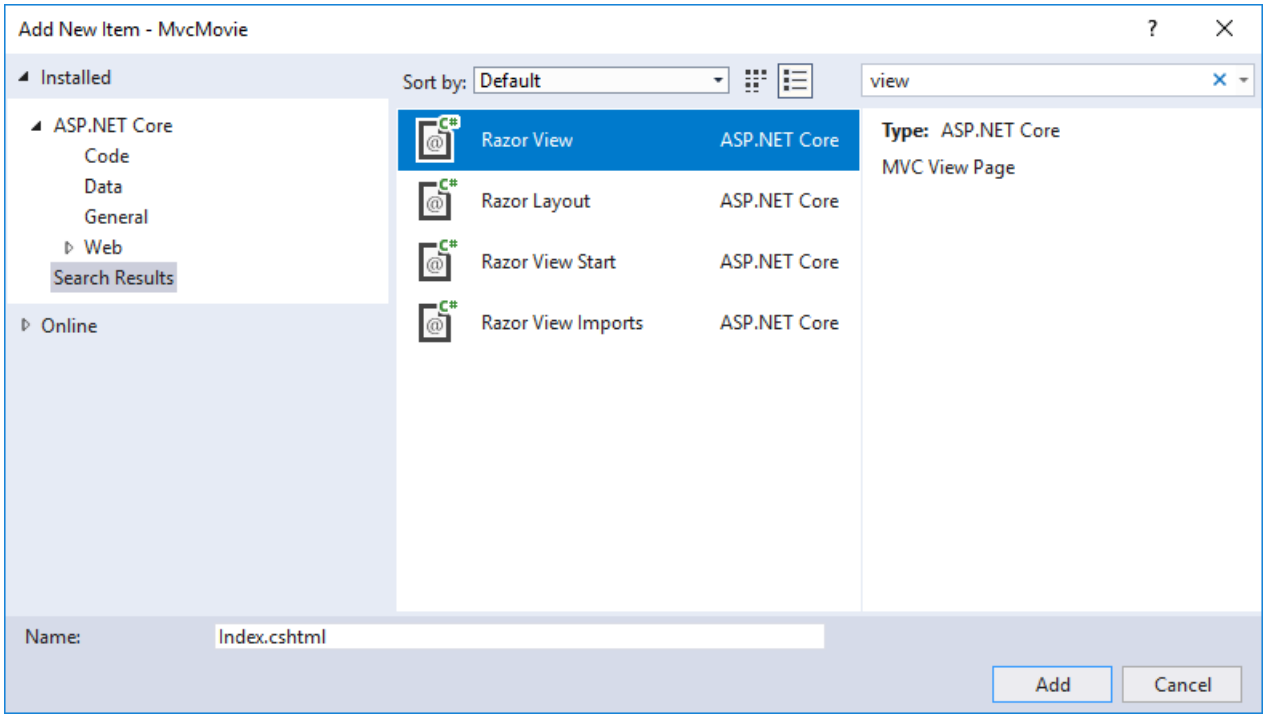
The preceding code calls the controller's View method. It uses a view template to generate an HTML response. Controller methods (also known as *action methods*), such as the `Index` method above, generally return an IActionResult (or a class derived from ActionResult), not a type like `string`.

Add a view

- Visual Studio

- Visual Studio Code
- Visual Studio for Mac

- Right click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.
- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item - MvcMovie** dialog
  - In the search box in the upper-right, enter *view*
  - Select **Razor View**
  - Keep the **Name** box value, *Index.cshtml*.
  - Select **Add**



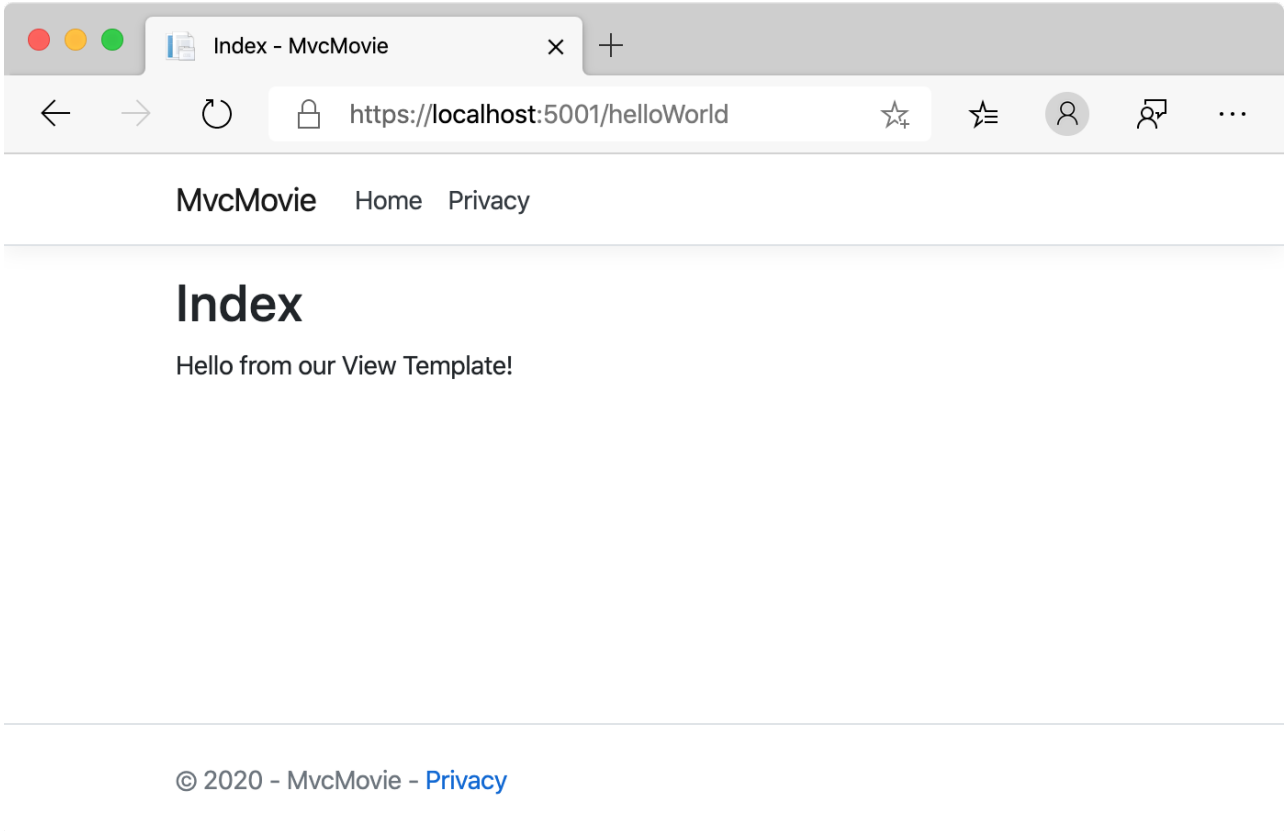
Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://localhost:{PORT}/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much; it ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because a view template file name wasn't specified, MVC defaulted to using the default view file. The default view file has the same name as the method (`Index`), so in the */Views/HelloWorld/Index.cshtml* is used. The image below shows the string "Hello from our View Template!" hard-coded in the view.



Change views and layout pages

Select the menu links (**MvcMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Views/Shared/_Layout.cshtml* file. Open the *Views/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the **Views/Home/Privacy.cshtml** view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

- In the title and footer elements, change `MvcMovie` to `Movie App`.
- Change the anchor element
  `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to
  `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

The following markup shows the highlighted changes:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute"
```

```
value= absolute
              crossorigin="anonymous"
              integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE="/>
    </environment>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow
mb-3">
            <div class="container">
                <a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>

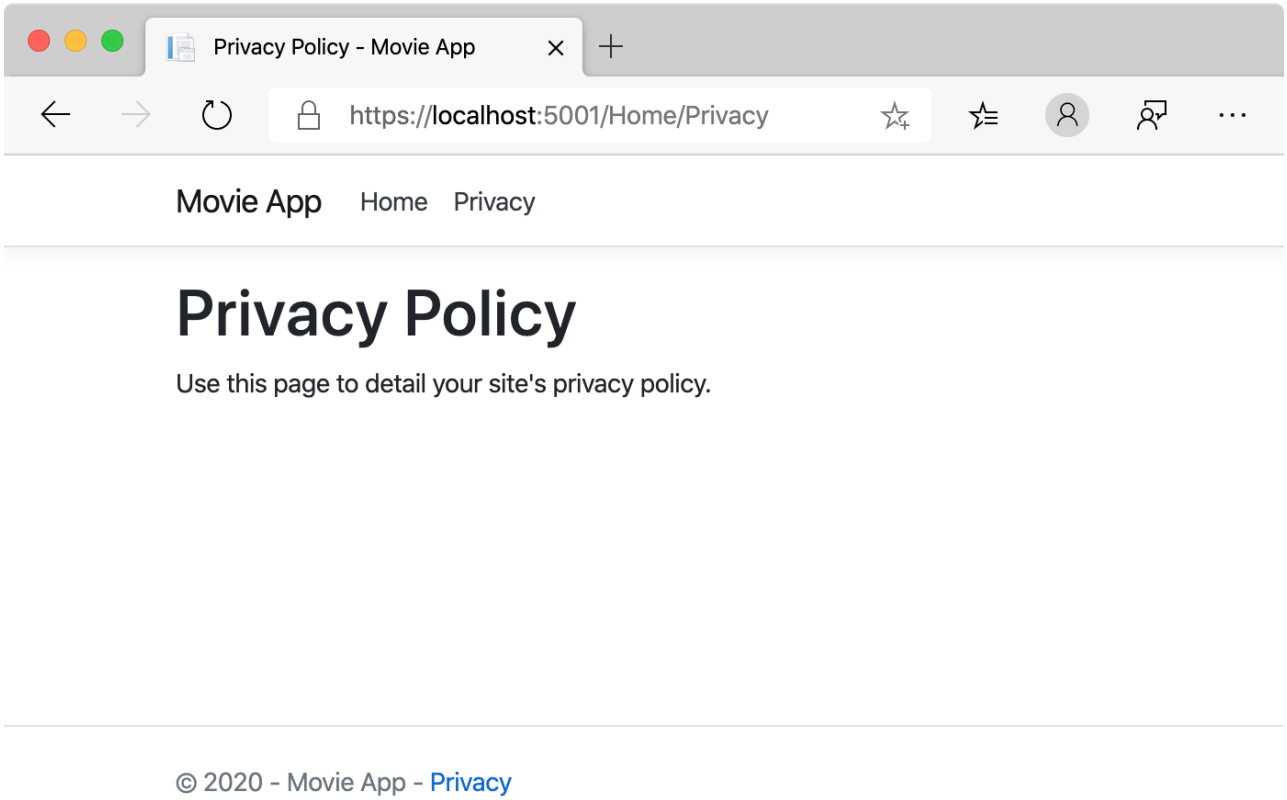
    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">
        </script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/js/bootstrap.bundle.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha256-E/V4cWE4qvAeO5MOhjtGtqDzPndRO1LBk8lJ/PR7CA4=">
        </script>
    </environment>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

In the preceding markup, the `asp-area` anchor Tag Helper attribute was omitted because this app is not using Areas.

Note: The `Movies` controller has not been implemented. At this point, the `Movie App` link is not functional.

Save your changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - Mvc Movie**:



Select the **Home** link and notice that the title and anchor text also display **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the *Views/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Change the title and `<h2>` element of the *Views/HelloWorld/Index.cshtml* view file:

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

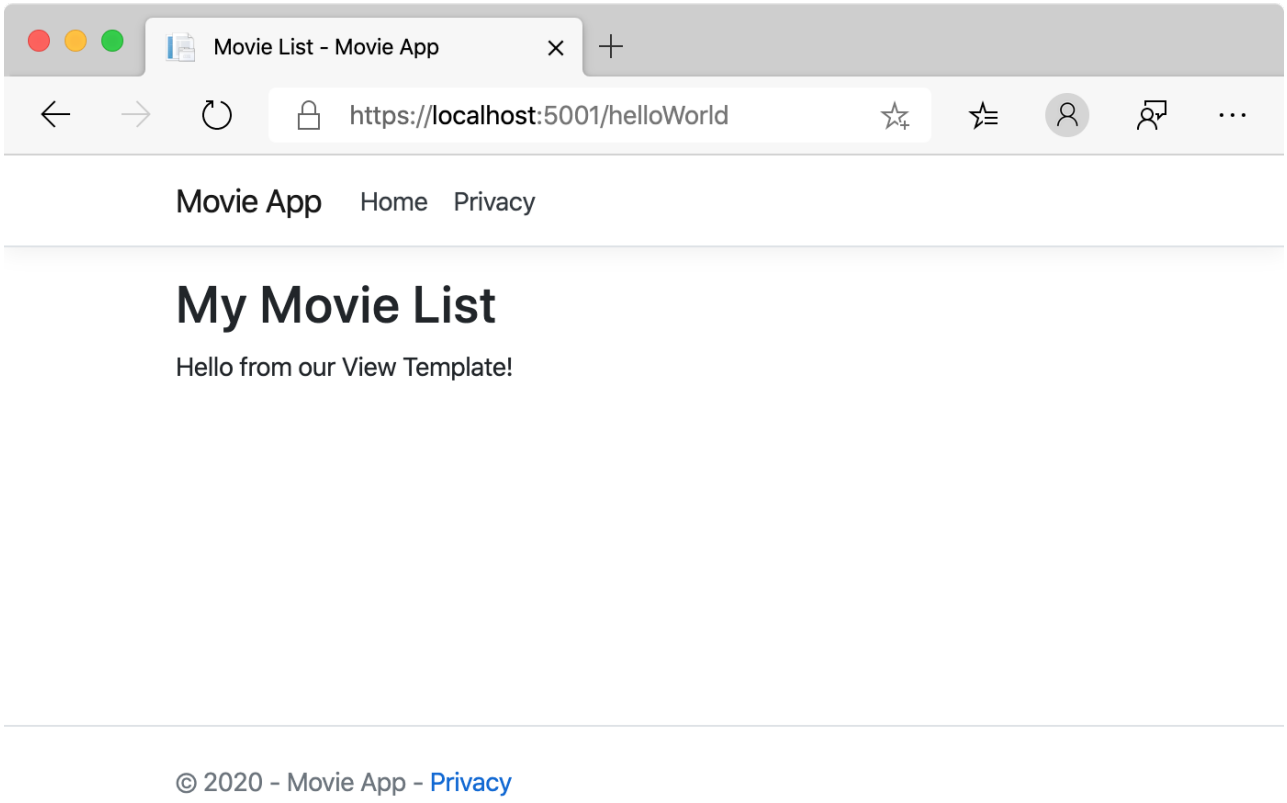
The title and `<h2>` element are slightly different so you can see which bit of code changes the display.

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

Save the change and navigate to `https://localhost:{PORT}/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with `ViewData["Title"]` we set in the *Index.cshtml* view template and the additional "- Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *Views/Shared/_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application. To learn more see Layout.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code clean, testable, and maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and a `ID` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate bits of data must be passed from the controller to the view in order to generate the response. Do this

by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary that the view template can then access.

In *HelloWorldController.cs*, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means any type can be used; the `ViewData` object has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete *HelloWorldController.cs* file looks like this:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named *Views/HelloWorld/Welcome.cshtml*.

You'll create a loop in the *Welcome.cshtml* view template that displays "Hello" `NumTimes`. Replace the contents of *Views/HelloWorld/Welcome.cshtml* with the following:

```
@{
    ViewData["Title"] = "Welcome";
}

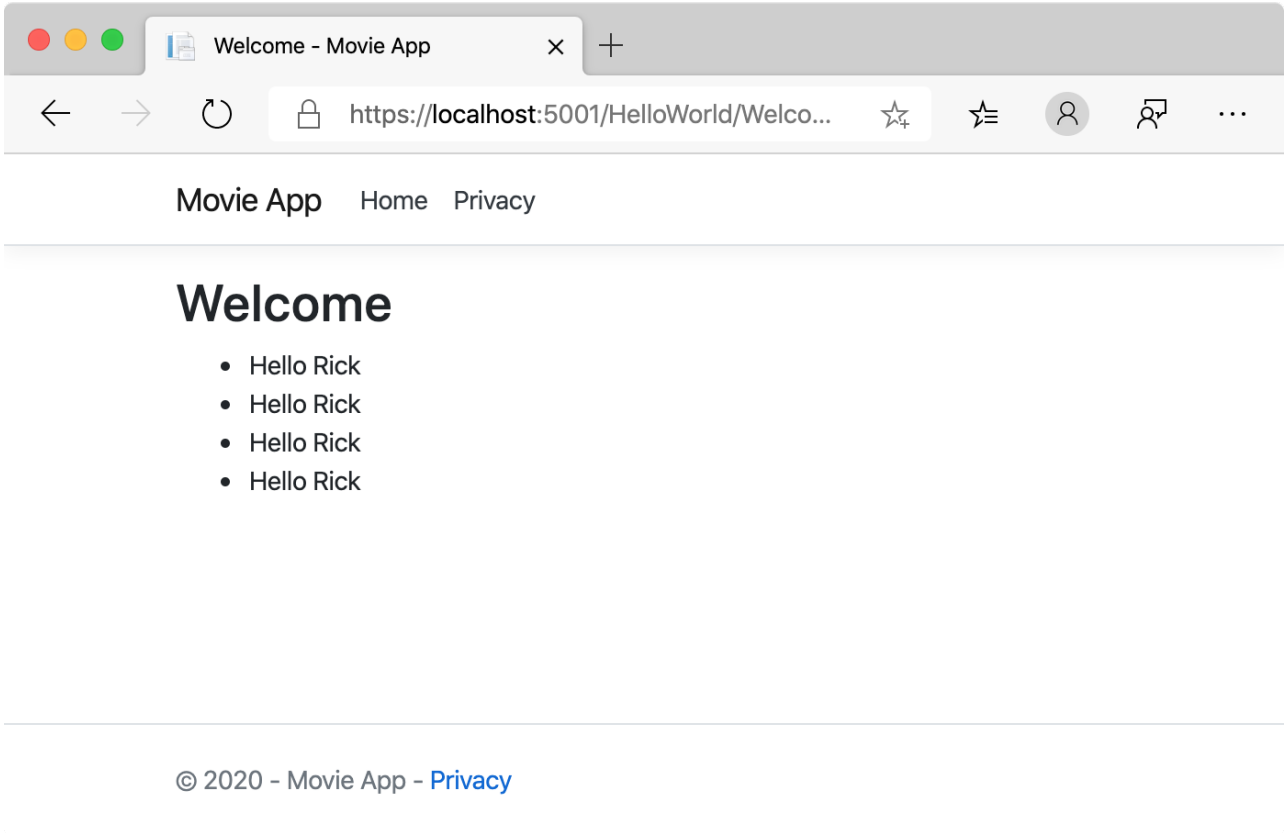
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder . The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the sample above, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See When to use ViewBag, ViewData, or TempData for more information.

In the next tutorial, a database of movies is created.

PREVIOUS NEXT

Part 4, add a model to an ASP.NET Core MVC app

9/22/2020 • 28 minutes to read • Edit Online

By Rick Anderson and Tom Dykstra

In this section, you add classes for managing movies in a database. These classes will be the "**M**odel" part of the **M**VC app.

You use these classes with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes you create are known as POCO classes (from **P**lain **O**ld **C**LR **O**bjects) because they don't have any dependency on EF Core. They just define the properties of the data that will be stored in the database.

In this tutorial, you write the model classes first, and EF Core creates the database.

Add a data model class

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Right-click the *Models* folder > **Add** > **Class**. Name the file *Movie.cs*.

Update the *Movie.cs* file with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains an `Id` field, which is required by the database for the primary key.

The DataType attribute on `ReleaseDate` specifies the type of the data (`Date`). With this attribute:

- The user is not required to enter time information in the date field.
- Only the date is displayed, not time information.

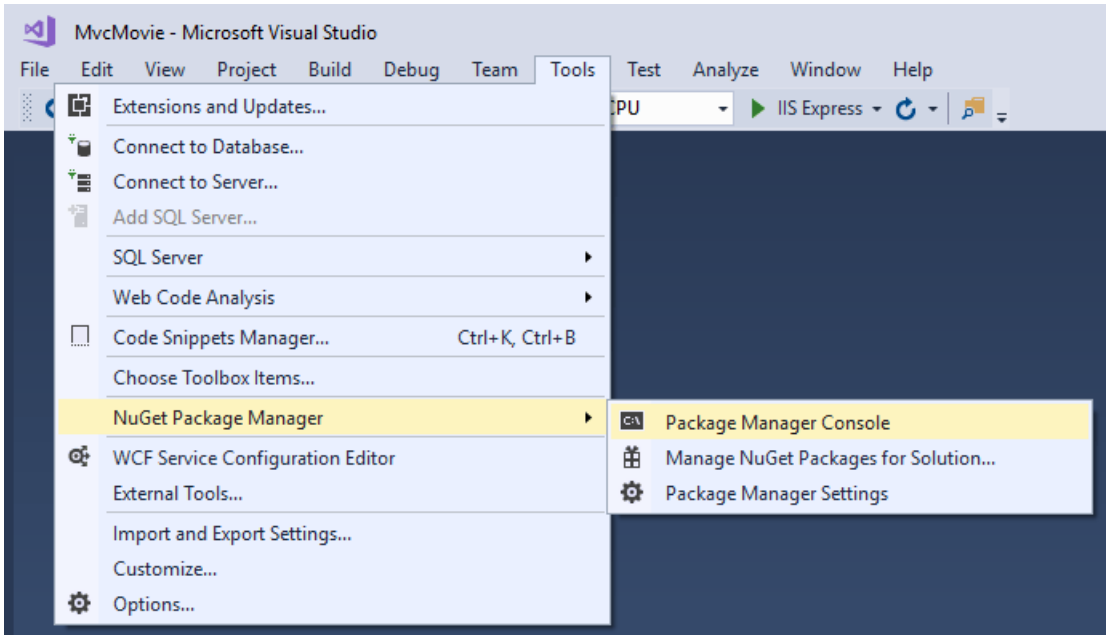
DataAnnotations are covered in a later tutorial.

Add NuGet packages

- Visual Studio

- Visual Studio Code
- Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).



In the PMC, run the following command:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

The preceding command adds the EF Core SQL Server provider. The provider package installs the EF Core package as a dependency. Additional packages are installed automatically in the scaffolding step later in the tutorial.

Create a database context class

A database context class is needed to coordinate EF Core functionality (Create, Read, Update, Delete) for the `Movie` model. The database context is derived from Microsoft.EntityFrameworkCore.DbContext and specifies the entities to include in the data model.

Create a *Data* folder.

Add a *Data/MvcMovieContext.cs* file with the following code:

```
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

Register the database context

ASP.NET Core is built with dependency injection (DI). Services (such as the EF Core DB context) must be registered with DI during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial. In this section, you register the database context with the DI container.

Add the following `using` statements at the top of *Startup.cs*:

```
using MvcMovie.Data;
using Microsoft.EntityFrameworkCore;
```

Add the following highlighted code in `Startup.ConfigureServices`:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddDbContext<MvcMovieContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Add a database connection string

Add a connection string to the *appsettings.json* file:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

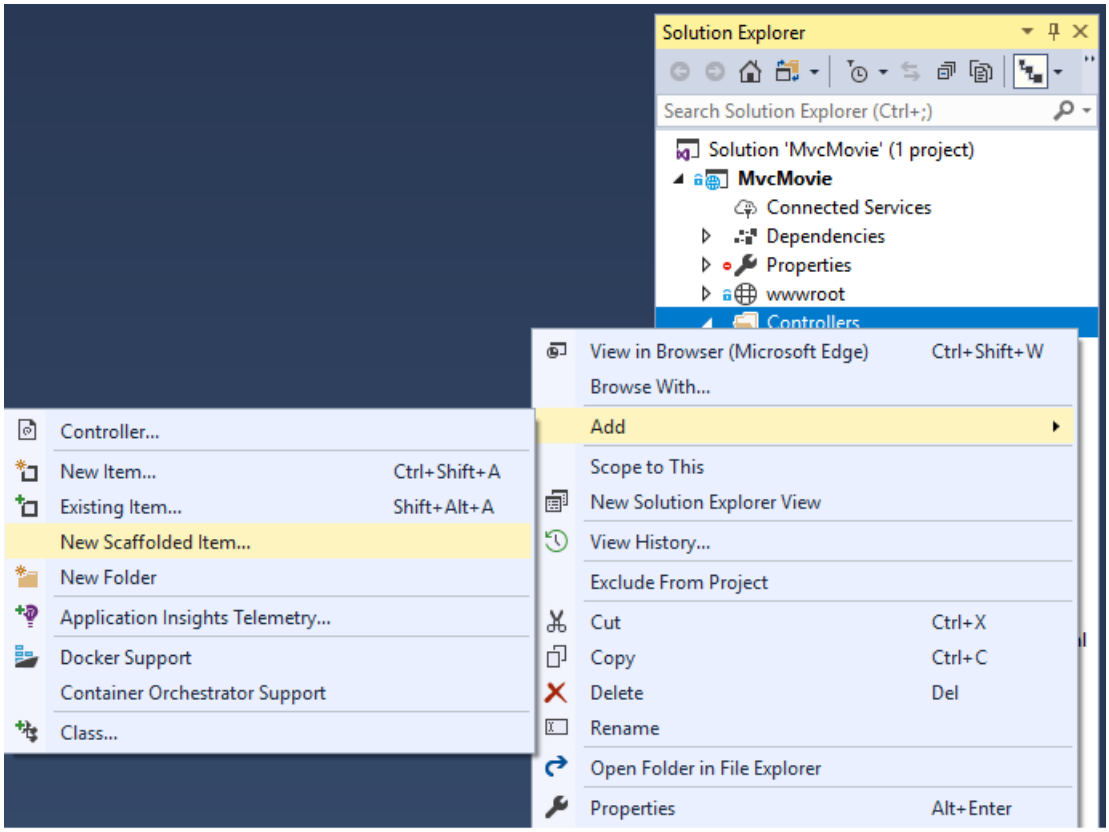
Build the project as a check for compiler errors.

Scaffold movie pages

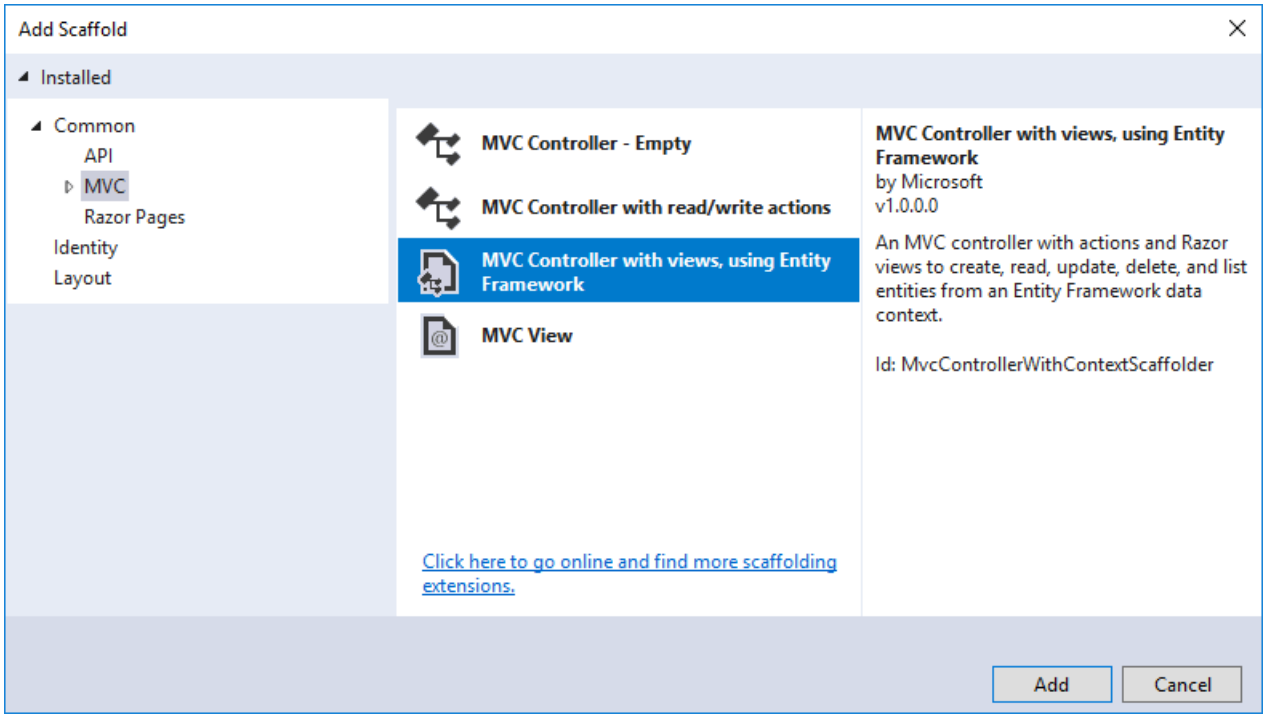
Use the scaffolding tool to produce Create, Read, Update, and Delete (CRUD) pages for the movie model.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In **Solution Explorer**, right-click the *Controllers* folder > **Add** > **New Scaffolded Item**.

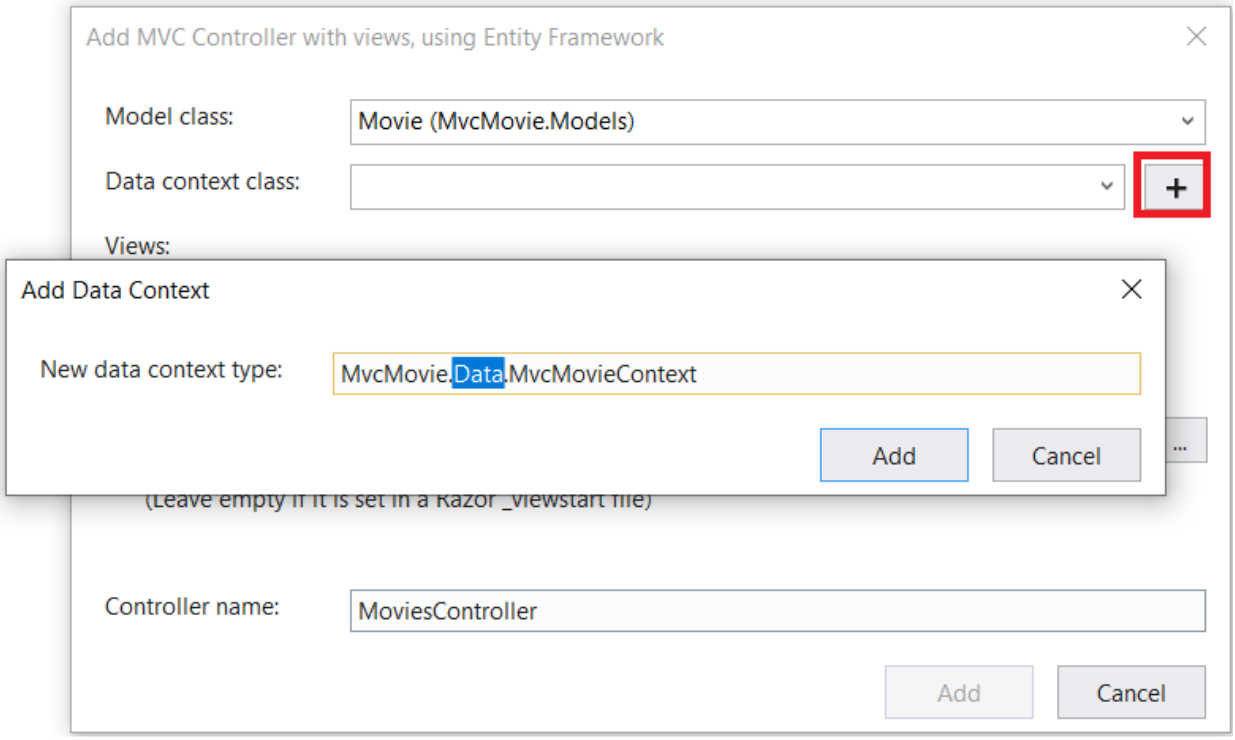


In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework** > **Add**.



Complete the **Add Controller** dialog:

- **Model class:** *Movie (MvcMovie.Models)*
- **Data context class:** *MvcMovieContext (MvcMovie.Data)*



- **Views:** Keep the default of each option checked
- **Controller name:** Keep the default *MoviesController*
- Select **Add**

Visual Studio creates:

- A movies controller (*Controllers/MoviesController.cs*)
- Razor view files for Create, Delete, Details, Edit, and Index pages (*Views/Movies/*.cshtml*)

The automatic creation of these files is known as *scaffolding*.

You can't use the scaffolded pages yet because the database doesn't exist. If you run the app and click on the **Movie App** link, you get a *Cannot open database* or *no such table: Movie* error message.

Initial migration

Use the EF Core Migrations feature to create the database. Migrations is a set of tools that let you create and update a database to match your data model.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).

In the PMC, enter the following commands:

```
Add-Migration InitialCreate
Update-Database
```

- `Add-Migration InitialCreate`: Generates a *Migrations/{timestamp}_InitialCreate.cs* migration file. The `InitialCreate` argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the `MvcMovieContext` class.

- `Update-Database`: Updates the database to the latest migration, which the previous command created. This command runs the `Up` method in the *Migrations/{time-stamp}_InitialCreate.cs* file, which creates the database.

  The database update command generates the following warning:

  > No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.

  You can ignore that warning, it will be fixed in a later tutorial.

For more information on the PMC tools for EF Core, see EF Core tools reference - PMC in Visual Studio.

The InitialCreate class

Examine the *Migrations/{timestamp}_InitialCreate.cs* migration file:

```
public partial class Initial : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Movie",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
                                SqlServerValueGenerationStrategy.IdentityColumn),
                Title = table.Column<string>(nullable: true),
                ReleaseDate = table.Column<DateTime>(nullable: false),
                Genre = table.Column<string>(nullable: true),
                Price = table.Column<decimal>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Movie", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Movie");
    }
}
```

The `Up` method creates the Movie table and configures `Id` as the primary key. The `Down` method reverts the schema changes made by the `Up` migration.

Test the app

- Run the app and click the **Movie App** link.

  If you get an exception similar to one of the following:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
SqlException: Cannot open database "MvcMovieContext-1" requested by the login. The login failed.
```

You probably missed the migrations step.

- Test the **Create** page. Enter and submit data.

> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** pages.

Dependency injection in the controller

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Open the *Controllers/MoviesController.cs* file and examine the constructor:

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the `MoviesController` class and views.

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller (the first URL segment).
- The action to `details` (the second URL segment).
- The id to 1 (the last URL segment).

You can also pass in the `id` with a query string as follows:

`https://localhost:5001/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in case an ID value isn't provided.

A lambda expression is passed in to `FirstOrDefaultAsync` to select movie entities that match the route data or query string value.

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the *Views/Movies/Details.cshtml* file:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

The `@model` statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following `@model` statement was included:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows access to the movie that the controller passed to the view. The `Model` object is strongly typed. For example, in the *Details.cshtml* view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the *Index.cshtml* view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

When the movies controller was created, scaffolding included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each item in the loop is typed as `Movie`. Among other benefits, this means that you get compile time checking of the code.

Additional resources

- Tag Helpers
- Globalization and localization

PREVIOUS ADDING A VIEW

NEXT WORKING WITH SQL

Add a data model class

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Right-click the *Models* folder > **Add** > **Class**. Name the class **Movie**.

Add the following properties to the `Movie` class:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains:

- The `Id` field which is required by the database for the primary key.
- `[DataType(DataType.Date)]`: The DataType attribute specifies the type of the data (`Date`). With this attribute:
  - The user is not required to enter time information in the date field.
  - Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

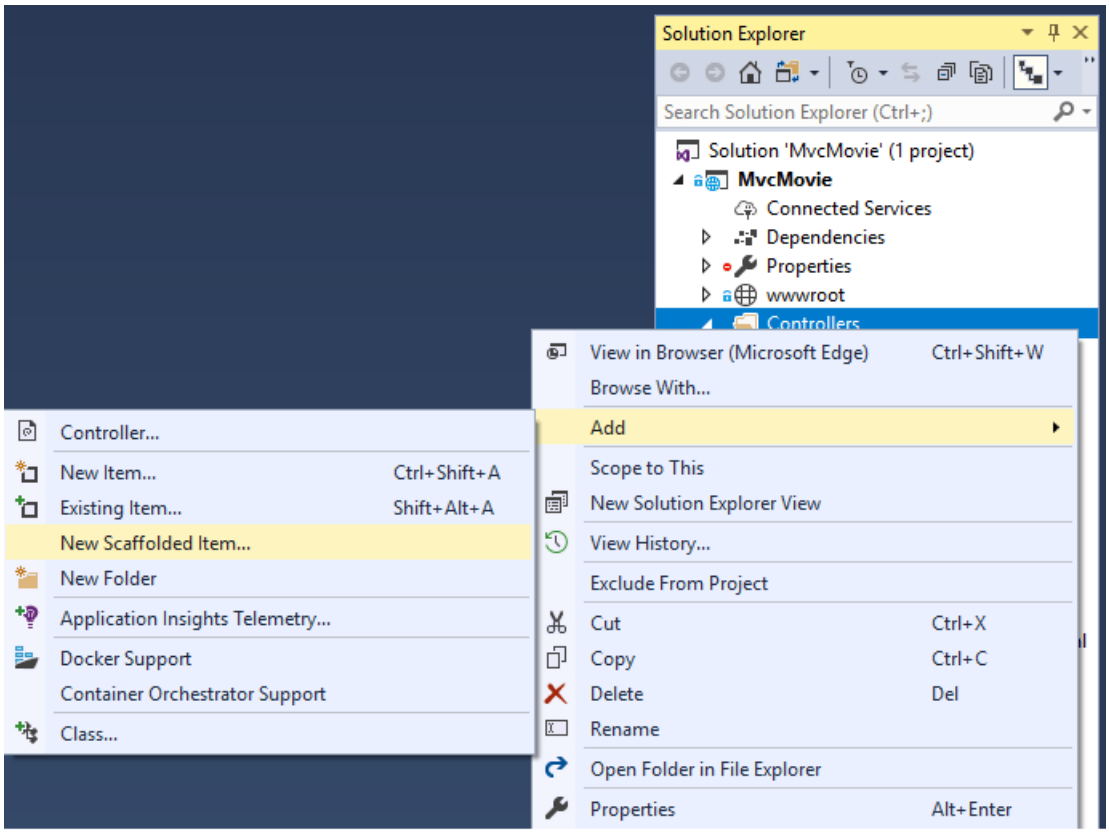
Scaffold the movie model

In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

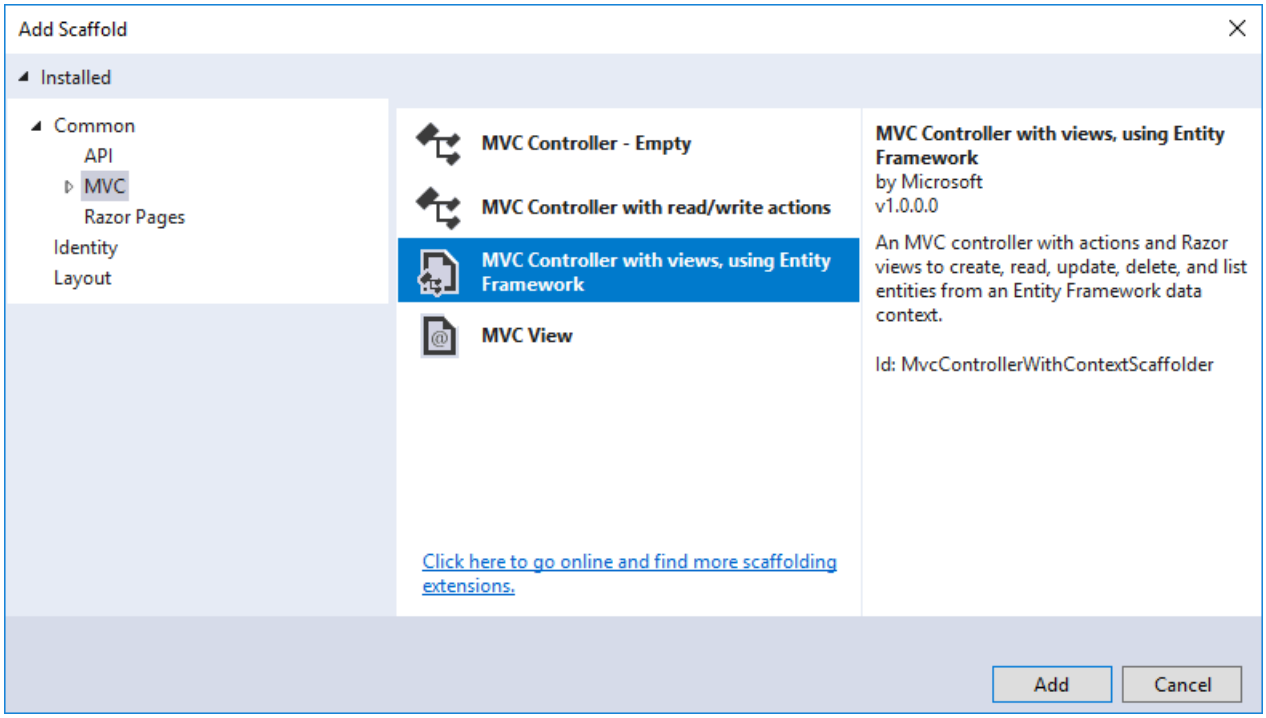
- Visual Studio
- Visual Studio Code

- Visual Studio for Mac

In **Solution Explorer**, right-click the *Controllers* folder > **Add** > **New Scaffolded Item**.

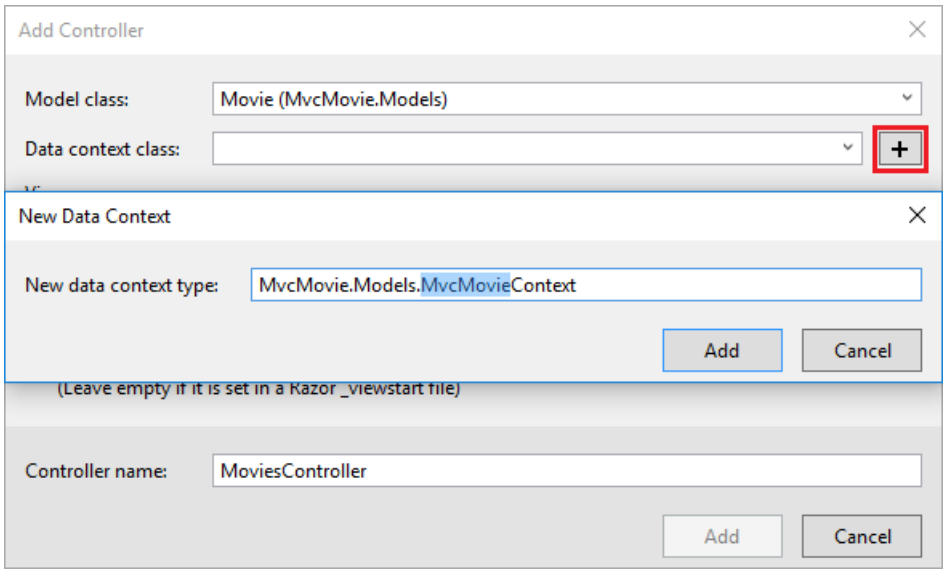


In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework** > **Add**.

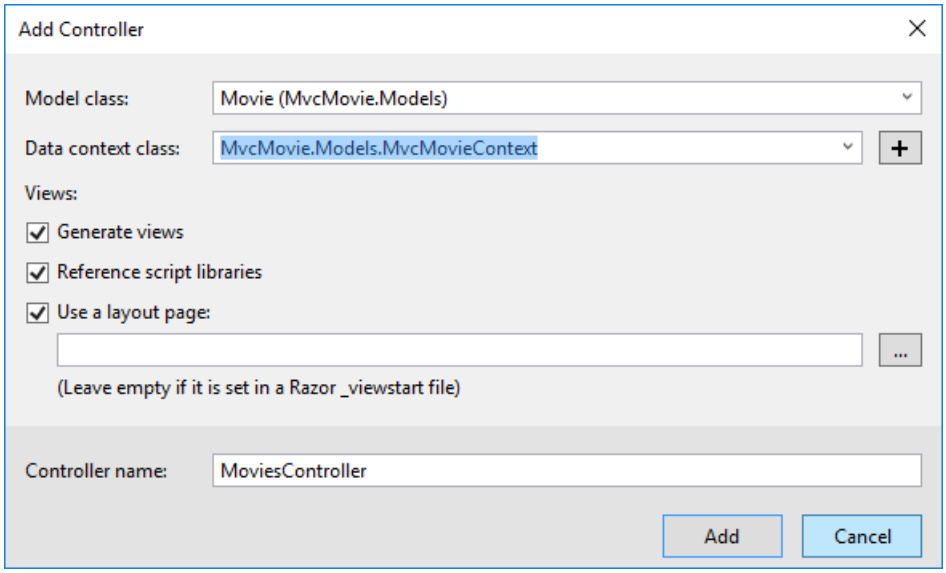


Complete the **Add Controller** dialog:

- **Model class:** *Movie (MvcMovie.Models)*
- **Data context class:** Select the **+** icon and add the default **MvcMovie.Models.MvcMovieContext**



- **Views**: Keep the default of each option checked
- **Controller name**: Keep the default *MoviesController*
- Select **Add**



Visual Studio creates:

- An Entity Framework Core database context class (*Data/MvcMovieContext.cs*)
- A movies controller (*Controllers/MoviesController.cs*)
- Razor view files for Create, Delete, Details, Edit, and Index pages (*Views/Movies/*.cshtml*)

The automatic creation of the database context and CRUD (create, read, update, and delete) action methods and views is known as *scaffolding*.

If you run the app and click on the **Mvc Movie** link, you get an error similar to the following:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
An unhandled exception occurred while processing the request.

SqlException: Cannot open database "MvcMovieContext-<GUID removed>" requested by the login. The login failed.
Login failed for user 'Rick'.

System.Data.SqlClient.SqlInternalConnectionTds..ctor(DbConnectionPoolIdentity identity, SqlConnectionString
```

You need to create the database, and you use the EF Core Migrations feature to do that. Migrations lets you create a database that matches your data model and update the database schema when your data model changes.

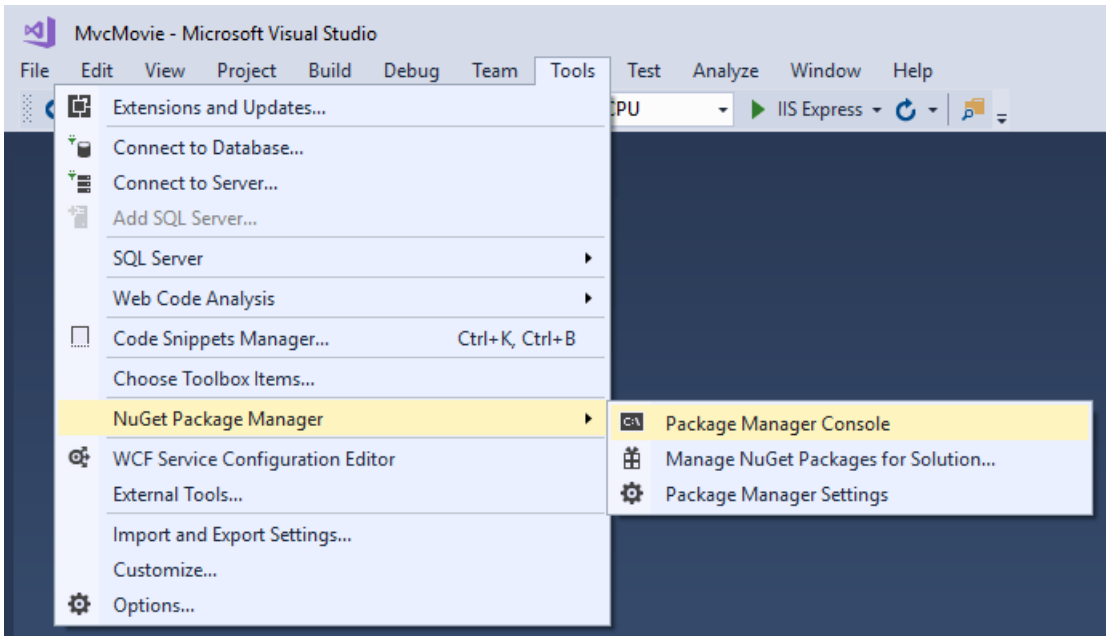
Initial migration

In this section, the following tasks are completed:

- Add an initial migration.
- Update the database with the initial migration.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

1. From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).



2. In the PMC, enter the following commands:

```
Add-Migration Initial
Update-Database
```

   The `Add-Migration` command generates code to create the initial database schema.

   The database schema is based on the model specified in the `MvcMovieContext` class. The `Initial` argument is the migration name. Any name can be used, but by convention, a name that describes the migration is used. For more information, see Tutorial: Using the migrations feature - ASP.NET MVC with EF Core.

   The `Update-Database` command runs the `Up` method in the *Migrations/{time-stamp}_InitialCreate.cs* file, which creates the database.

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection (DI). Services (such as the EF Core DB context) are registered with DI during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

The scaffolding tool automatically created a DB context and registered it with the DI container.

Examine the following `Startup.ConfigureServices` method. The highlighted line was added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The `MvcMovieContext` coordinates EF Core functionality (Create, Read, Update, Delete, etc.) for the `Movie` model. The data context (`MvcMovieContext`) is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model:

```
// Unused usings removed.
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;  // Enables public DbSet<Movie> Movie

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Test the app

- Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

If you get a database exception similar to the following:

```
SqlException: Cannot open database "MvcMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

- Test the **Create** link. Enter and submit data.

> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** links.

Examine the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The preceding highlighted code shows the movie database context being added to the Dependency Injection container:

- `services.AddDbContext<MvcMovieContext>(options =>` specifies the database to use and the connection string.
- `=>` is a lambda operator

Open the *Controllers/MoviesController.cs* file and examine the constructor:

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables better compile time checking of your code. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the `MoviesController` class and views when it created the methods and views.

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller (the first URL segment).
- The action to `details` (the second URL segment).
- The id to 1 (the last URL segment).

You can also pass in the `id` with a query string as follows:

`https://localhost:5001/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in case an ID value isn't provided.

A lambda expression is passed in to `FirstOrDefaultAsync` to select movie entities that match the route data or query string value.

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the *Views/Movies/Details.cshtml* file:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

By including a `@model` statement at the top of the view file, you can specify the type of object that the view expects. When you created the movie controller, the following `@model` statement was automatically included at the top of the *Details.cshtml* file:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows you to access the movie that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Details.cshtml* view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the *Index.cshtml* view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

When you created the movies controller, scaffolding automatically included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each item in the loop is typed as `Movie`. Among other benefits, this means that you get compile time checking of the code:

Additional resources

- Tag Helpers
- Globalization and localization

PREVIOUS ADDING A VIEW

NEXT WORKING WITH A DATABASE

Part 5, work with a database in an ASP.NET Core MVC app

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in the *Startup.cs* file:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddDbContext<MvcMovieContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file:

```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

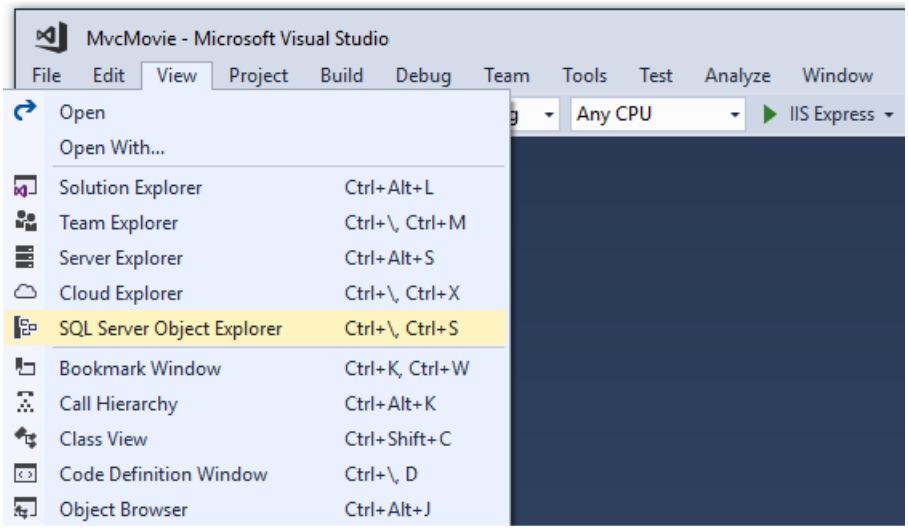
When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a production SQL Server. See Configuration for more information.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

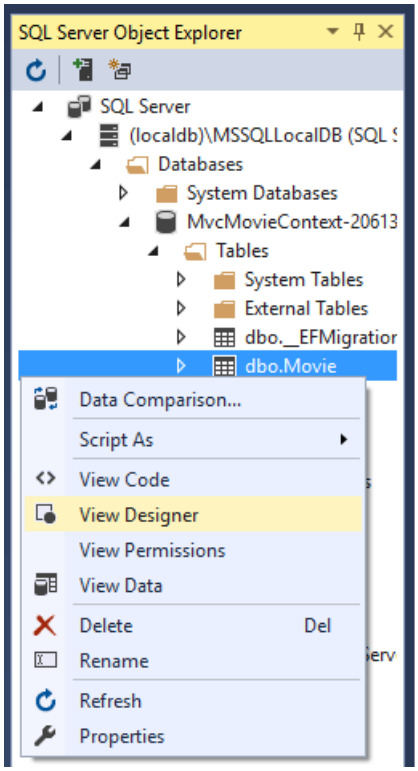
SQL Server Express LocalDB

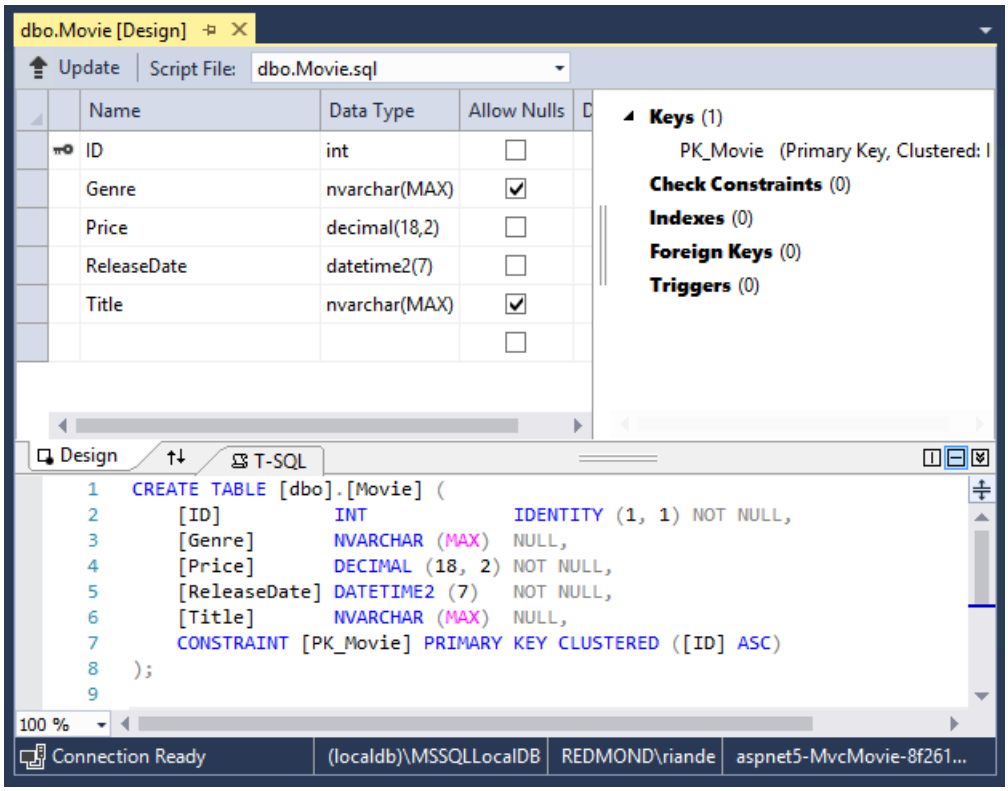
LocalDB is a lightweight version of the SQL Server Express Database Engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates *.mdf* files in the *C:/Users/{user}* directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



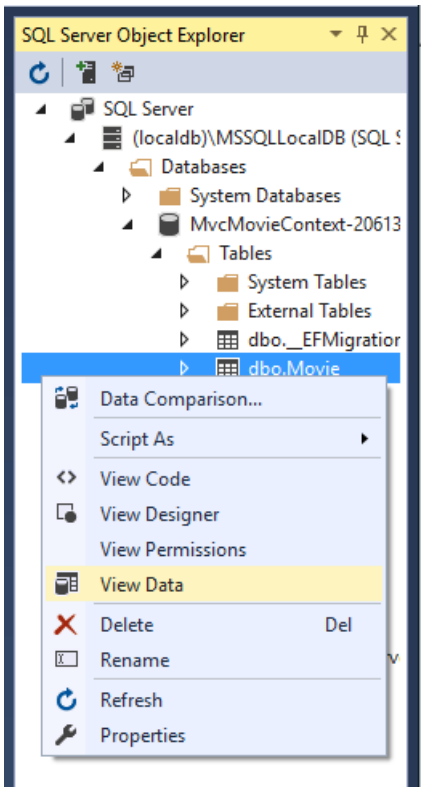
- Right click on the `Movie` table > **View Designer**

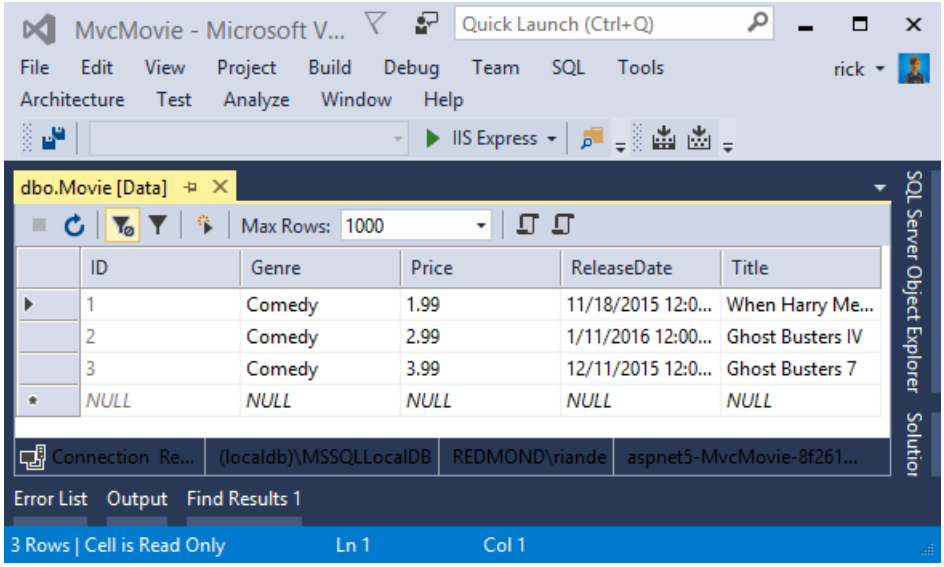




Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.

- Right click on the `Movie` table > **View Data**





Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

Replace the contents of *Program.cs* with the following code:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using MvcMovie.Data;
using MvcMovie.Models;
using System;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();

        }

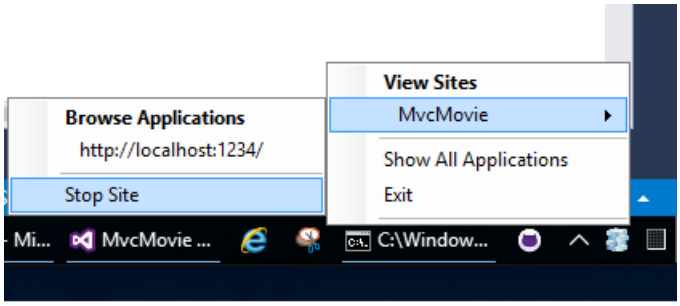
        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Test the app

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

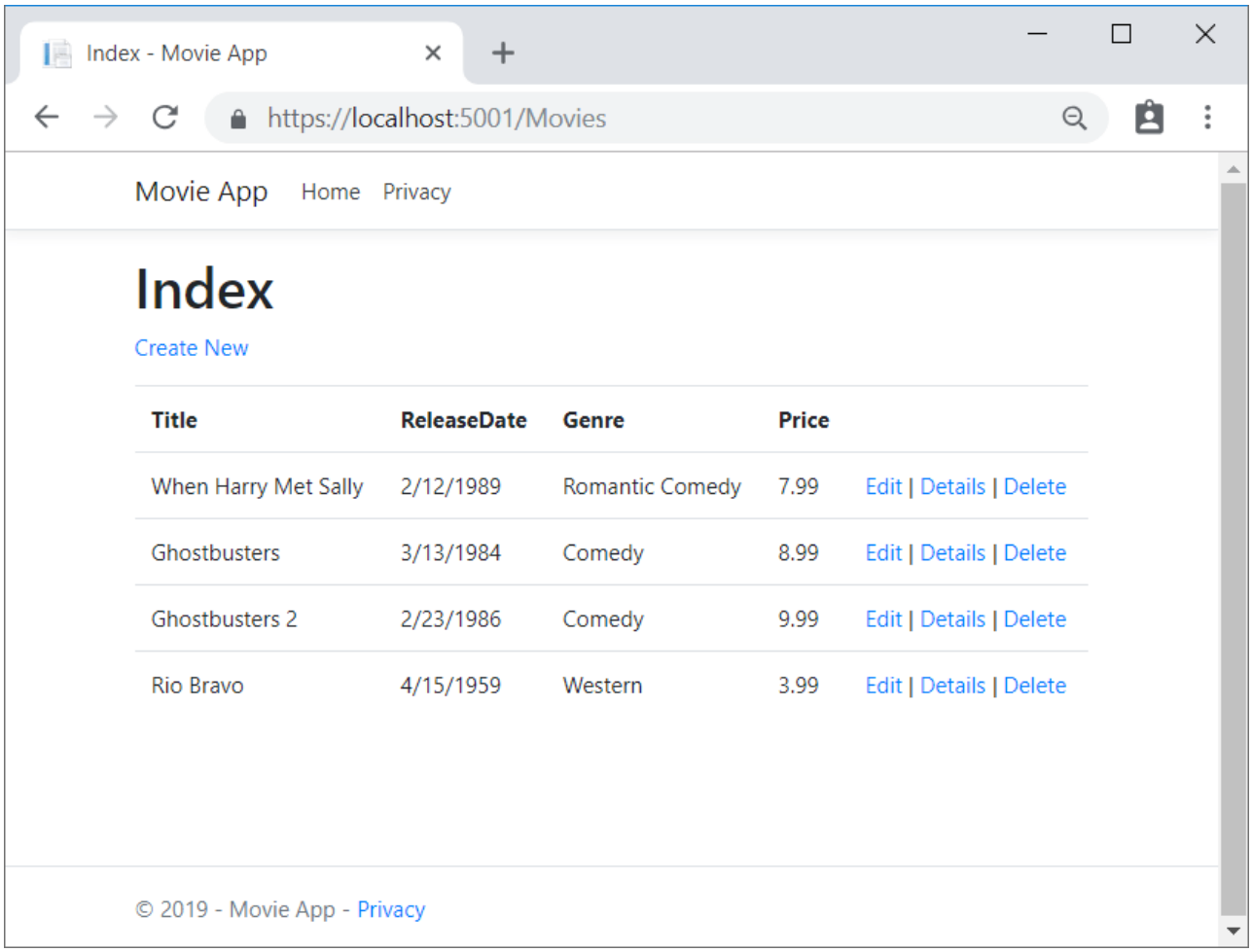
- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:

- Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**



  - If you were running VS in non-debug mode, press F5 to run in debug mode
  - If you were running VS in debug mode, stop the debugger and press F5

The app shows the seeded data.



PREVIOUS NEXT

By Rick Anderson

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to

database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in the *Startup.cs* file:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file:

```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

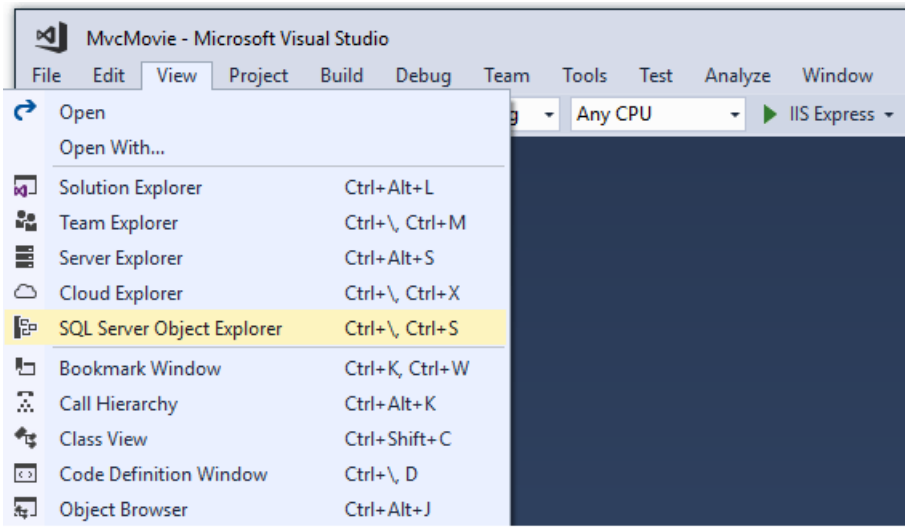
When you deploy the app to a test or production server, you can use an environment variable or another approach to set the connection string to a real SQL Server. See Configuration for more information.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

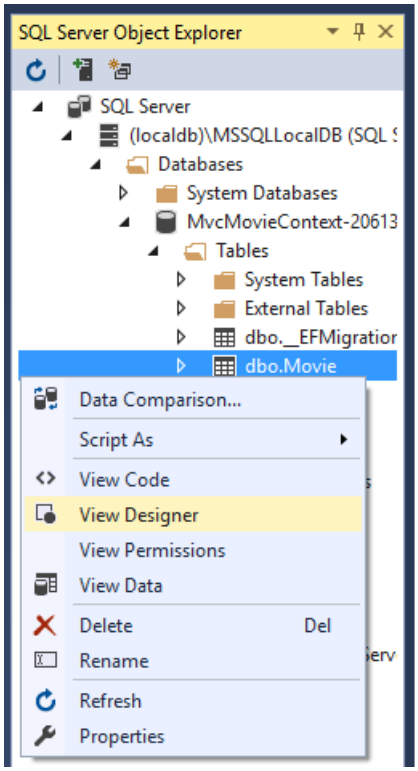
SQL Server Express LocalDB

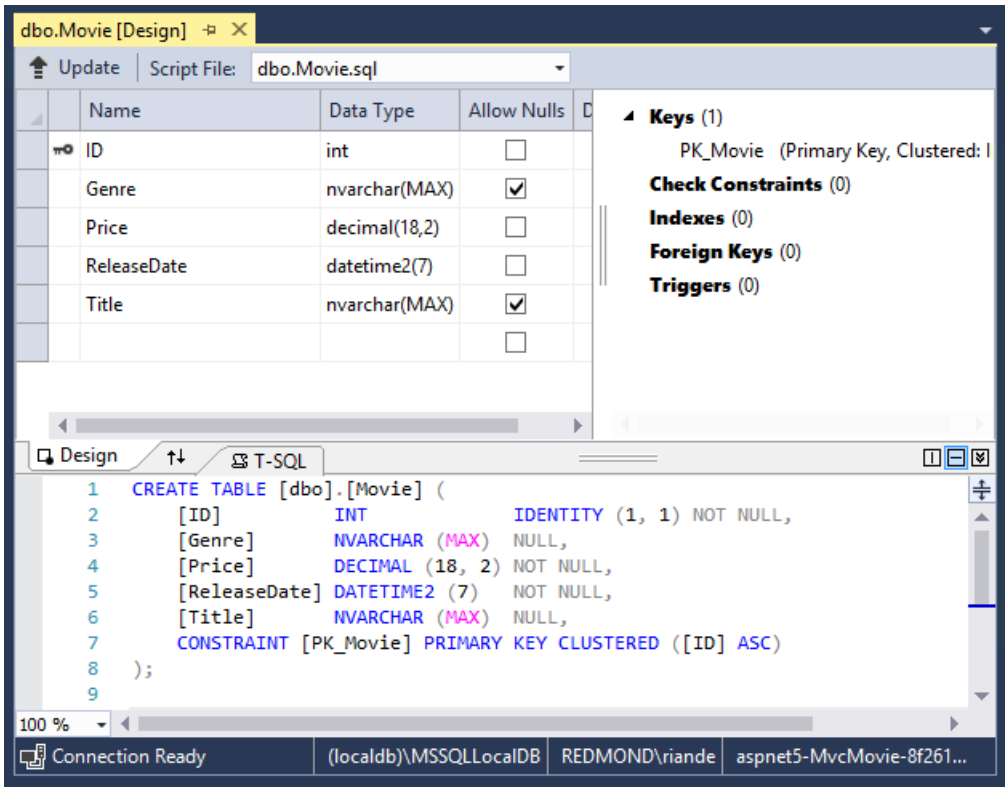
LocalDB is a lightweight version of the SQL Server Express Database Engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates *.mdf* files in the *C:/Users/{user}* directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



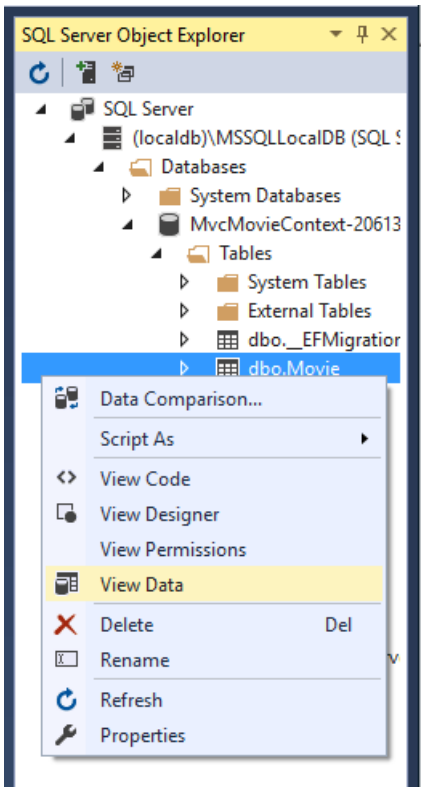
- Right click on the `Movie` table > **View Designer**

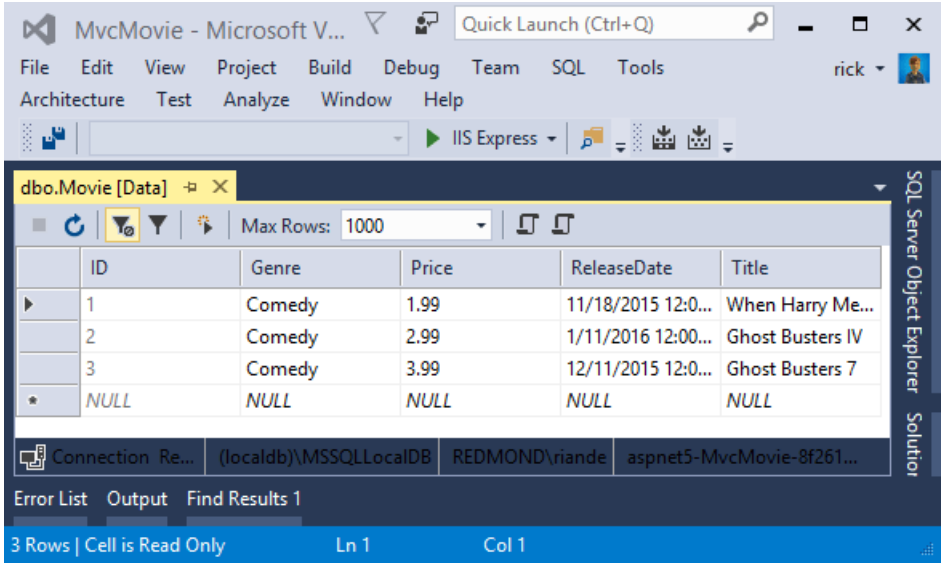




Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.

- Right click on the `Movie` table > **View Data**





Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

Replace the contents of *Program.cs* with the following code:

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System;
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;
using MvcMovie;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<MvcMovieContext>();
                    context.Database.Migrate();
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

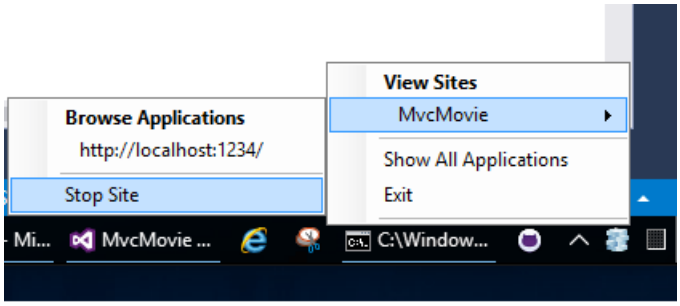
        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

Test the app

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

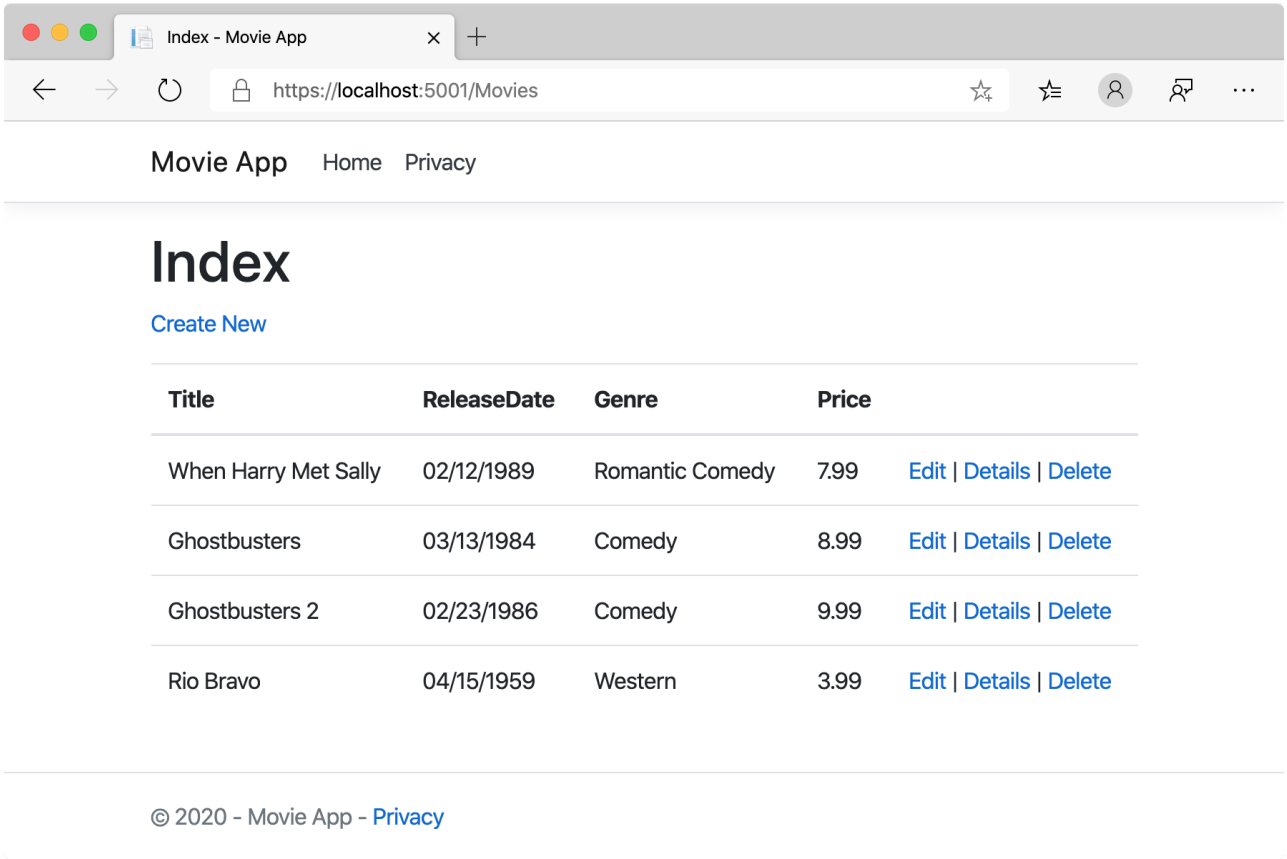
- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:

- Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**



  - If you were running VS in non-debug mode, press F5 to run in debug mode
  - If you were running VS in debug mode, stop the debugger and press F5

The app shows the seeded data.

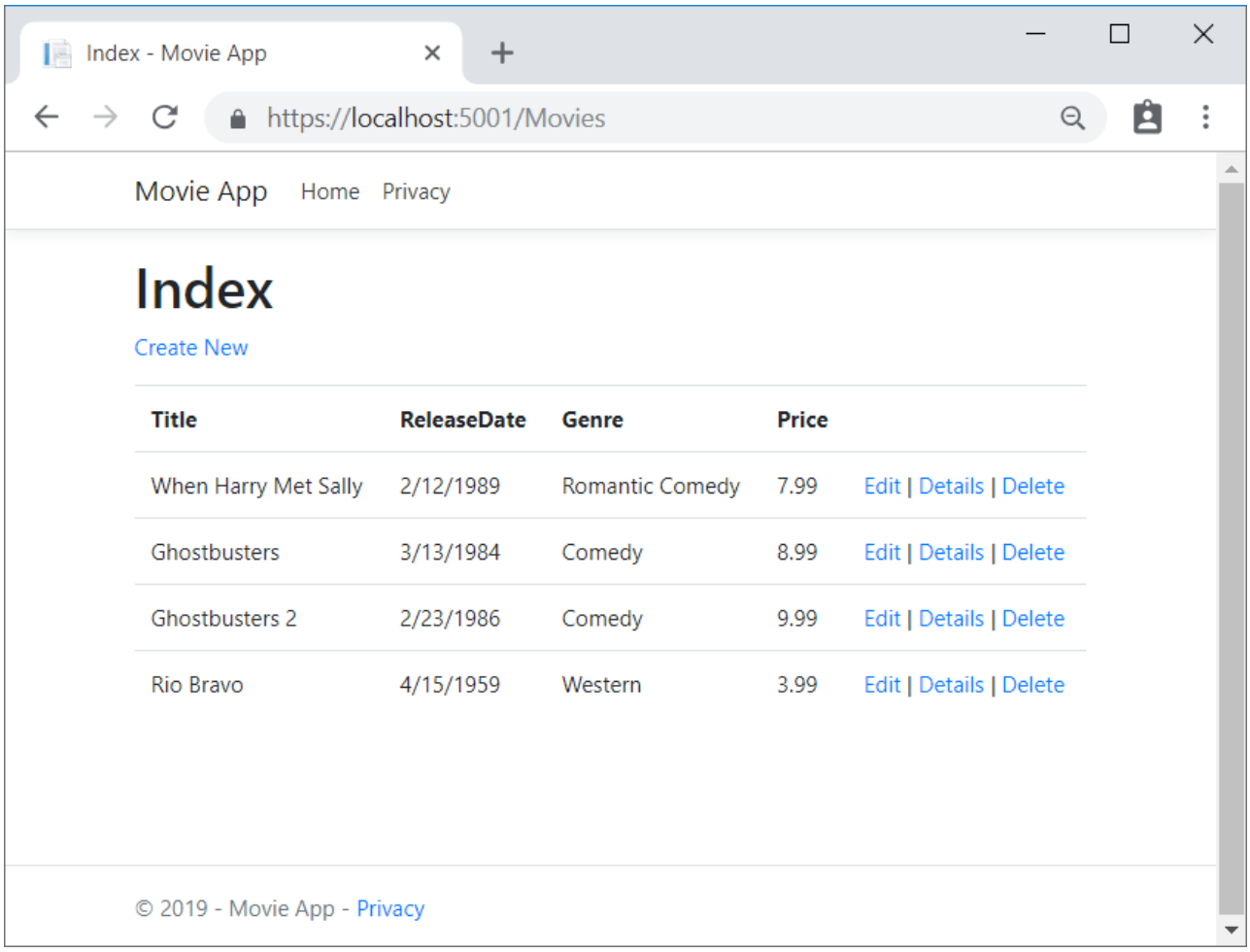


Part 6, controller methods and views in ASP.NET Core

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

We have a good start to the movie app, but the presentation isn't ideal, for example, **ReleaseDate** should be two words.



Open the *Models/Movie.cs* file and add the highlighted lines shown below:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

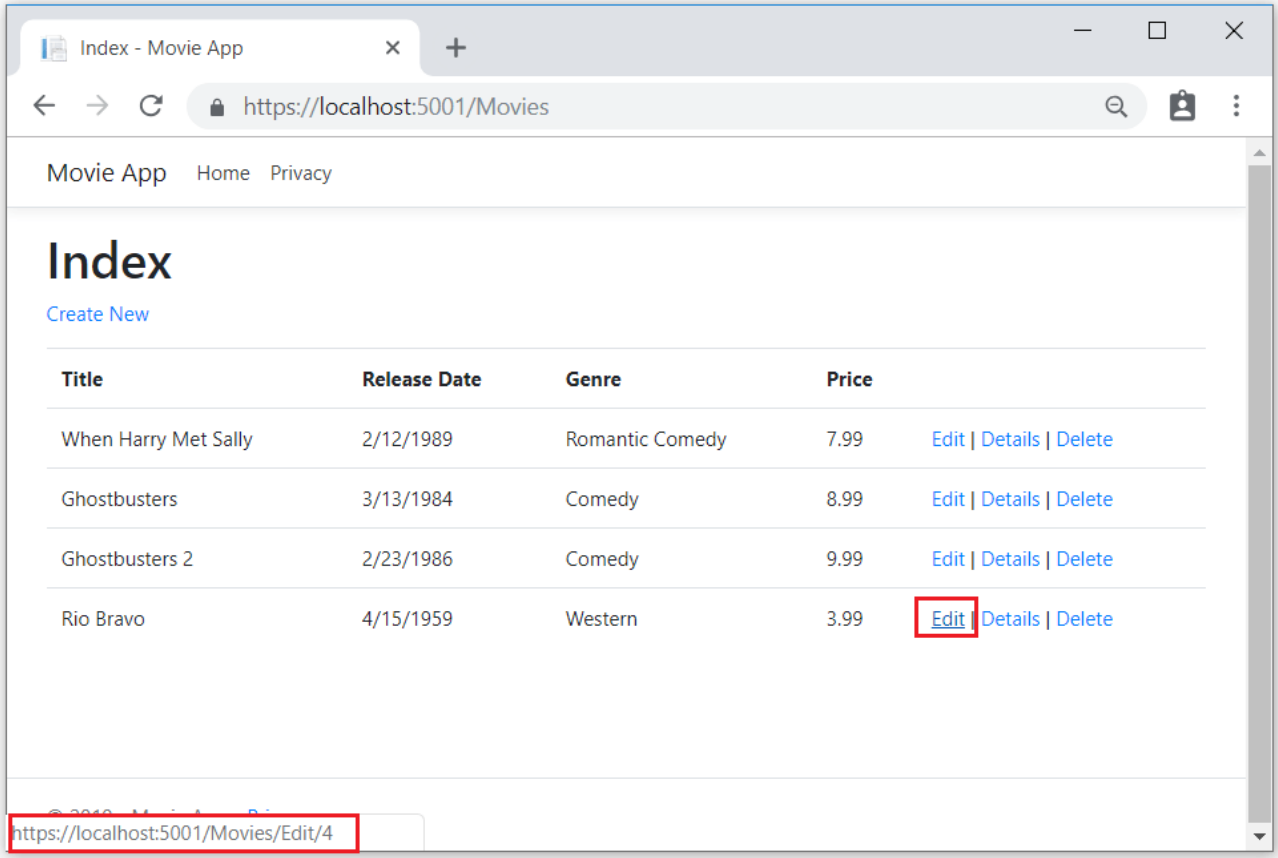
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

We cover DataAnnotations in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Core MVC Anchor Tag Helper in the *Views/Movies/Index.cshtml* file.

```
        <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
        <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
        <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
    </td>
</tr>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

```
 <td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for routing set in the *Startup.cs* file:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

ASP.NET Core translates `https://localhost:5001/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. For more information, see Additional resources.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the *Edit.cshtml* Razor file.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `[Bind]` attribute is one way to protect against over-posting. You should only include properties in the `[Bind]` attribute that you want to change. For more information, see Protect your controller from over-posting. ViewModels provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to prevent forgery of a request and is paired up with an anti-forgery token generated in the edit view file (*Views/Movies/Edit.cshtml*). The edit view file generates the anti-forgery token with the Form Tag Helper.

```
<form asp-action="Edit">
```

The Form Tag Helper generates a hidden anti-forgery token that must match the `[ValidateAntiForgeryToken]` generated anti-forgery token in the `Edit` method of the Movies controller. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `FindAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file. `@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The Label Tag Helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML `<input>` element. The Validation Tag Helper displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID"
value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-
replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true" data-val-number="The field Price must
be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-
replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The `<input>` elements are in an `HTML <form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the `Save` button is clicked. The last line before the closing `</form>` element shows the hidden XSRF token generated by the Form Tag Helper.

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

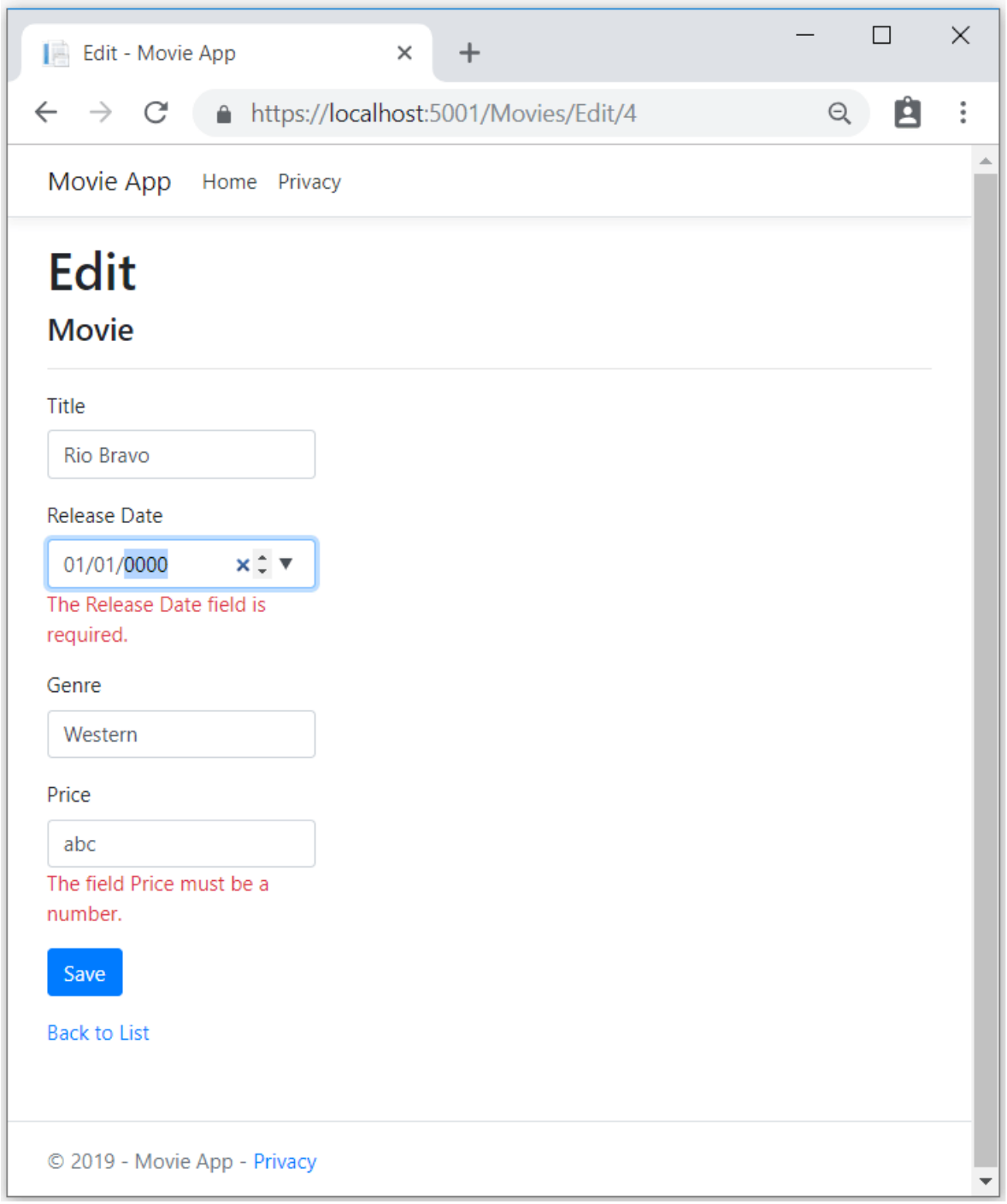
```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the anti-forgery token generator in the Form Tag Helper

The model binding system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` method verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine Model Validation in more detail. The Validation Tag Helper in the *Views/Movies/Edit.cshtml* view template takes care of displaying appropriate error messages.



All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests shouldn't change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers
- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core
- Protect your controller from over-posting

- ViewModels
- Form Tag Helper
- Input Tag Helper
- Label Tag Helper
- Select Tag Helper
- Validation Tag Helper

PREVIOUS NEXT

Part 7, add search to an ASP.NET Core MVC app

9/22/2020 • 7 minutes to read • Edit Online

By Rick Anderson

In this section, you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method found inside *Controllers/MoviesController.cs* with the following code:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

The first line of the `Index` action method creates a LINQ query to select the movies:

```
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

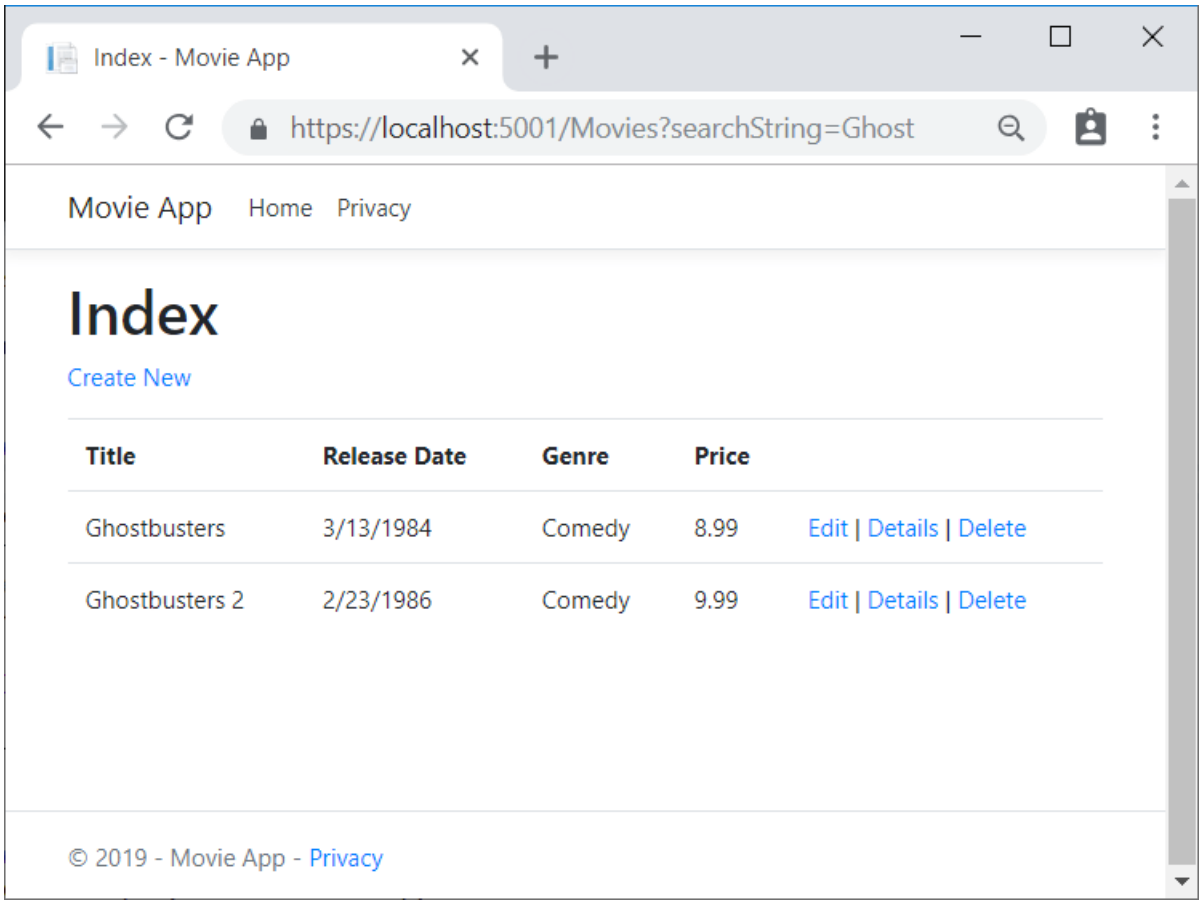
If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

The `s => s.Title.Contains()` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see Query Execution.

Note: The Contains method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, Contains maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in *Startup.cs*.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Change the parameter to `id` and all occurrences of `searchString` change to `id`.

The previous `Index` method:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

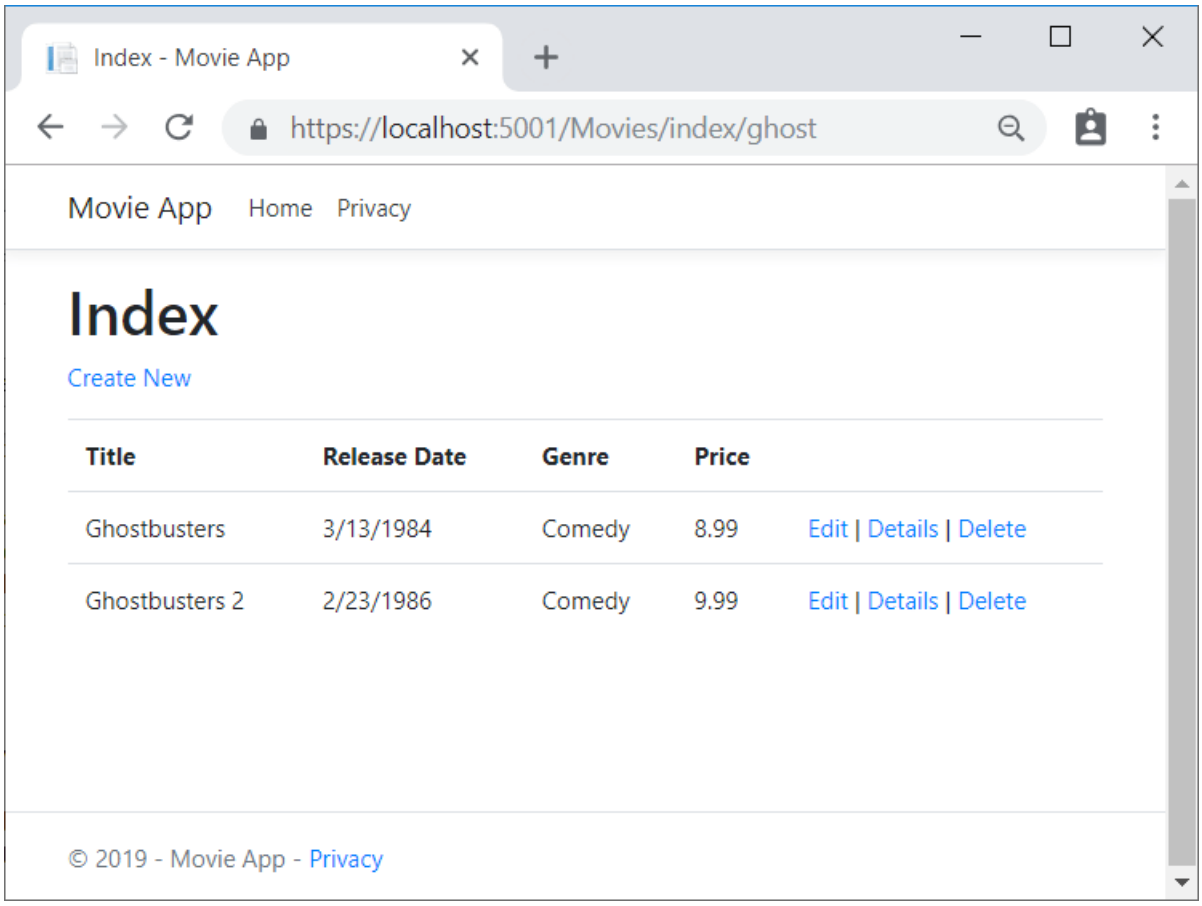
The updated `Index` method with `id` parameter:

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

Open the *Views/Movies/Index.cshtml* file, and add the `<form>` markup highlighted below:

```
    ViewData["Title"] = "Index";
}

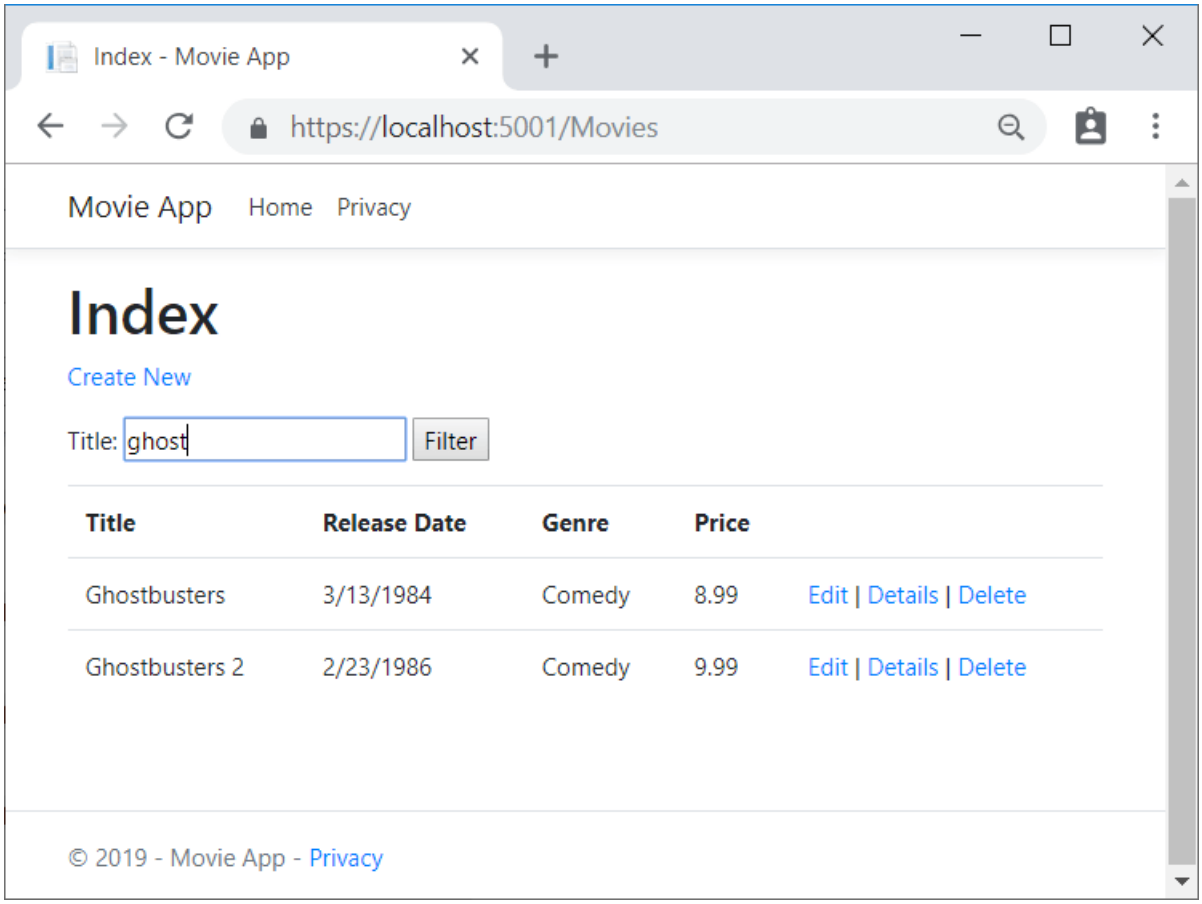
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

The HTML `<form>` tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.



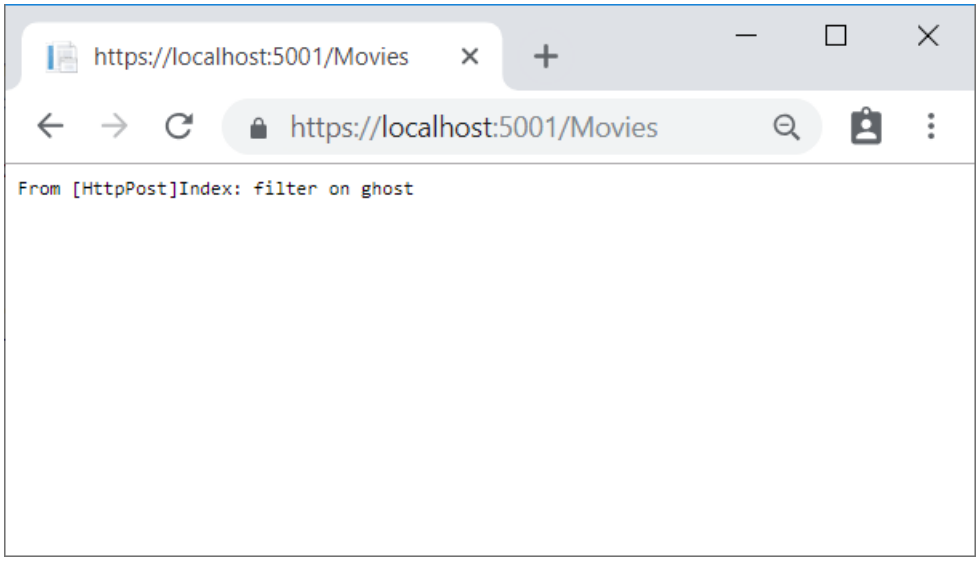
There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following `[HttpPost] Index` method.

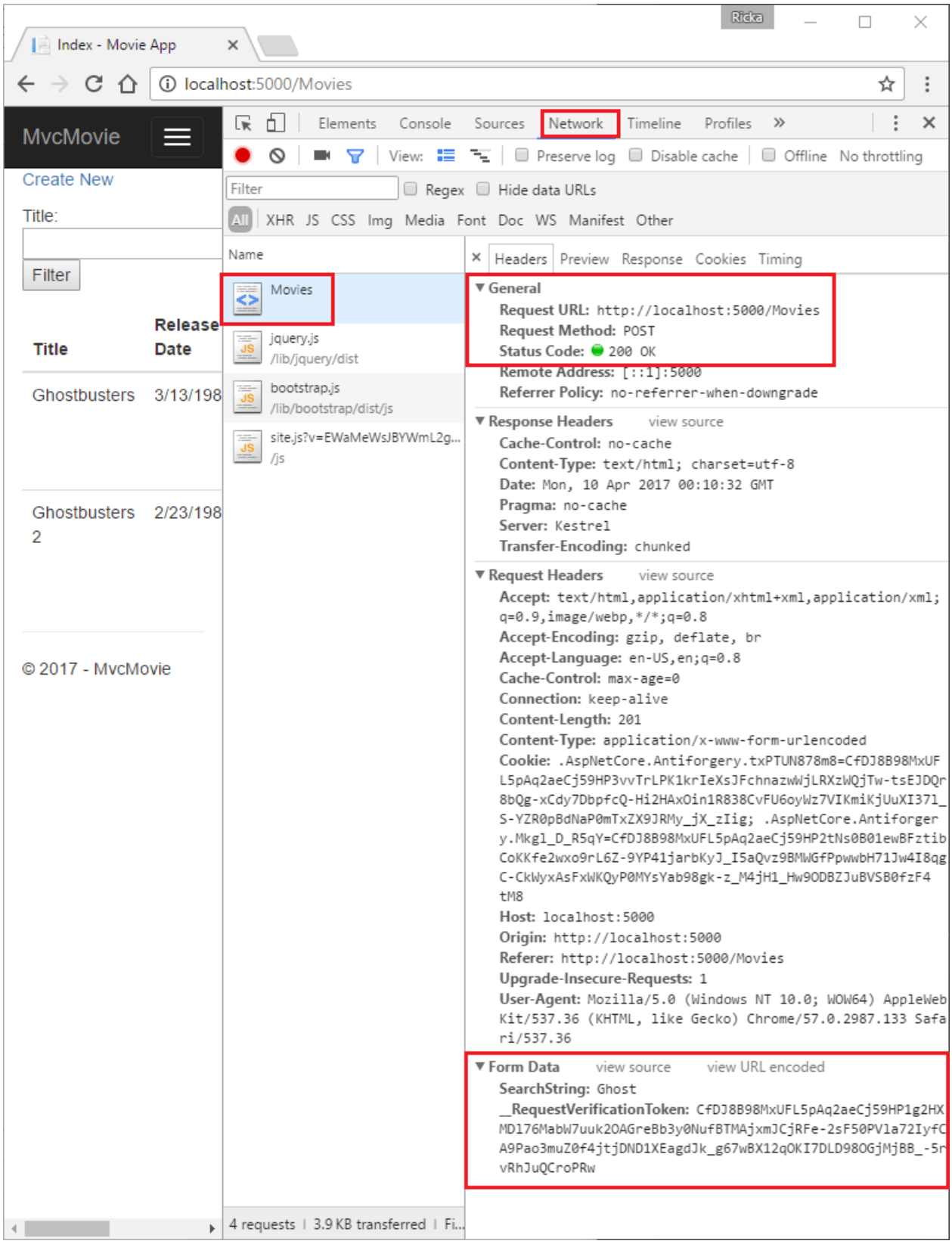
```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:{PORT}/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool. The image below shows the Chrome browser Developer tools:



You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be `HTTP GET` found in the *Views/Movies/Index.cshtml* file.

```
@model IEnumerable<MvcMovie.Models.Movie>

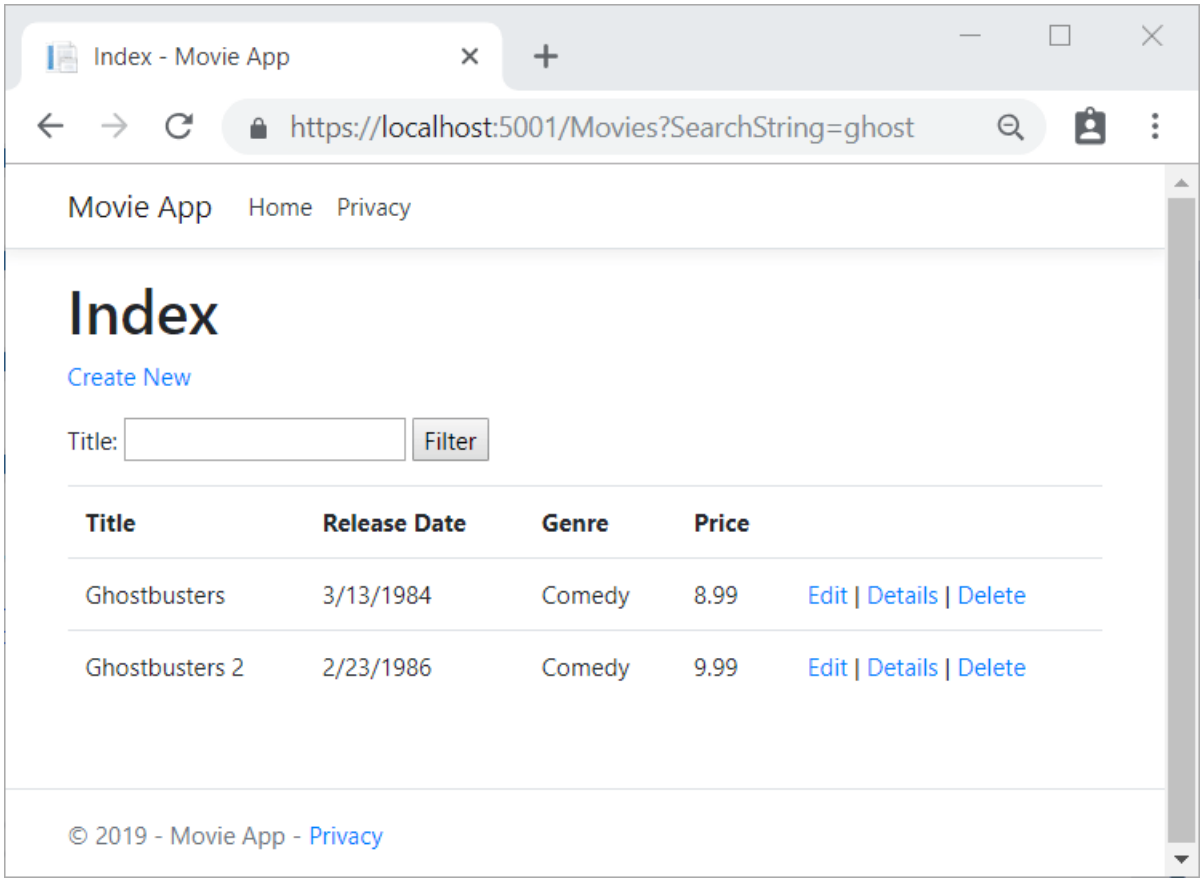
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



The following markup shows the change to the `form` tag:

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Add Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> Movies { get; set; }
        public SelectList Genres { get; set; }
        public string MovieGenre { get; set; }
        public string SearchString { get; set; }
    }
}
```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This allows the user to select a genre from the list.
- `MovieGenre`, which contains the selected genre.
- `SearchString`, which contains the text users enter in the search text box.

Replace the `Index` method in `MoviesController.cs` with the following code:

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

The following code is a `LINQ` query that retrieves all the genres from the database.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

When the user searches for the item, the search value is retained in the search box.

Add search by genre to the Index view

Update `Index.cshtml` found in *Views/Movies/* as follows:

```
@model MvcMovie.Models.MovieGenreViewModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

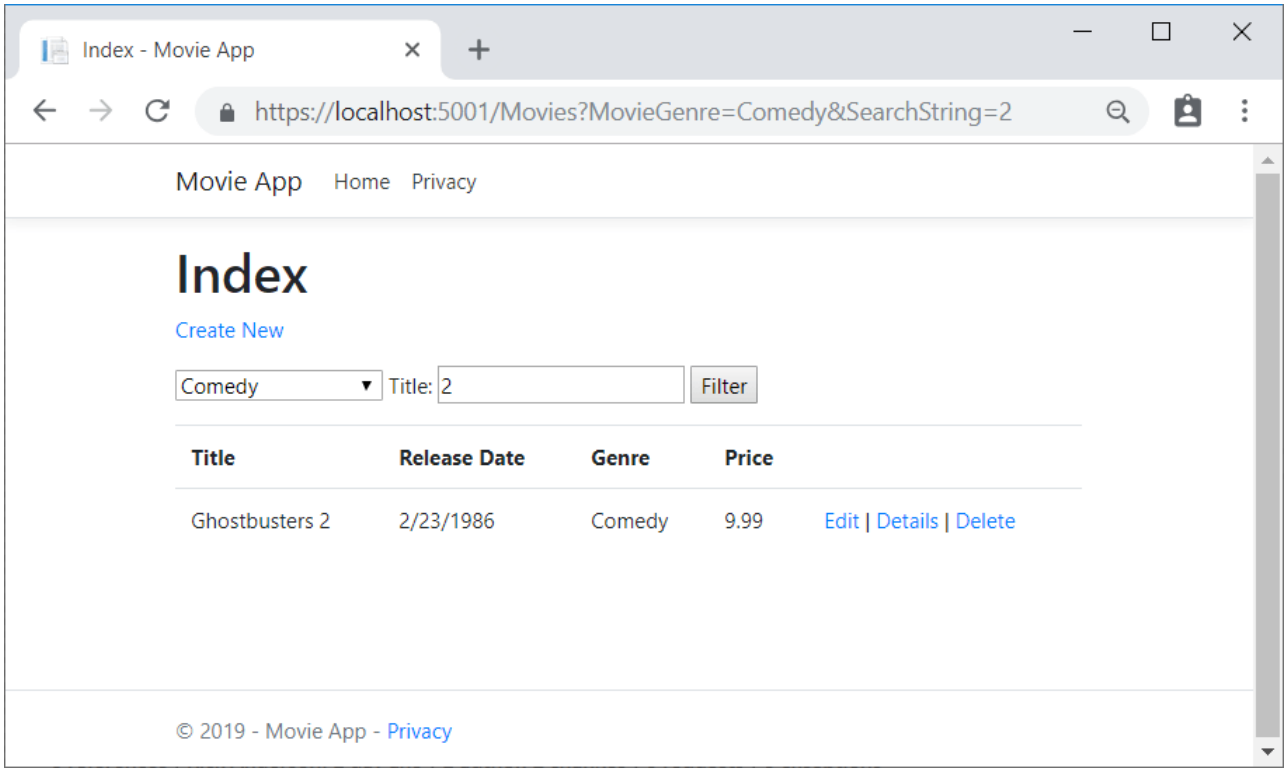
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movies[0].Title)
```

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.Movies`, or `model.Movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

Test the app by searching by genre, by movie title, and by both:



Part 8, add a new field to an ASP.NET Core MVC app

9/22/2020 • 4 minutes to read • Edit Online

By Rick Anderson

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When EF Code First is used to automatically create a database, Code First:

- Adds a table to the database to track the schema of the database.
- Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Add a Rating Property to the Movie Model

Add a `Rating` property to *Models/Movie.cs*:

```
public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Ctrl+Shift+B

Because you've added a new field to the `Movie` class, you need to update the binding white list so this new property will be included. In *MoviesController.cs*, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

Update the view templates in order to display, create, and edit the new `Rating` property in the browser view.

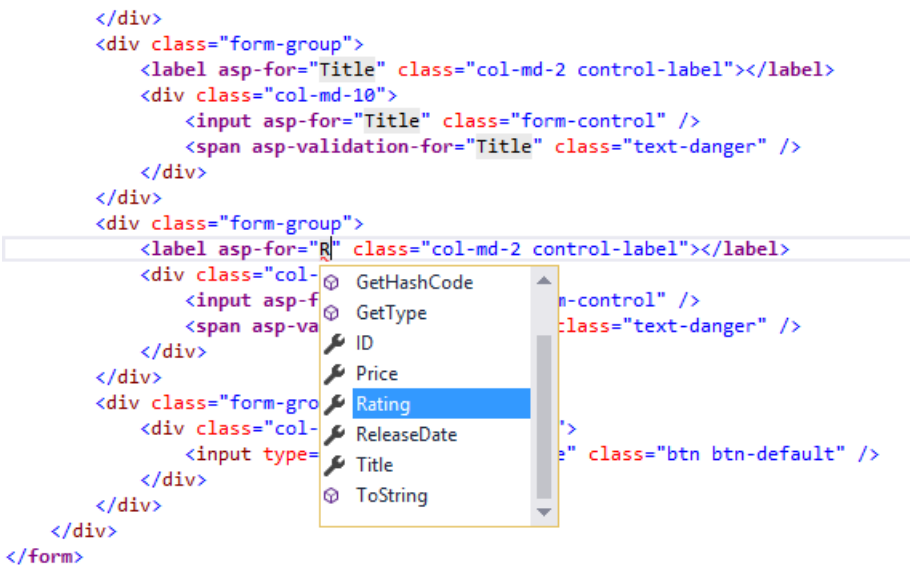
Edit the */Views/Movies/Index.cshtml* file and add a `Rating` field:

```
<thead>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Rating)
        </th>
        <th></th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model.Movies)
    {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
```

Update the */Views/Movies/Create.cshtml* with a `Rating` field.

- Visual Studio / Visual Studio for Mac
- Visual Studio Code

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers.



Update the remaining templates.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

The app won't work until the DB is updated to include the new field. If it's run now, the following `SqlException` is thrown:

```
SqlException: Invalid column name 'Rating'.
```

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.

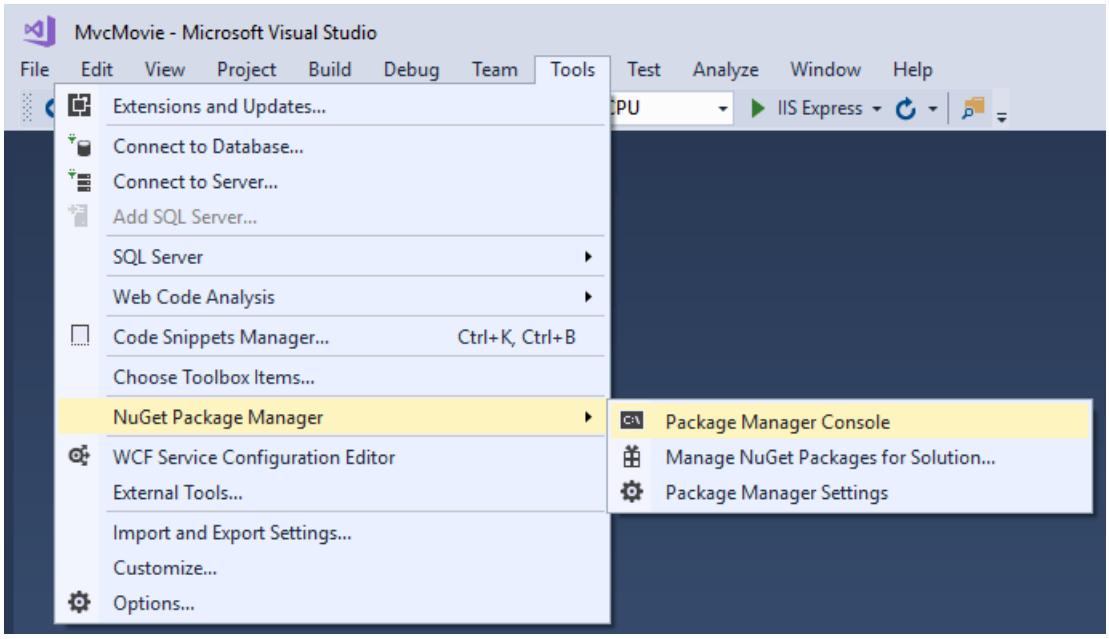
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

3. Use Code First Migrations to update the database schema.

For this tutorial, Code First Migrations is used.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the `Rating` field.

Run the app and verify you can create, edit, and display movies with a `Rating` field.

PREVIOUS NEXT

Part 9, add validation to an ASP.NET Core MVC app

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

In this section:

- Validation logic is added to the `Movie` model.
- You ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```
public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

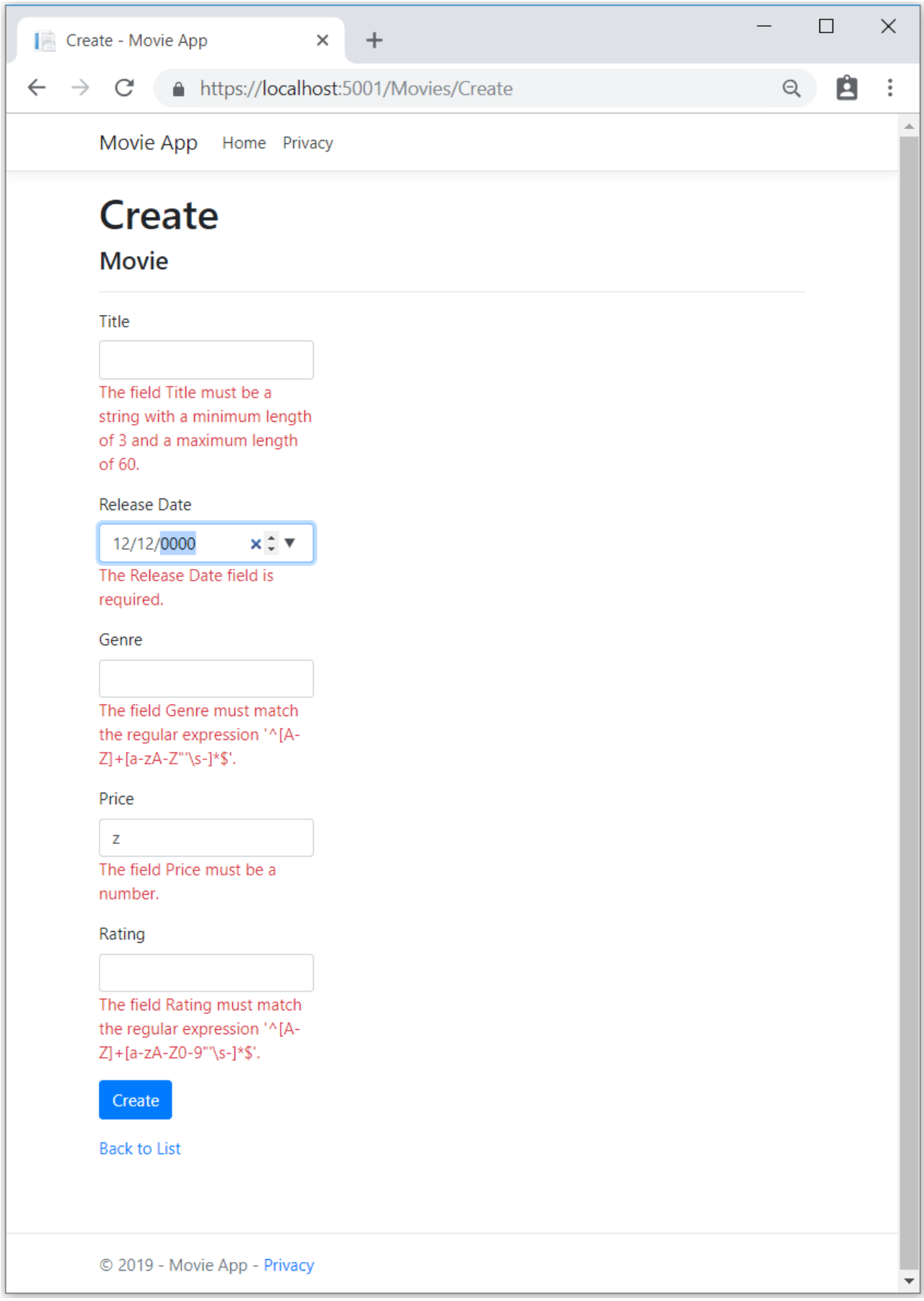
- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White space, numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI

Run the app and navigate to the Movies controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



NOTE

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the *Create.cshtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data isn't sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the Fiddler tool , or the F12 Developer tools.

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

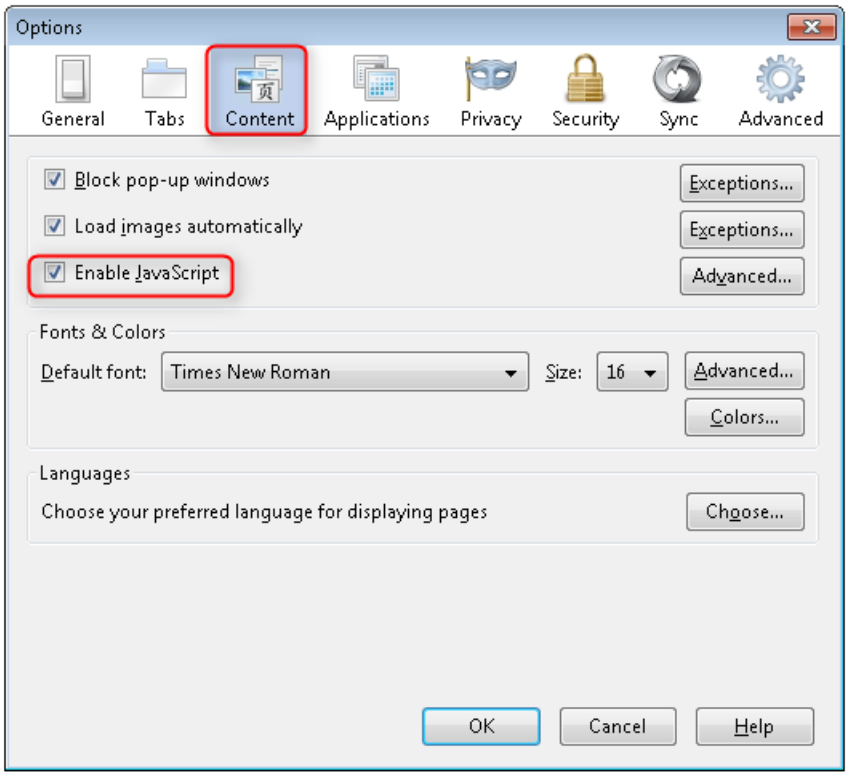
```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

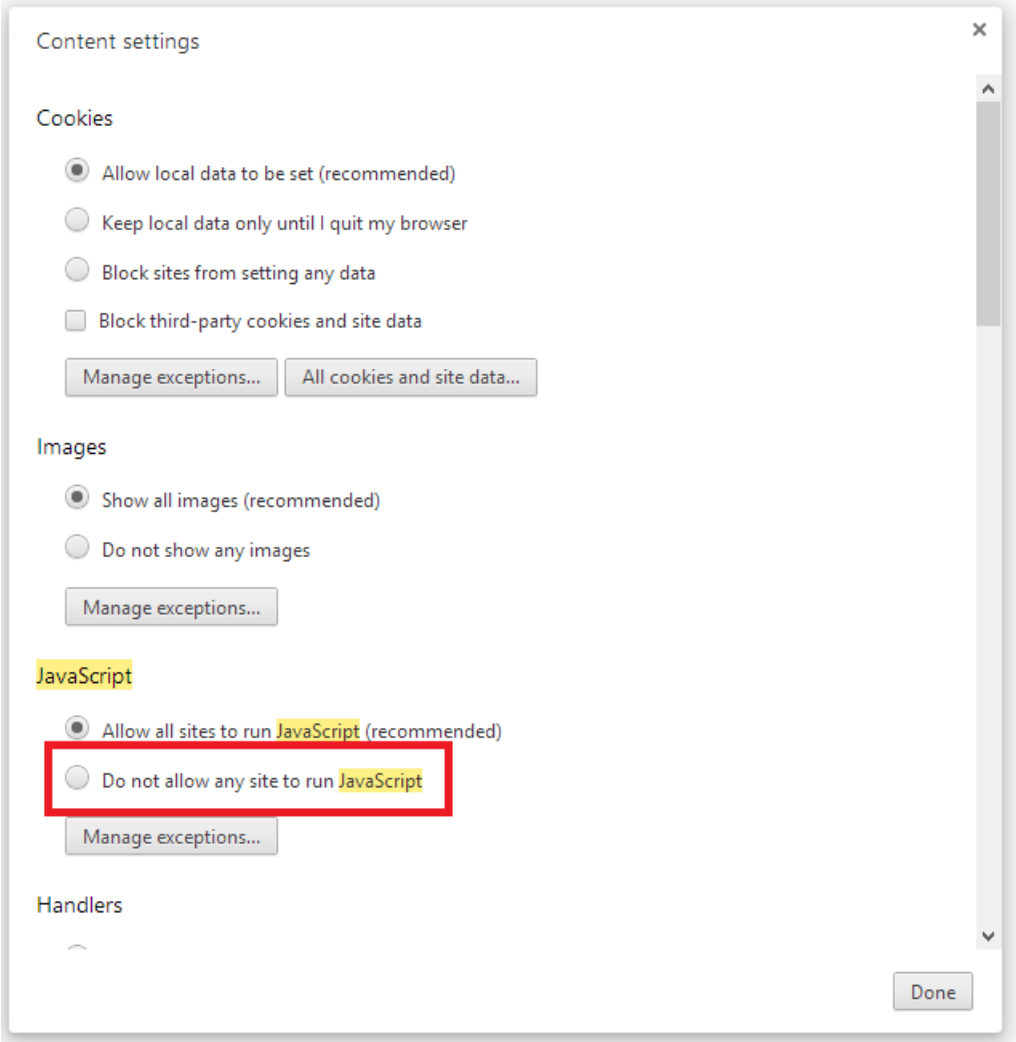
The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation won't submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

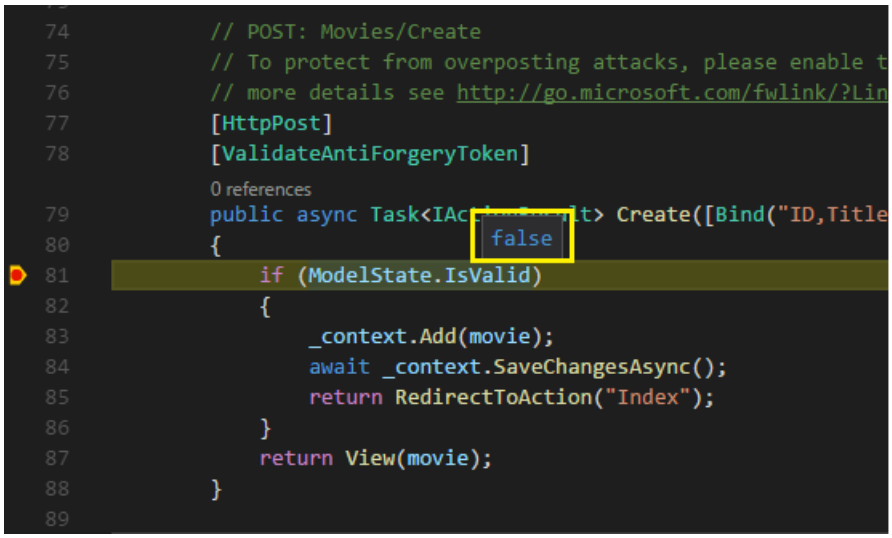
The following image shows how to disable JavaScript in the Firefox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.



The portion of the *Create.cshtml* view template is shown in the following markup:

```
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>

        @*Markup removed for brevity.*@
```

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The Validation Tag Helper displays validation errors. See Validation for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the *Movie.cs* file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supplies elements/attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

> **NOTE**
> jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:
>
> `[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]`

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

In the next part of the series, we review the app and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- Working with Forms
- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers

Part 10, examine the Details and Delete methods of an ASP.NET Core app

9/22/2020 • 3 minutes to read • Edit Online

By Rick Anderson

Open the Movie controller and examine the `Details` method:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method, and an `id` value. Recall these segments are defined in *Startup.cs*.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

EF makes it easy to search for data using the `FirstOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:{PORT}/Movies/Details/1` to something like `http://localhost:{PORT}/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the

signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

```
// POST: Movies/Delete/6
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

PREVIOUS

Build a Blazor todo list app

9/22/2020 • 4 minutes to read • Edit Online

By Daniel Roth and Luke Latham

This tutorial shows you how to build and modify a Blazor app. You learn how to:

- Create a todo list Blazor app project
- Modify Razor components
- Use event handling and data binding in components
- Use routing in a Blazor app

At the end of this tutorial, you'll have a working todo list app.

Prerequisites

.NET Core 3.1 SDK or later

Create a todo list Blazor app

1. Create a new Blazor app named `TodoList` in a command shell:

   ```
   dotnet new blazorserver -o TodoList
   ```

   The preceding command creates a folder named `TodoList` to hold the app. The `TodoList` folder is the *root folder* of the project. Change directories to the `TodoList` folder with the following command:

   ```
   cd TodoList
   ```

2. Add a new `Todo` Razor component to the app in the `Pages` folder using the following command:

   ```
   dotnet new razorcomponent -n Todo -o Pages
   ```

   > **IMPORTANT**
   > Razor component file names require a capitalized first letter. Open the `Pages` folder and confirm that the `Todo` component file name starts with a capital letter `T`. The file name should be `Todo.razor`.

3. In `Pages/Todo.razor` provide the initial markup for the component:

   ```
   @page "/todo"

   <h3>Todo</h3>
   ```

4. Add the `Todo` component to the navigation bar.

   The `NavMenu` component (`Shared/NavMenu.razor`) is used in the app's layout. Layouts are components that allow you to avoid duplication of content in the app.

Add a `<NavLink>` element for the `Todo` component by adding the following list item markup below the existing list items in the `Shared/NavMenu.razor` file:

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="todo">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Todo
    </NavLink>
</li>
```

5. Build and run the app by executing the `dotnet run` command in the command shell from the `TodoList` folder. Visit the new Todo page to confirm that the link to the `Todo` component works.

6. Add a `TodoItem.cs` file to the root of the project (the `TodoList` folder) to hold a class that represents a todo item. Use the following C# code for the `TodoItem` class:

```
public class TodoItem
{
    public string Title { get; set; }
    public bool IsDone { get; set; }
}
```

7. Return to the `Todo` component (`Pages/Todo.razor`):

   - Add a field for the todo items in an `@code` block. The `Todo` component uses this field to maintain the state of the todo list.
   - Add unordered list markup and a `foreach` loop to render each todo item as a list item (`<li>`).

```
@page "/todo"

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
}
```

8. The app requires UI elements for adding todo items to the list. Add a text input (`<input>`) and a button (`<button>`) below the unordered list (`<ul>...</ul>`):

```
@page "/todo"

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" />
<button>Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
}
```

9. Stop the running app in the command shell. Many command shells accept the keyboard command `Ctrl+c` to stop an app. Rebuild and run the app with the `dotnet run` command. When the `Add todo` button is selected, nothing happens because an event handler isn't wired up to the button.

10. Add an `AddTodo` method to the `Todo` component and register it for button selections using the `@onclick` attribute. The `AddTodo` C# method is called when the button is selected:

```
<input placeholder="Something todo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();

    private void AddTodo()
    {
        // Todo: Add the todo
    }
}
```

11. To get the title of the new todo item, add a `newTodo` string field at the top of the `@code` block and bind it to the value of the text input using the `bind` attribute in the `<input>` element:

```
private IList<TodoItem> todos = new List<TodoItem>();
private string newTodo;
```

```
<input placeholder="Something todo" @bind="newTodo" />
```

12. Update the `AddTodo` method to add the `TodoItem` with the specified title to the list. Clear the value of the text input by setting `newTodo` to an empty string:

```
@page "/todo"

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
    private string newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

13. Stop the running app in the command shell. Rebuild and run the app with the `dotnet run` command. Add some todo items to the todo list to test the new code.

14. The title text for each todo item can be made editable, and a check box can help the user keep track of completed items. Add a check box input for each todo item and bind its value to the `IsDone` property. Change `@todo.Title` to an `<input>` element bound to `@todo.Title`:

```
<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
        </li>
    }
</ul>
```

15. To verify that these values are bound, update the `<h3>` header to show a count of the number of todo items that aren't complete (`IsDone` is `false`).

```
<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>
```

16. The completed `Todo` component (`Pages/Todo.razor`):

```
@page "/todo"

<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
        </li>
    }
</ul>

<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
    private string newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

17. Stop the running app in the command shell. Rebuild and run the app with the `dotnet run` command. Add todo items to test the new code.

Next steps

In this tutorial, you learned how to:

- Create a todo list Blazor app project
- Modify Razor components
- Use event handling and data binding in components
- Use routing in a Blazor app

Learn about tooling for ASP.NET Core Blazor:

Tooling for ASP.NET Core Blazor

Tutorial: Create a web API with ASP.NET Core

9/22/2020 • 47 minutes to read • Edit Online

By Rick Anderson, Kirk Larkin, and Mike Wasson

This tutorial teaches the basics of building a web API with ASP.NET Core.

In this tutorial, you learn how to:

- Create a web API project.
- Add a model class and a database context.
- Scaffold a controller with CRUD methods.
- Configure routing, URL paths, and return values.
- Call the web API with Postman.

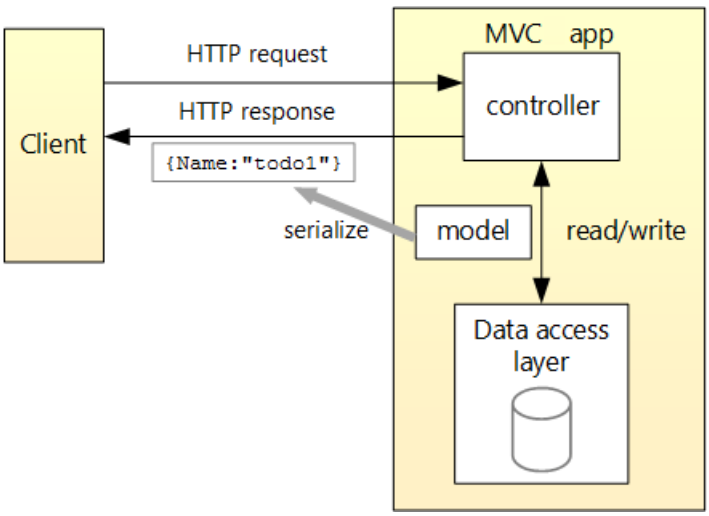
At the end, you have a web API that can manage "to-do" items stored in a database.

Overview

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
`GET /api/TodoItems`	Get all to-do items	None	Array of to-do items
`GET /api/TodoItems/{id}`	Get an item by ID	None	To-do item
`POST /api/TodoItems`	Add a new item	To-do item	To-do item
`PUT /api/TodoItems/{id}`	Update an existing item	To-do item	None
`DELETE /api/TodoItems/{id}`	Delete an item	None	None

The following diagram shows the design of the app.



Prerequisites

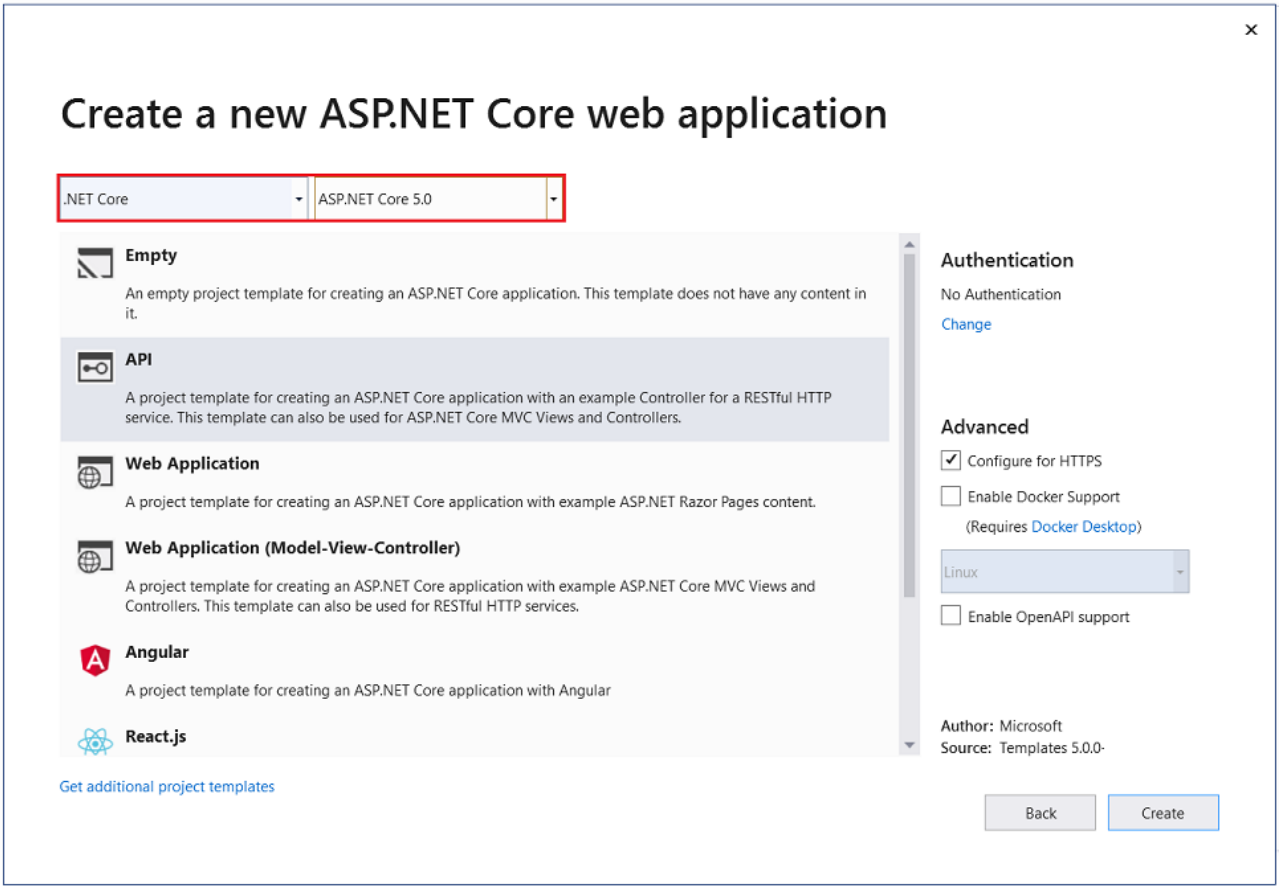
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.8 or later with the **ASP.NET and web development** workload
- .NET 5.0 SDK or later

Create a web project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the **File** menu, select **New** > **Project**.
- Select the **ASP.NET Core Web Application** template and click **Next**.
- Name the project *TodoApi* and click **Create**.
- In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 5.0** are selected. Select the **API** template and click **Create**.



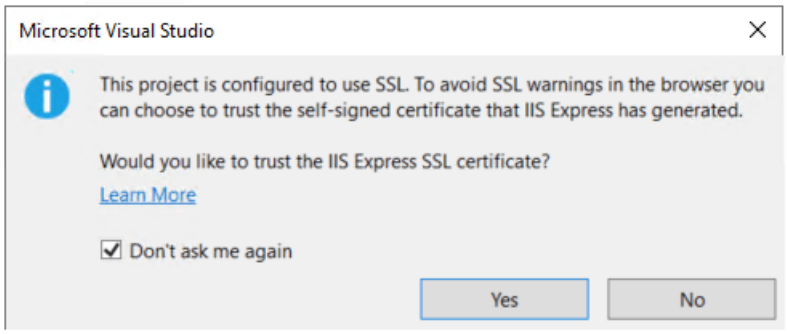
Test the project

The project template creates a `WeatherForecast` API with support for Swagger.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

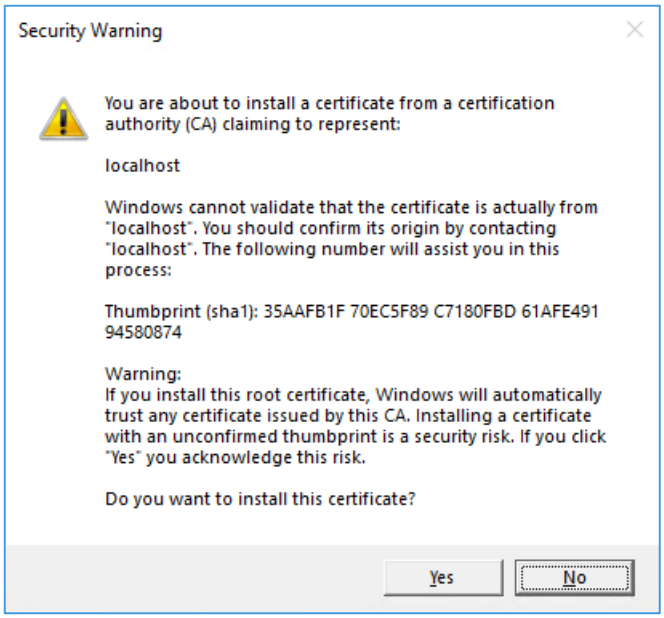
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

Visual Studio launches:

- The IIS Express web server.
- The default browser and navigates to `https://localhost:<port>/https://localhost:5001/swagger/index.html`, where `<port>` is a randomly chosen port number.

The Swagger page `/swagger/index.html` is displayed. Select **GET** > **Try it out** > **Execute**. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop down list box with media types and the example value and schema.

Swagger is used to generate useful documentation and help pages for web APIs. This tutorial focuses on creating a web API. For more information on Swagger, see ASP.NET Core Web API help pages with Swagger / OpenAPI.

Copy and past the **Request URL** in the browser: `https://localhost:<port>/WeatherForecast`

JSON similar to the following is returned:

```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
        "summary": "Chilly"
    }
]
```


Update the launchUrl

In *Properties\launchSettings.json*, update `launchUrl` from `"swagger"` to `"api/TodoItems"`:

```
"launchUrl": "api/TodoItems",
```

Because Swagger has been removed, the preceding markup changes the URL that is launched to the GET method of the controller added in the following sections.

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is a single `TodoItem` class.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.
- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the *Models* folder is used by convention.

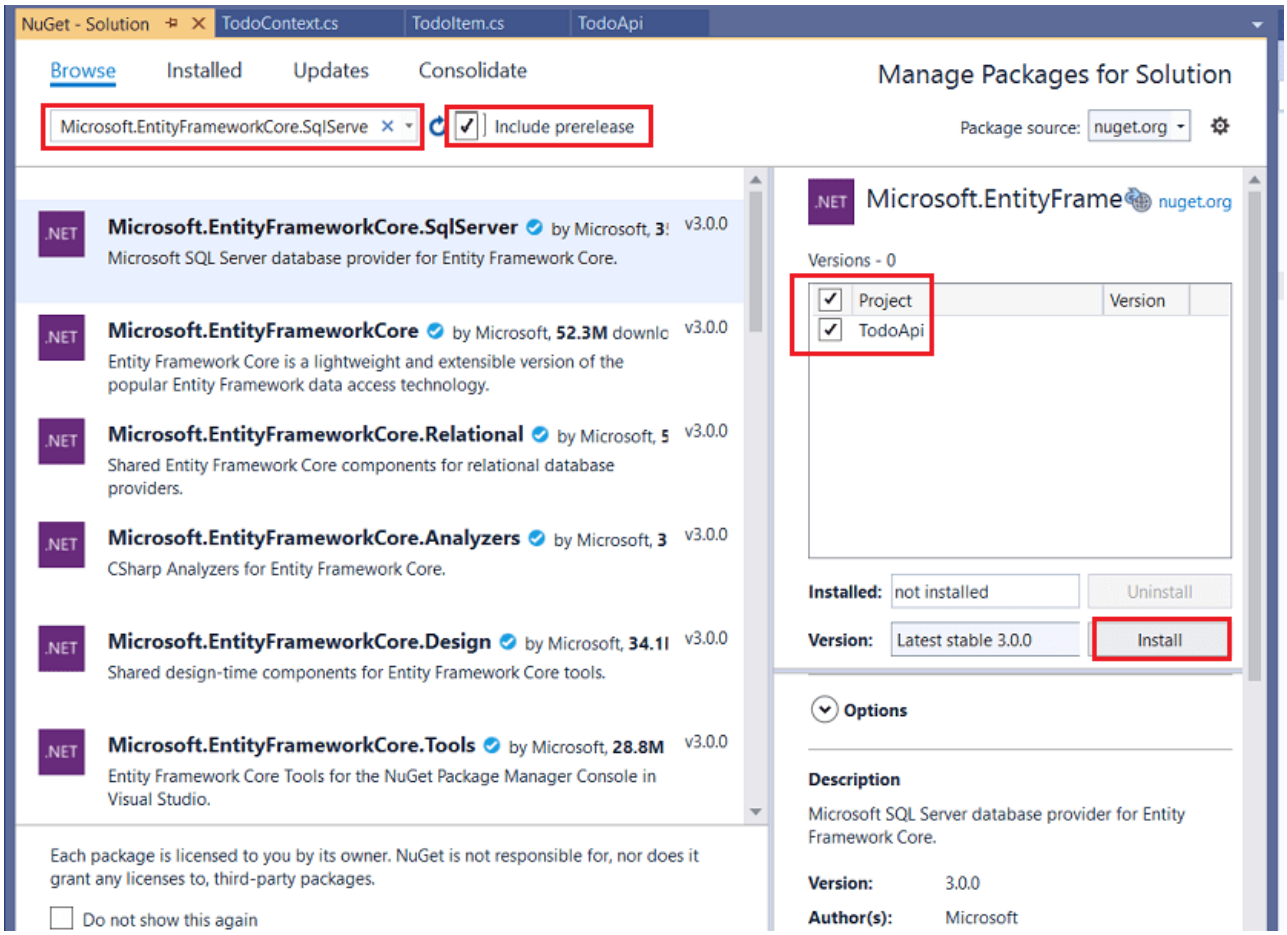
Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add NuGet packages

- From the **Tools** menu, select **NuGet Package Manager > Manage NuGet Packages for Solution**.
- Select the **Browse** tab, and then enter **Microsoft. **EntityFrameworkCore.SqlServer** in the search box.
- Select the **Include prerelease** checkbox so the 5.0 RC version is available.
- Select **Microsoft.EntityFrameworkCore.SqlServer** in the left pane.
- Select the **Project** check box in the right pane and then select **Install**.
- Use the preceding instructions to add the **Microsoft.EntityFrameworkCore.InMemory** NuGet package.



Add the TodoContext database context

- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.
- Enter the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update *Startup.cs* with the following code:

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
                                               opt.UseInMemoryDatabase("TodoList"));
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

The preceding code:

- Removes the Swagger calls.
- Removes unused `using` declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Controllers* folder.
- Select **Add** > **New Scaffolded Item**.

- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

The generated code:

- Marks the class with the `[ApiController]` attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name is excluded from the route. That is, the action's associated method name isn't used in the matching route.

Update the PostTodoItem create method

Replace the return statement in the `PostTodoItem` to use the nameof operator:

```
// POST: api/TodoItems
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
}
```

The preceding code is an HTTP POST method, as indicated by the `[HttpPost]` attribute. The method gets the value of the to-do item from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

Install Postman

This tutorial uses Postman to test the web API.

- Install Postman

- Start the web app.
- Start Postman.
- Disable **SSL certificate verification**
  - From **File** > **Settings** (**General** tab), disable **SSL certificate verification**.

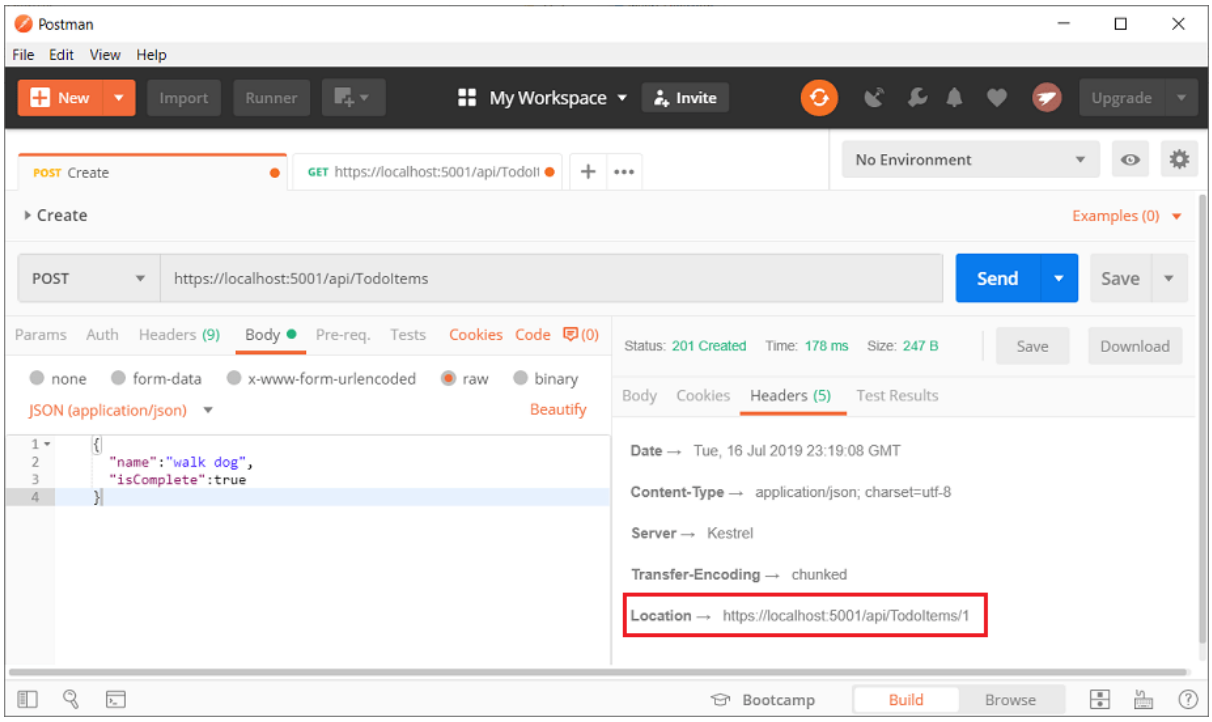
> **WARNING**
> Re-enable SSL certificate verification after testing the controller.

Test PostTodoItem with Postman

- Create a new request.
- Set the HTTP method to `POST`.
- Set the URI to `https://localhost:<port>/api/TodoItems`. For example, `https://localhost:5001/api/TodoItems`.
- Select the **Body** tab.
- Select the **raw** radio button.
- Set the type to **JSON (application/json)**.
- In the request body enter JSON for a to-do item:

```
{
  "name":"walk dog",
  "isComplete":true
}
```

- Select **Send**.



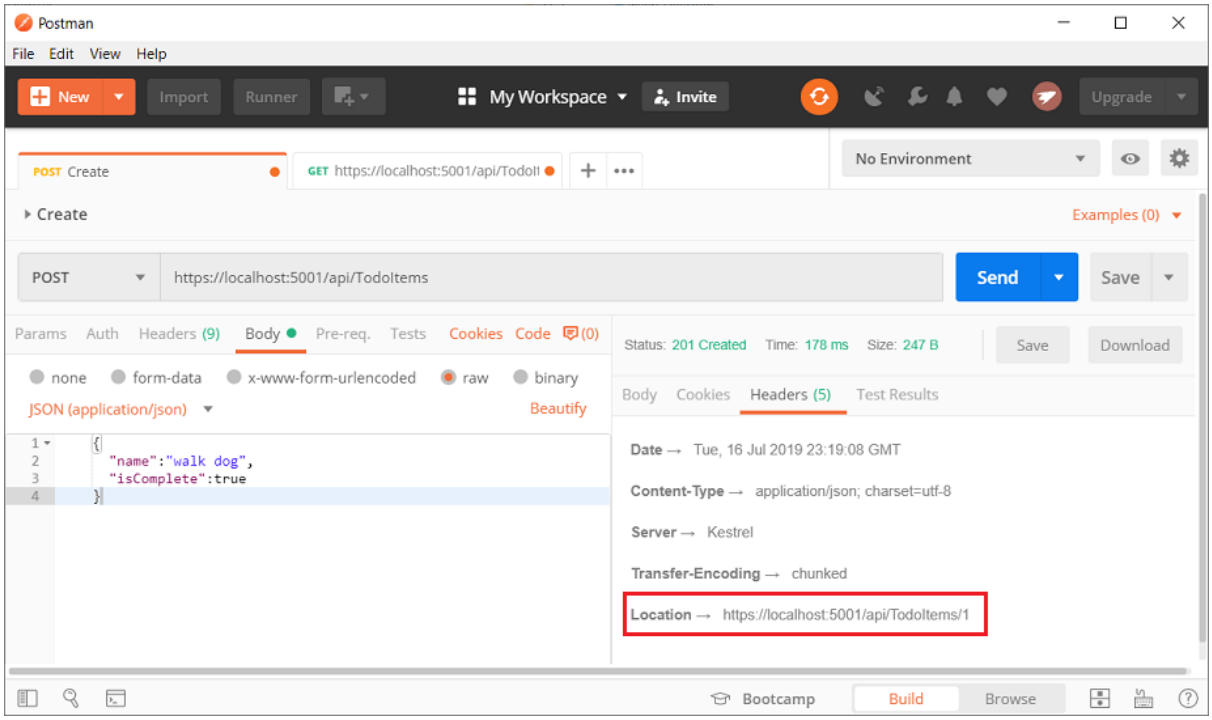
Test the location header URI

The location header URI can be tested in the browser. Copy and paste the location header URI into the browser.

To test in Postman:

- Select the **Headers** tab in the **Response** pane.

- Copy the **Location** header value:



- Set the HTTP method to `GET`.
- Set the URI to `https://localhost:<port>/api/TodoItems/1`. For example, `https://localhost:5001/api/TodoItems/1`.
- Select **Send**.

Examine the GET methods

Two GET endpoints are implemented:

- `GET /api/TodoItems`
- `GET /api/TodoItems/{id}`

Test the app by calling the two endpoints from a browser or Postman. For example:

- `https://localhost:5001/api/TodoItems`
- `https://localhost:5001/api/TodoItems/1`

A response similar to the following is produced by the call to `GetTodoItems`:

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Test Get with Postman

- Create a new request.
- Set the HTTP method to **GET**.
- Set the request URI to `https://localhost:<port>/api/TodoItems`. For example, `https://localhost:5001/api/TodoItems`.

- Set **Two pane view** in Postman.
- Select **Send**.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not return any data. If no data is returned, POST data to the app.

Routing and URL paths

The `[HttpGet]` attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
    {
        _context = context;
    }
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.
- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

```
// GET: api/TodoItems/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

The PutTodoItem method

Examine the `PutTodoItem` method:

```
// PUT: api/TodoItems/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses HTTP PUT. The response is 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

If you get an error calling `PutTodoItem`, call `GET` to ensure there's an item in the database.

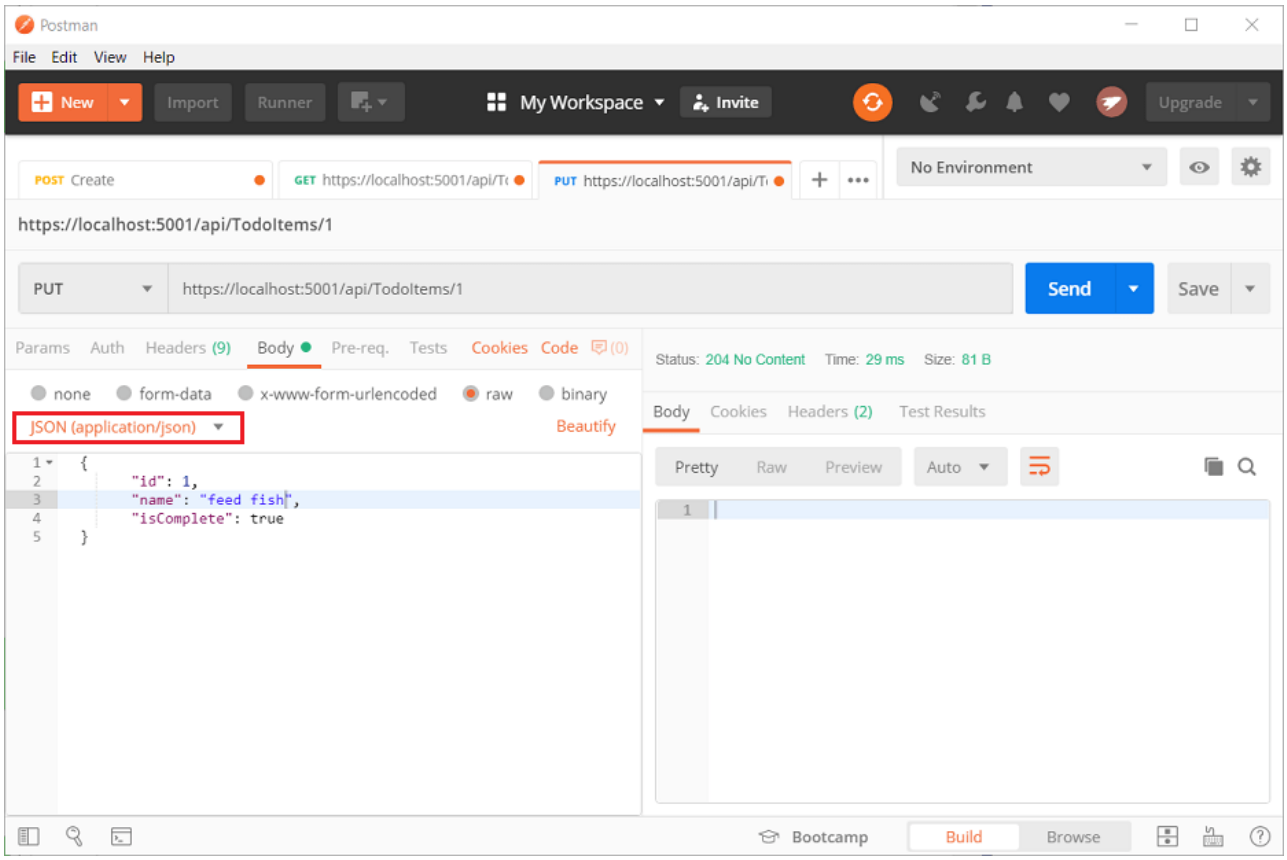
Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to `"feed fish"`:

```
{
  "Id":1,
  "name":"feed fish",
  "isComplete":true
}
```

The following image shows the Postman update:



The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

```
// DELETE: api/TodoItems/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Test the DeleteTodoItem method

Use Postman to delete a to-do item:

- Set the method to `DELETE`.
- Set the URI of the object to delete (for example `https://localhost:5001/api/TodoItems/1`).
- Select **Send**.

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
        public string Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

```
public class TodoItemDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

```
// GET: api/TodoItems
[HttpGet]
public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
{
    return await _context.TodoItems
        .Select(x => ItemToDTO(x))
        .ToListAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return ItemToDTO(todoItem);
}

[HttpPut("{id}")]
public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO todoItemDTO)
{
    if (id != todoItemDTO.Id)
    {
```

```
    {
        return BadRequest();
    }

    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    todoItem.Name = todoItemDTO.Name;
    todoItem.IsComplete = todoItemDTO.IsComplete;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
    {
        return NotFound();
    }

    return NoContent();
}

[HttpPost]
public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO todoItemDTO)
{
    var todoItem = new TodoItem
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    return CreatedAtAction(
        nameof(GetTodoItem),
        new { id = todoItem.Id },
        ItemToDTO(todoItem));
}

[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}

private bool TodoItemExists(long id) =>
     _context.TodoItems.Any(e => e.Id == id);

private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
    new TodoItemDTO
    {
        Id = todoItem.Id,
        Name = todoItem.Name,
        IsComplete = todoItem.IsComplete
    };
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

In this tutorial, you learn how to:

- Create a web API project.
- Add a model class and a database context.
- Scaffold a controller with CRUD methods.
- Configure routing, URL paths, and return values.
- Call the web API with Postman.

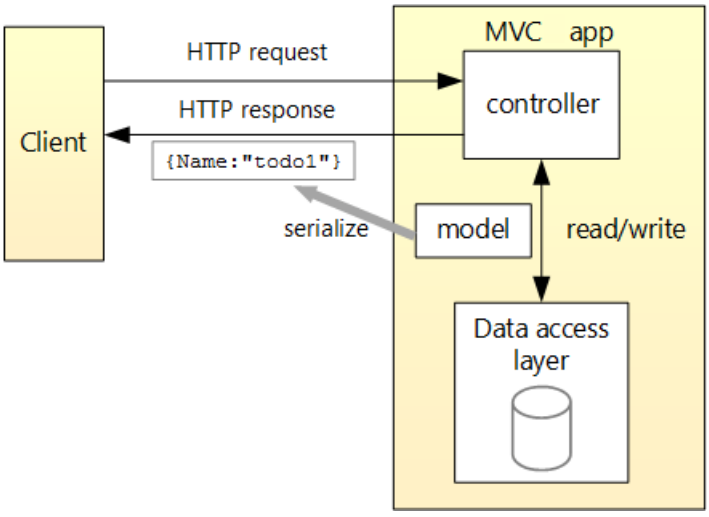
At the end, you have a web API that can manage "to-do" items stored in a database.

Overview

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
`GET /api/TodoItems`	Get all to-do items	None	Array of to-do items
`GET /api/TodoItems/{id}`	Get an item by ID	None	To-do item
`POST /api/TodoItems`	Add a new item	To-do item	To-do item
`PUT /api/TodoItems/{id}`	Update an existing item	To-do item	None
`DELETE /api/TodoItems/{id}`	Delete an item	None	None

The following diagram shows the design of the app.



Prerequisites

- Visual Studio

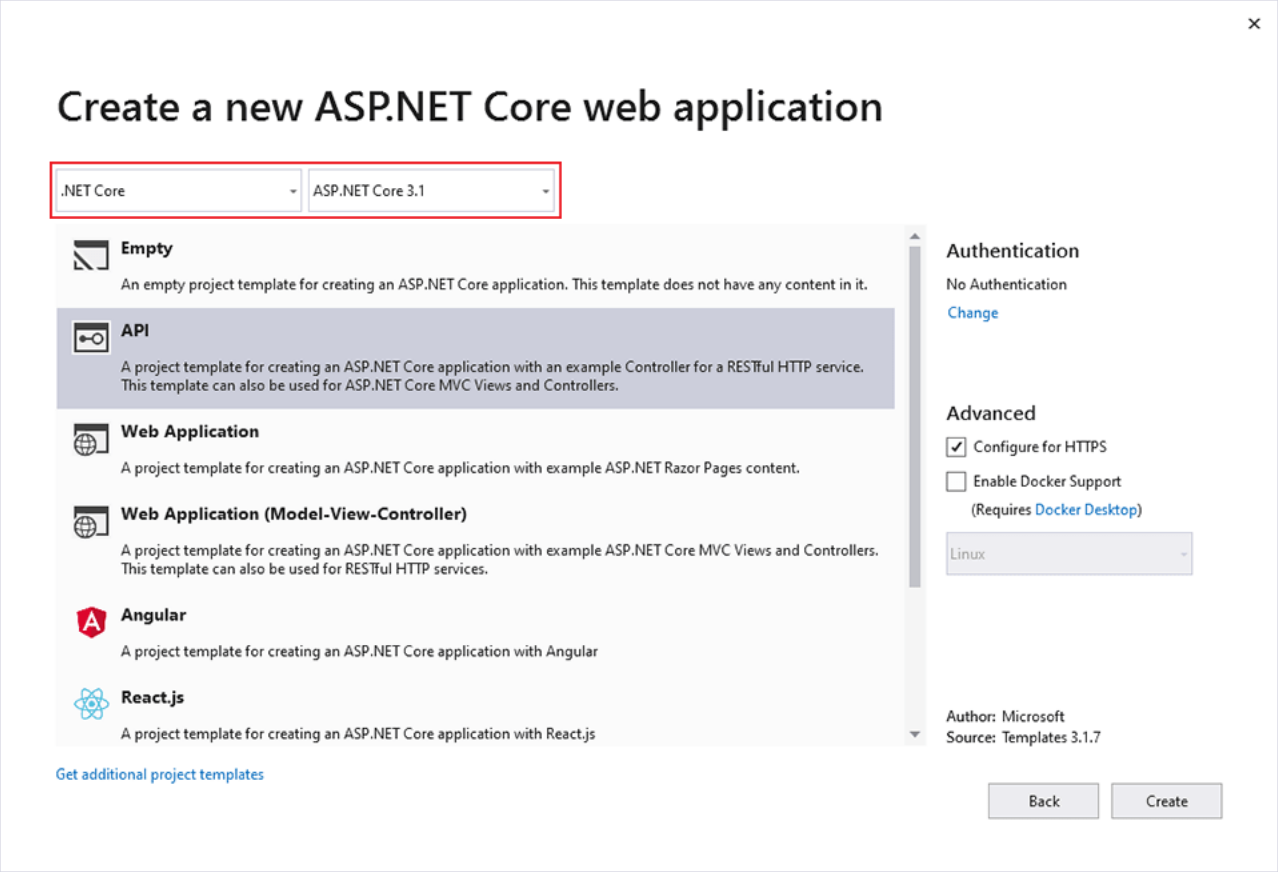
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a web project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the **File** menu, select **New** > **Project**.
- Select the **ASP.NET Core Web Application** template and click **Next**.
- Name the project *TodoApi* and click **Create**.
- In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 3.1** are selected. Select the **API** template and click **Create**.



Test the API

The project template creates a `WeatherForecast` API. Call the `Get` method from a browser to test the app.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Press Ctrl+F5 to run the app. Visual Studio launches a browser and navigates to `https://localhost:<port>/WeatherForecast`, where `<port>` is a randomly chosen port number.

If you get a dialog box that asks if you should trust the IIS Express certificate, select **Yes**. In the **Security Warning** dialog that appears next, select **Yes**.

JSON similar to the following is returned:

```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
        "summary": "Chilly"
    }
]
```

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is a single `TodoItem` class.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.
- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following code:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the *Models* folder is used by convention.

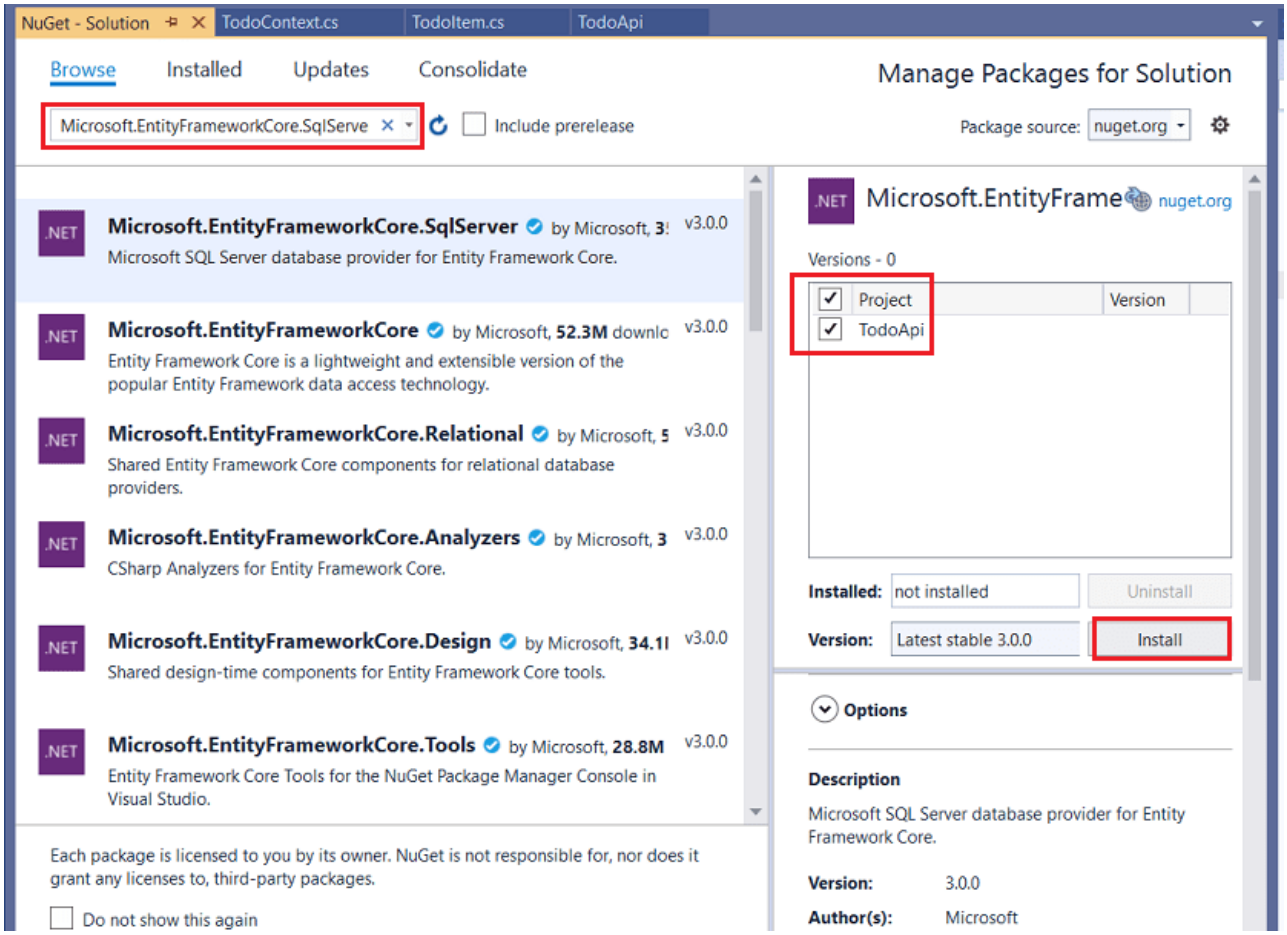
Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add NuGet packages

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab, and then enter **Microsoft.EntityFrameworkCore.SqlServer** in the search box.
- Select **Microsoft.EntityFrameworkCore.SqlServer** in the left pane.
- Select the **Project** check box in the right pane and then select **Install**.
- Use the preceding instructions to add the **Microsoft.EntityFrameworkCore.InMemory** NuGet package.



Add the TodoContext database context

- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.
- Enter the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update *Startup.cs* with the following highlighted code:

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
               opt.UseInMemoryDatabase("TodoList"));
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

The preceding code:

- Removes unused `using` declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Controllers* folder.
- Select **Add** > **New Scaffolded Item**.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.

- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

The generated code:

- Marks the class with the `[ApiController]` attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name is excluded from the route. That is, the action's associated method name isn't used in the matching route.

Examine the PostTodoItem create method

Replace the return statement in the `PostTodoItem` to use the nameof operator:

```
// POST: api/TodoItems
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
}
```

The preceding code is an HTTP POST method, as indicated by the `[HttpPost]` attribute. The method gets the value of the to-do item from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

Install Postman

This tutorial uses Postman to test the web API.

- Install Postman
- Start the web app.
- Start Postman.

- Disable **SSL certificate verification**
  - From **File** > **Settings** (**General** tab), disable **SSL certificate verification**.

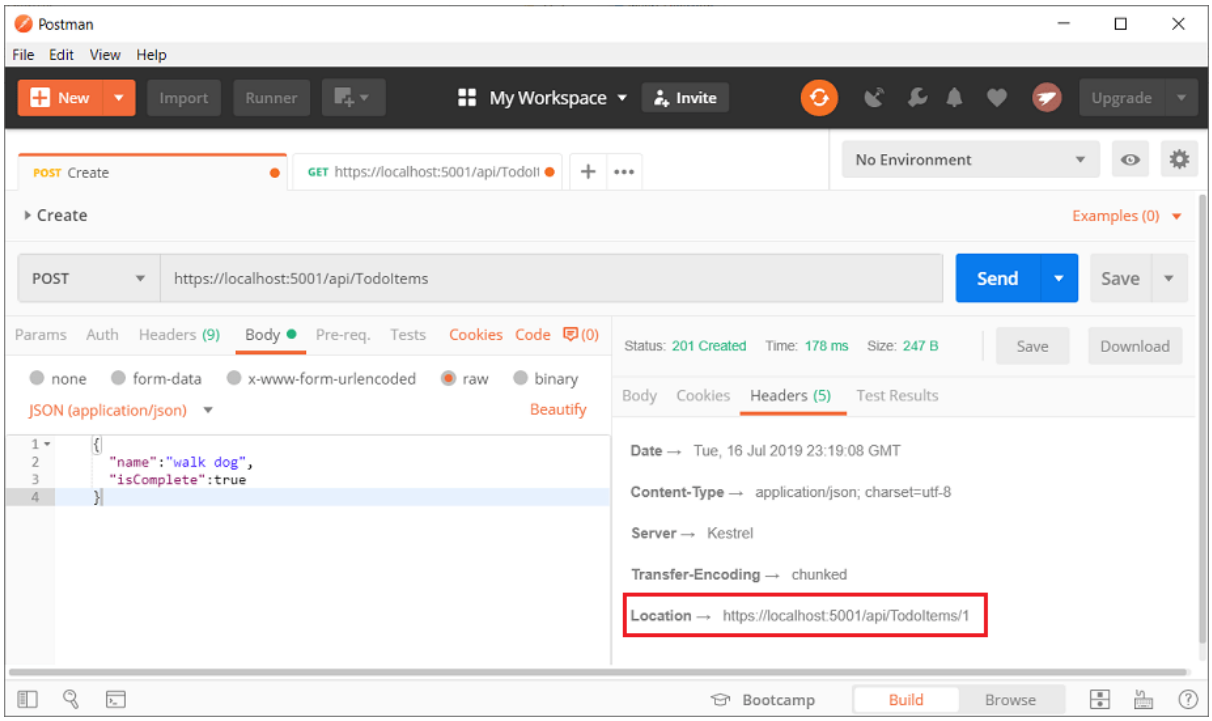
> **WARNING**
> Re-enable SSL certificate verification after testing the controller.

Test PostTodoItem with Postman

- Create a new request.
- Set the HTTP method to `POST`.
- Set the URI to `https://localhost:<port>/api/TodoItems`. For example, `https://localhost:5001/api/TodoItems`.
- Select the **Body** tab.
- Select the **raw** radio button.
- Set the type to **JSON (application/json)**.
- In the request body enter JSON for a to-do item:

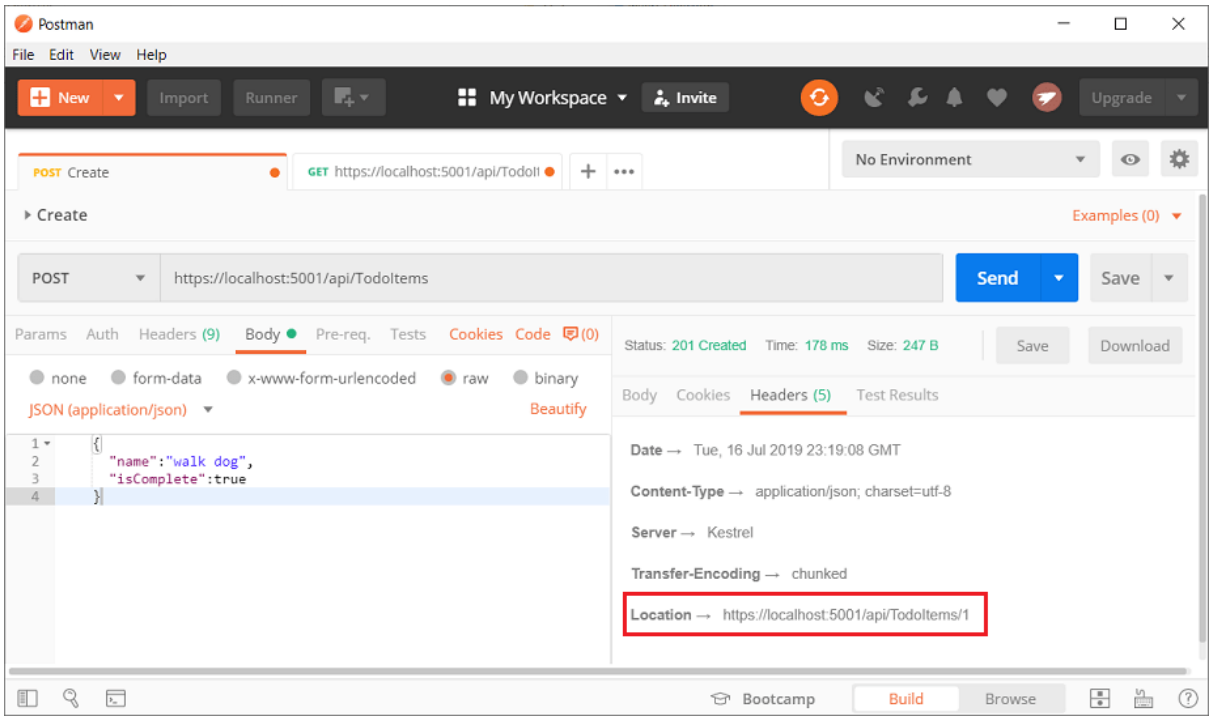
```
{
  "name":"walk dog",
  "isComplete":true
}
```

- Select **Send**.



Test the location header URI with Postman

- Select the **Headers** tab in the **Response** pane.
- Copy the **Location** header value:



- Set the HTTP method to `GET`.
- Set the URI to `https://localhost:<port>/api/TodoItems/1`. For example, `https://localhost:5001/api/TodoItems/1`.
- Select **Send**.

Examine the GET methods

These methods implement two GET endpoints:

- `GET /api/TodoItems`
- `GET /api/TodoItems/{id}`

Test the app by calling the two endpoints from a browser or Postman. For example:

- `https://localhost:5001/api/TodoItems`
- `https://localhost:5001/api/TodoItems/1`

A response similar to the following is produced by the call to `GetTodoItems`:

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Test Get with Postman

- Create a new request.
- Set the HTTP method to **GET**.
- Set the request URI to `https://localhost:<port>/api/TodoItems`. For example, `https://localhost:5001/api/TodoItems`.
- Set **Two pane view** in Postman.
- Select **Send**.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not return any data. If no data is returned, POST data to the app.

Routing and URL paths

The `[HttpGet]` attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
    {
        _context = context;
    }
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.
- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

```
// GET: api/TodoItems/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200

response.

The PutTodoItem method

Examine the `PutTodoItem` method:

```
// PUT: api/TodoItems/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses HTTP PUT. The response is 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

If you get an error calling `PutTodoItem`, call `GET` to ensure there's an item in the database.

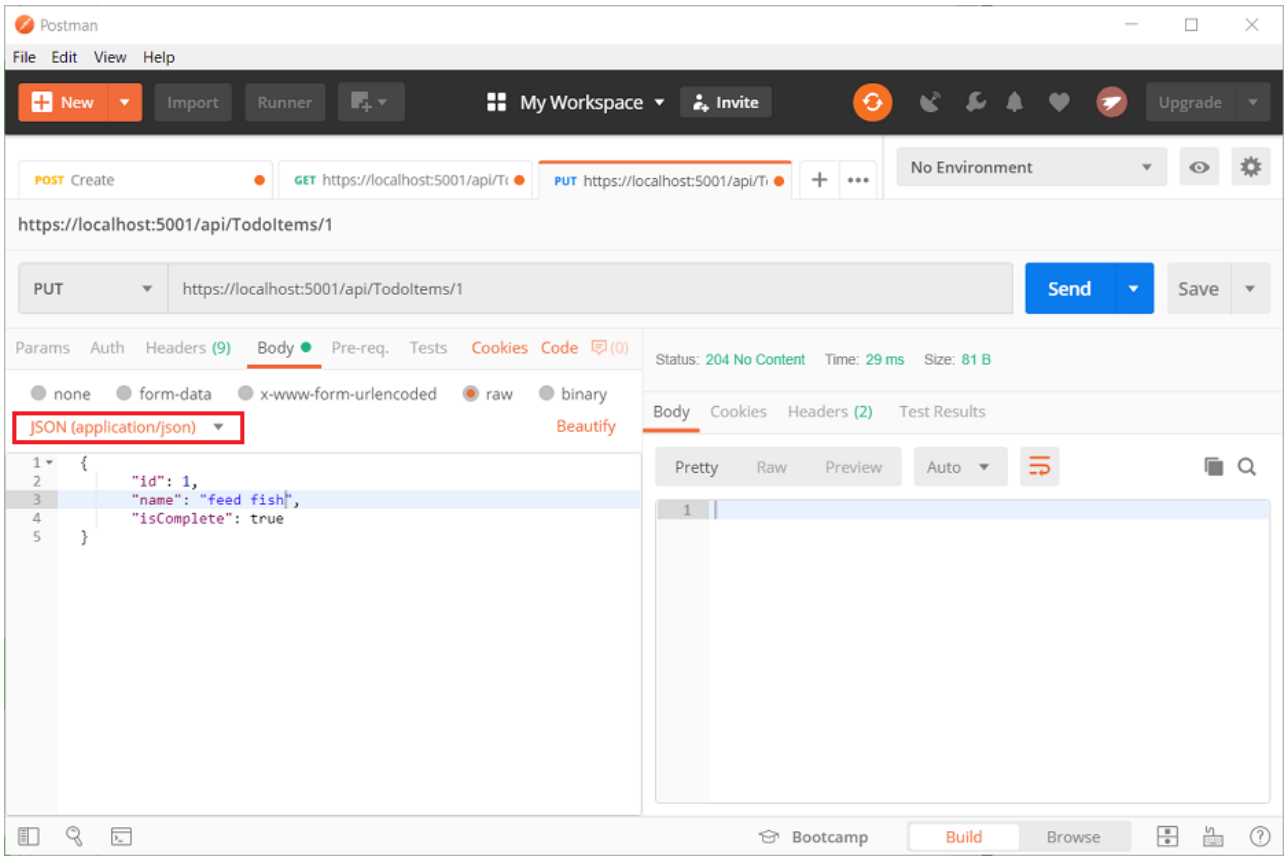
Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to "feed fish":

```
{
  "Id":1,
  "name":"feed fish",
  "isComplete":true
}
```

The following image shows the Postman update:



The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

```
// DELETE: api/TodoItems/5
[HttpDelete("{id}")]
public async Task<ActionResult<TodoItem>> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return todoItem;
}
```

Test the DeleteTodoItem method

Use Postman to delete a to-do item:

- Set the method to `DELETE`.
- Set the URI of the object to delete (for example `https://localhost:5001/api/TodoItems/1`).
- Select **Send**.

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
    public string Secret { get; set; }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

```
public class TodoItemDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

```
    [HttpGet]
    public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
        return await _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO todoItemDTO)
    {
        if (id != todoItemDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
```

```
        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoItemDTO.Name;
        todoItem.IsComplete = todoItemDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }

    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO todoItemDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoItemDTO.IsComplete,
            Name = todoItemDTO.Name
        };

        _context.TodoItems.Add(todoItem);
        await _context.SaveChangesAsync();

        return CreatedAtAction(
            nameof(GetTodoItem),
            new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        _context.TodoItems.Remove(todoItem);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool TodoItemExists(long id) =>
         _context.TodoItems.Any(e => e.Id == id);

    private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
        new TodoItemDTO
        {
            Id = todoItem.Id,
            Name = todoItem.Name,
            IsComplete = todoItem.IsComplete
        };
}
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

In this tutorial, you learn how to:

- Create a web API project.
- Add a model class and a database context.
- Add a controller.
- Add CRUD methods.
- Configure routing and URL paths.
- Specify return values.
- Call the web API with Postman.
- Call the web API with JavaScript.

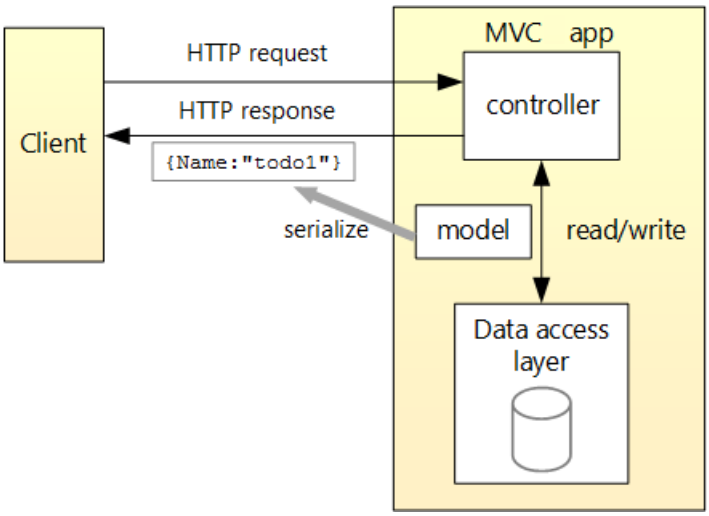
At the end, you have a web API that can manage "to-do" items stored in a relational database.

Overview 2.1

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/TodoItems	Get all to-do items	None	Array of to-do items
GET /api/TodoItems/{id}	Get an item by ID	None	To-do item
POST /api/TodoItems	Add a new item	To-do item	To-do item
PUT /api/TodoItems/{id}	Update an existing item	To-do item	None
DELETE /api/TodoItems/{id}	Delete an item	None	None

The following diagram shows the design of the app.



Prerequisites 2.1

- Visual Studio
- Visual Studio Code

- Visual Studio for Mac

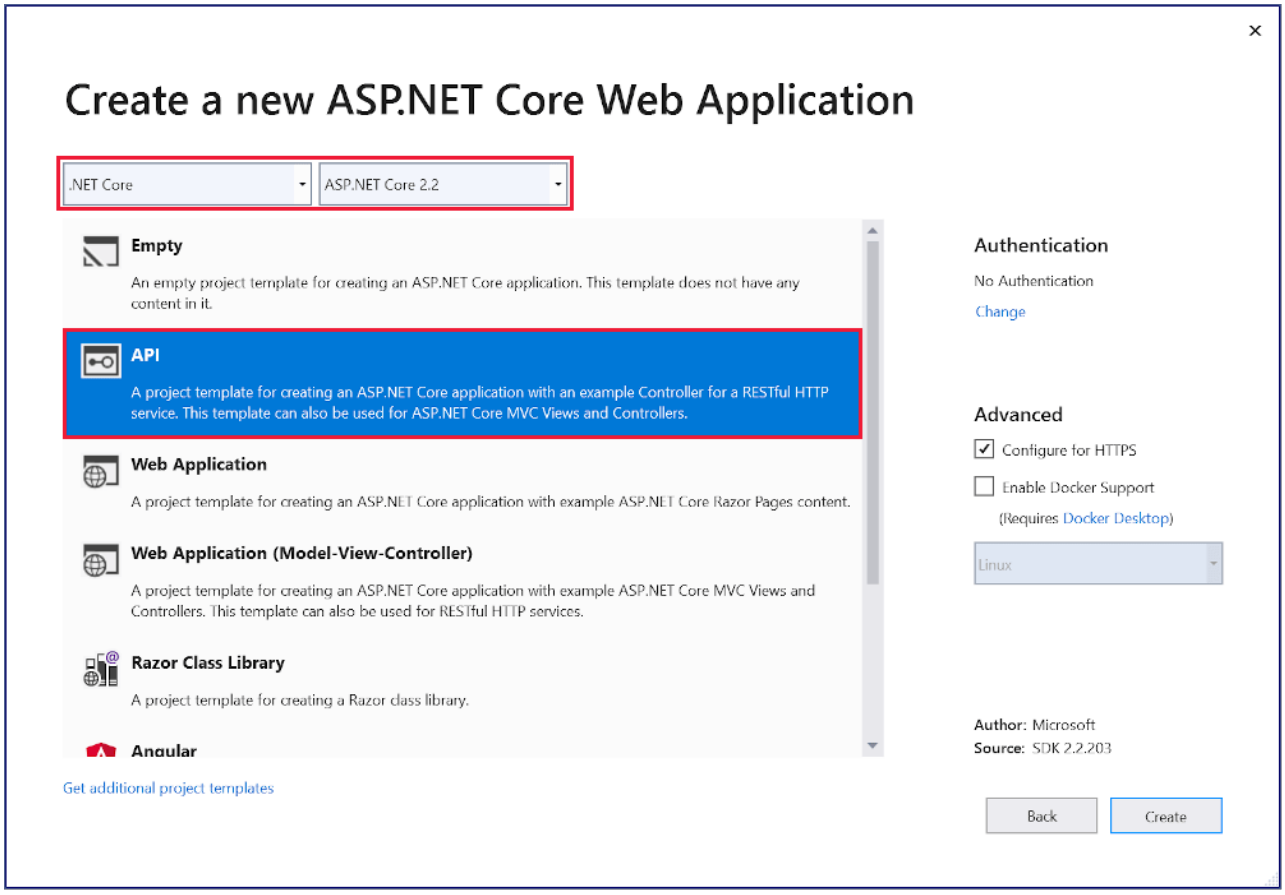
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

Create a web project 2.1

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the **File** menu, select **New** > **Project**.
- Select the **ASP.NET Core Web Application** template and click **Next**.
- Name the project *TodoApi* and click **Create**.
- In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 2.2** are selected. Select the **API** template and click **Create**. **Don't** select **Enable Docker Support**.



Test the API 2.1

The project template creates a `values` API. Call the `Get` method from a browser to test the app.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Press Ctrl+F5 to run the app. Visual Studio launches a browser and navigates to `https://localhost:<port>/api/values`, where `<port>` is a randomly chosen port number.

If you get a dialog box that asks if you should trust the IIS Express certificate, select **Yes**. In the **Security Warning** dialog that appears next, select **Yes**.

The following JSON is returned:

```
["value1","value2"]
```

Add a model class 2.1

A *model* is a set of classes that represent the data that the app manages. The model for this app is a single `TodoItem` class.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.
- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following code:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the *Models* folder is used by convention.

Add a database context 2.1

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.
- Replace the template code with the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context 2.1

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update *Startup.cs* with the following highlighted code:

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the
        //container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
                opt.UseInMemoryDatabase("TodoList"));
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        }

        // This method gets called by the runtime. Use this method to configure the HTTP
        //request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                // The default HSTS value is 30 days. You may want to change this for
                // production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseMvc();
        }
    }
}
```

The preceding code:

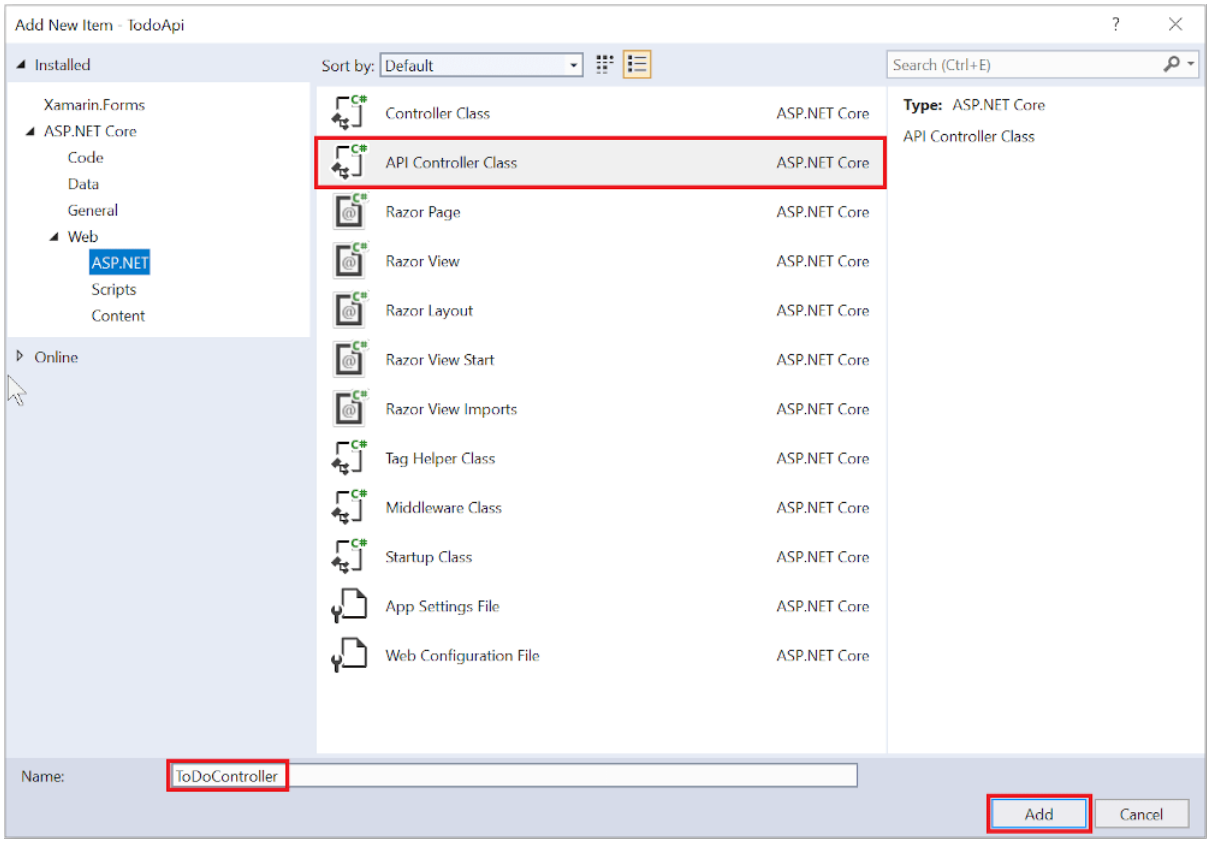
- Removes unused `using` declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Add a controller 2.1

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Controllers* folder.
- Select **Add** > **New Item**.

- In the **Add New Item** dialog, select the **API Controller Class** template.
- Name the class *TodoController*, and select **Add**.



- Replace the template code with the following code:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TodoApi.Models;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                // Create a new TodoItem if collection is empty,
                // which means you can't delete all TodoItems.
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

- Defines an API controller class without methods.
- Marks the class with the `[ApiController]` attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.
- Adds an item named `Item1` to the database if the database is empty. This code is in the constructor, so it runs every time there's a new HTTP request. If you delete all items, the constructor creates `Item1` again the next time an API method is called. So it may look like the deletion didn't work when it actually did work.

Add Get methods 2.1

To provide an API that retrieves to-do items, add the following methods to the `TodoController` class:

```
// GET: api/Todo
[HttpGet]
public async Task<ActionResult<IEnumerable<TodoItem>>> GetTodoItems()
{
    return await _context.TodoItems.ToListAsync();
}

// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

These methods implement two GET endpoints:

- `GET /api/todo`
- `GET /api/todo/{id}`

Stop the app if it's still running. Then run it again to include the latest changes.

Test the app by calling the two endpoints from a browser. For example:

- `https://localhost:<port>/api/todo`
- `https://localhost:<port>/api/todo/1`

The following HTTP response is produced by the call to `GetTodoItems` :

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Routing and URL paths 2.1

The `[HttpGet]` attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **Todo**Controller, so the controller name is "todo". ASP.NET Core routing is case insensitive.
- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values 2.1

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

Test the GetTodoItems method 2.1

This tutorial uses Postman to test the web API.

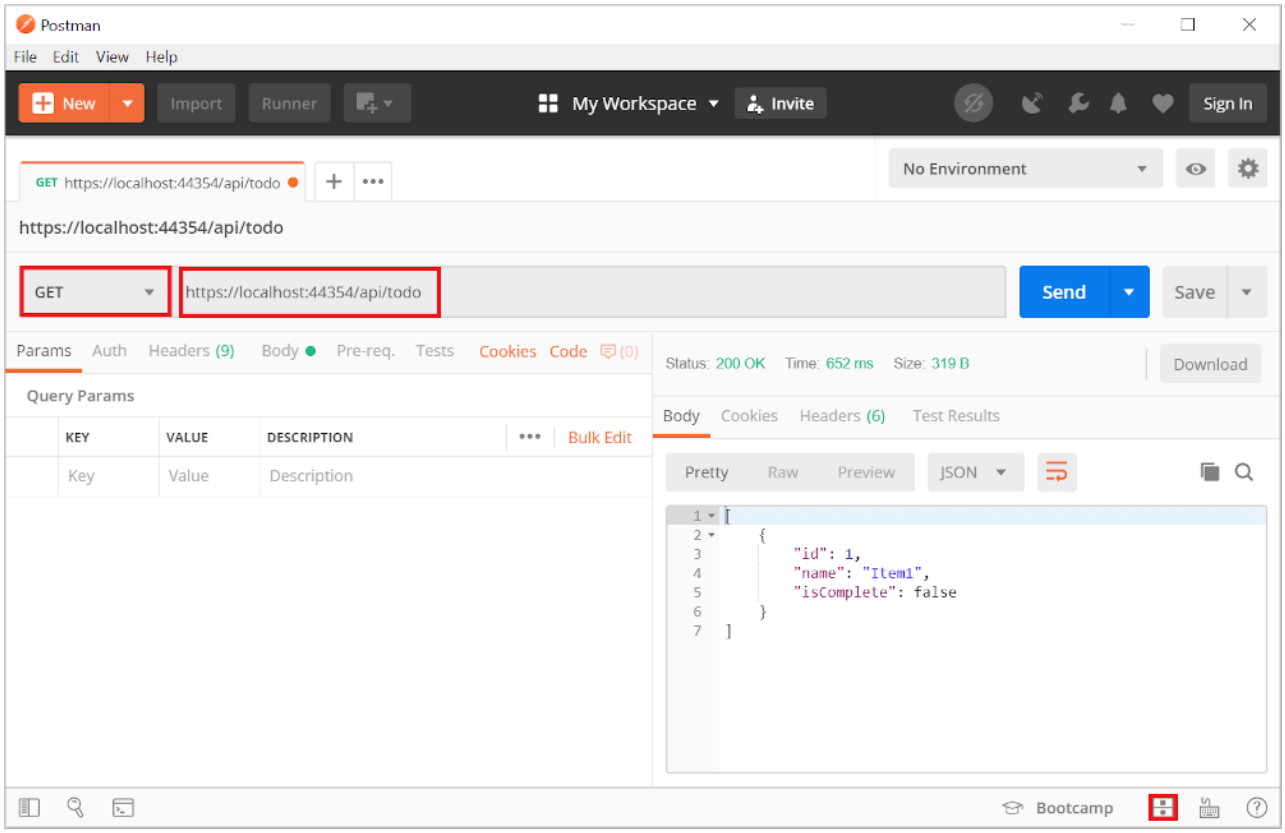
- Install Postman.
- Start the web app.
- Start Postman.
- Disable **SSL certificate verification**.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- From **File** > **Settings** (**General** tab), disable **SSL certificate verification**.

> **WARNING**
> Re-enable SSL certificate verification after testing the controller.

- Create a new request.
  - Set the HTTP method to **GET**.
  - Set the request URI to `https://localhost:<port>/api/todo`. For example, `https://localhost:5001/api/todo`.
- Set **Two pane view** in Postman.
- Select **Send**.



Add a Create method 2.1

Add the following `PostTodoItem` method inside of *Controllers/TodoController.cs*:

```
// POST: api/Todo
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem item)
{
    _context.TodoItems.Add(item);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(GetTodoItem), new { id = item.Id }, item);
}
```

The preceding code is an HTTP POST method, as indicated by the `[HttpPost]` attribute. The method gets the value of the to-do item from the body of the HTTP request.

The `CreatedAtAction` method:

- Returns an HTTP 201 status code, if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a `Location` header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

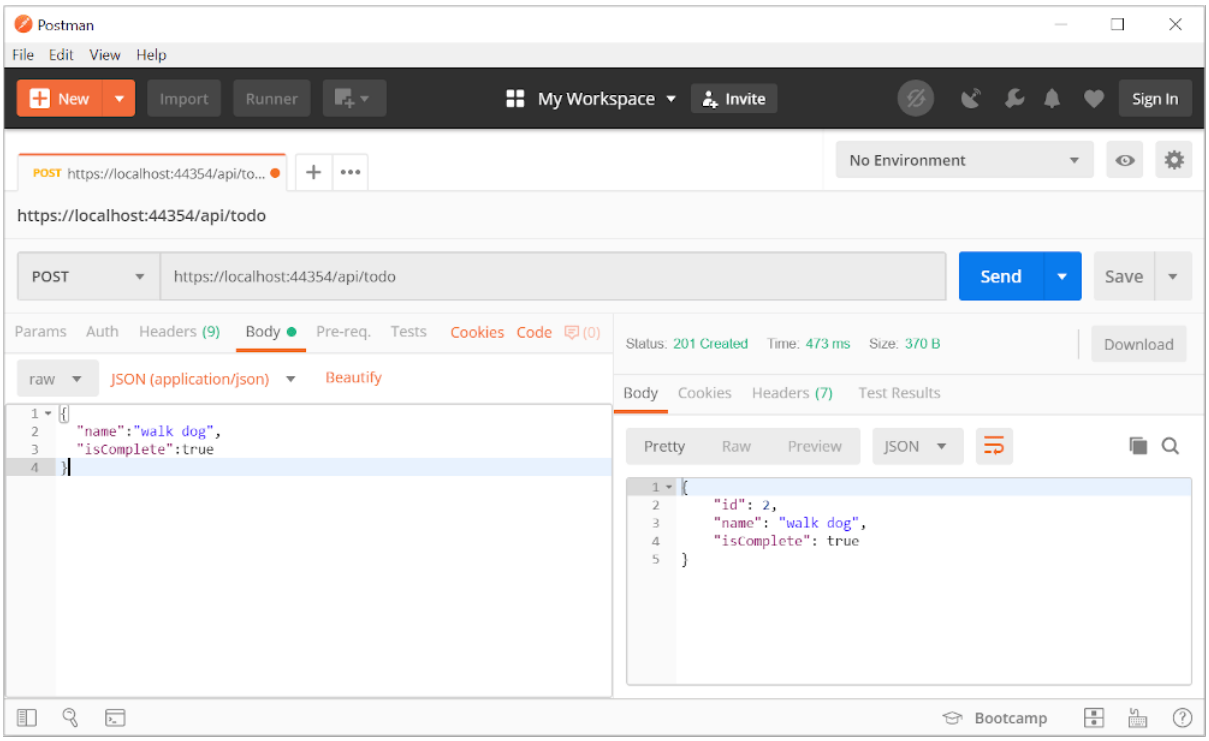
    return todoItem;
}
```

Test the PostTodoItem method 2.1

- Build the project.
- In Postman, set the HTTP method to `POST`.
- Set the URI to `https://localhost:<port>/api/TodoItem`. For example, `https://localhost:5001/api/TodoItem`.
- Select the **Body** tab.
- Select the **raw** radio button.
- Set the type to **JSON (application/json)**.
- In the request body enter JSON for a to-do item:

```
{
  "name":"walk dog",
  "isComplete":true
}
```

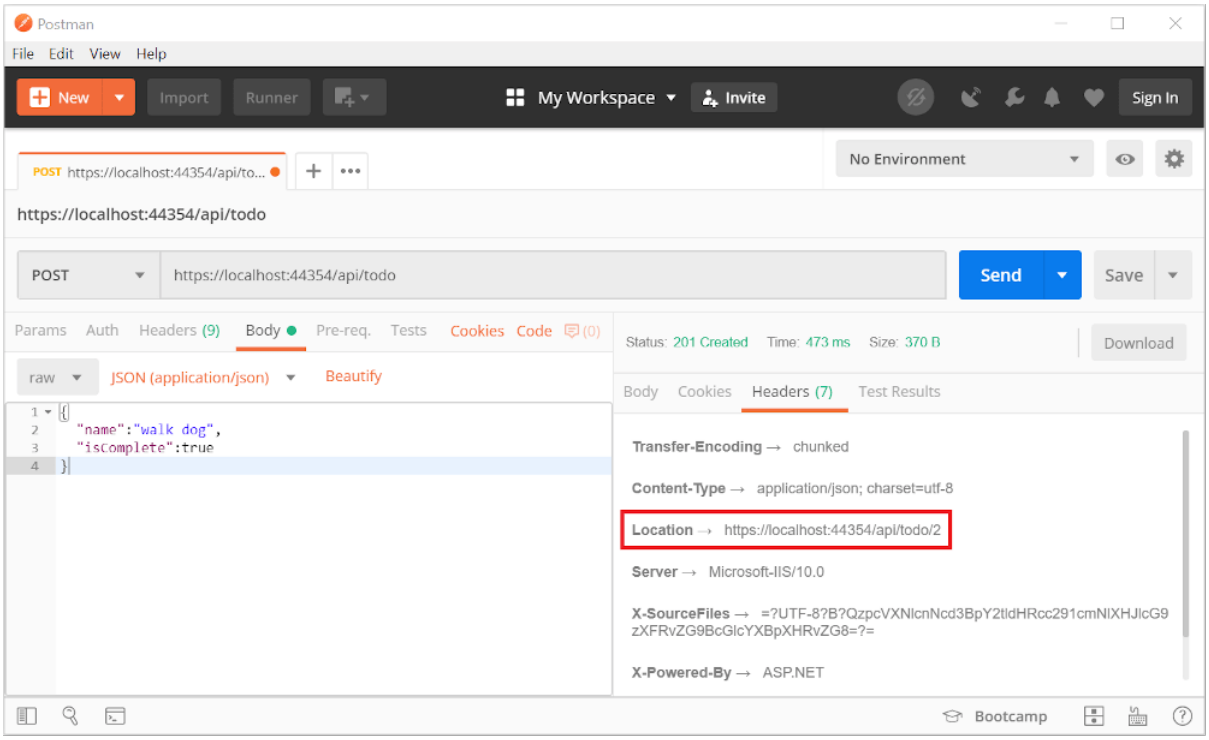
- Select **Send**.



If you get a 405 Method Not Allowed error, it's probably the result of not compiling the project after adding the `PostTodoItem` method.

Test the location header URI 2.1

- Select the **Headers** tab in the **Response** pane.
- Copy the **Location** header value:



- Set the method to GET. * Set the URI to `https://localhost:<port>/api/TodoItems/2`. For example, `https://localhost:5001/api/TodoItems/2`.
- Select **Send**.

Add a PutTodoItem method 2.1

Add the following `PutTodoItem` method:

```
// PUT: api/Todo/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem item)
{
    if (id != item.Id)
    {
        return BadRequest();
    }

    _context.Entry(item).State = EntityState.Modified;
    await _context.SaveChangesAsync();

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses HTTP PUT. The response is 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

If you get an error calling `PutTodoItem`, call `GET` to ensure there's an item in the database.

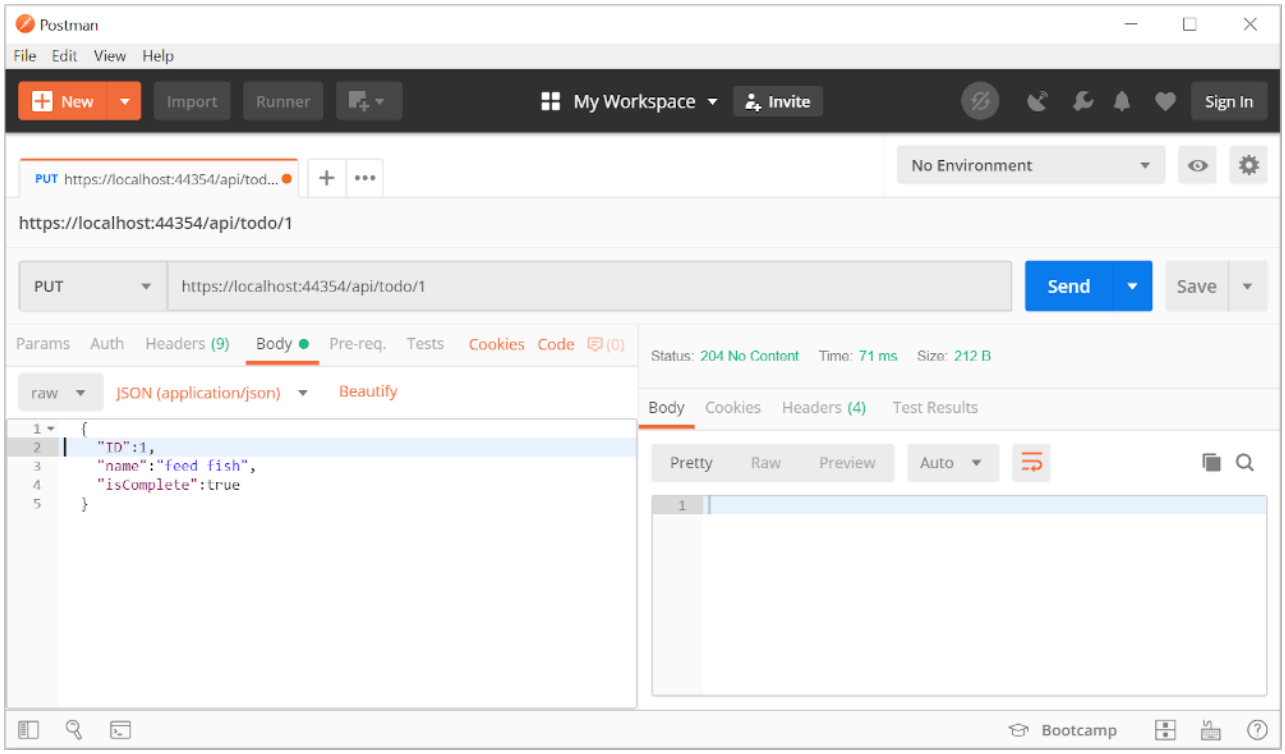
Test the PutTodoItem method 2.1

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to "feed fish":

```
{
  "Id":1,
  "name":"feed fish",
  "isComplete":true
}
```

The following image shows the Postman update:



Add a DeleteTodoItem method 2.1

Add the following `DeleteTodoItem` method:

```
// DELETE: api/Todo/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

The `DeleteTodoItem` response is 204 (No Content).

Test the DeleteTodoItem method 2.1

Use Postman to delete a to-do item:

- Set the method to `DELETE`.
- Set the URI of the object to delete (for example, `https://localhost:5001/api/todo/1`).
- Select **Send**.

The sample app allows you to delete all the items. However, when the last item is deleted, a new one is created by the model class constructor the next time the API is called.

Call the web API with JavaScript 2.1

In this section, an HTML page is added that uses JavaScript to call the web API. jQuery initiates the request. JavaScript updates the page with the details from the web API's response.

Configure the app to serve static files and enable default file mapping by updating *Startup.cs* with the following highlighted code:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change this for
        // production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseHttpsRedirection();
    app.UseMvc();
}
```

Create a *wwwroot* folder in the project directory.

Add an HTML file named *index.html* to the *wwwroot* directory. Replace its contents with the following markup:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }

        table {
            font-family: Arial, sans-serif;
            border: 1px solid;
            border-collapse: collapse;
        }

        th {
            background-color: #0066CC;
            color: white;
        }

        td {
            border: 1px solid;
            padding: 5px;
        }
    </style>
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="spoiler">
        <h3>Edit</h3>
        <form class="my-form">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"
            integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
```

```
            integrity="sha256-[illegible]"
            crossorigin="anonymous"></script>
    <script src="site.js"></script>
</body>
</html>
```

Add a JavaScript file named *site.js* to the *wwwroot* directory. Replace its contents with the following code:

```
const uri = "api/todo";
let todos = null;
function getCount(data) {
  const el = $("#counter");
  let name = "to-do";
  if (data) {
    if (data > 1) {
      name = "to-dos";
    }
    el.text(data + " " + name);
  } else {
    el.text("No " + name);
  }
}

$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });
```

```
      todos = data;
    }
  });
}

function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}

function deleteItem(id) {
  $.ajax({
    url: uri + "/" + id,
    type: "DELETE",
    success: function(result) {
      getData();
    }
  });
}

function editItem(id) {
  $.each(todos, function(key, item) {
    if (item.id === id) {
      $("#edit-name").val(item.name);
      $("#edit-id").val(item.id);
      $("#edit-isComplete")[0].checked = item.isComplete;
    }
  });
  $("#spoiler").css({ display: "block" });
}

$(".my-form").on("submit", function() {
  const item = {
    name: $("#edit-name").val(),
    isComplete: $("#edit-isComplete").is(":checked"),
    id: $("#edit-id").val()
  };

  $.ajax({
    url: uri + "/" + $("#edit-id").val(),
    type: "PUT",
    accepts: "application/json",
    contentType: "application/json",
    data: JSON.stringify(item),
    success: function(result) {
      getData();
    }
  });

  closeInput();
  return false;
```

```
  return false;
});

function closeInput() {
  $("#spoiler").css({ display: "none" });
}
```

A change to the ASP.NET Core project's launch settings may be required to test the HTML page locally:

- Open *Properties\launchSettings.json*.
- Remove the `launchUrl` property to force the app to open at *index.html*—the project's default file.

This sample calls all of the CRUD methods of the web API. Following are explanations of the calls to the API.

Get a list of to-do items 2.1

jQuery sends an HTTP GET request to the web API, which returns JSON representing an array of to-do items. The `success` callback function is invoked if the request succeeds. In the callback, the DOM is updated with the to-do information.

```
$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });

      todos = data;
    }
  });
}
```

Add a to-do item 2.1

jQuery sends an HTTP POST request with the to-do item in the request body. The `accepts` and `contentType` options are set to `application/json` to specify the media type being received and sent. The to-do item is converted to JSON by using JSON.stringify. When the API returns a successful status code, the `getData` function is invoked to update the HTML table.

```
function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}
```

Update a to-do item 2.1

Updating a to-do item is similar to adding one. The `url` changes to add the unique identifier of the item, and the `type` is `PUT`.

```
$.ajax({
  url: uri + "/" + $("#edit-id").val(),
  type: "PUT",
  accepts: "application/json",
  contentType: "application/json",
  data: JSON.stringify(item),
  success: function(result) {
    getData();
  }
});
```

Delete a to-do item 2.1

Deleting a to-do item is accomplished by setting the `type` on the AJAX call to `DELETE` and specifying the item's unique identifier in the URL.

```
$.ajax({
  url: uri + "/" + id,
  type: "DELETE",
  success: function(result) {
    getData();
  }
});
```

Add authentication support to a web API 2.1

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- IdentityServer4

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see Welcome to IdentityServer4.

Additional resources 2.1

View or download sample code for this tutorial. See how to download.

For more information, see the following resources:

- Create web APIs with ASP.NET Core
- ASP.NET Core Web API help pages with Swagger / OpenAPI
- Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8
- Routing to controller actions in ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Deploy ASP.NET Core apps to Azure App Service
- Host and deploy ASP.NET Core
- YouTube version of this tutorial

Create a web API with ASP.NET Core and MongoDB

9/22/2020 • 21 minutes to read • Edit Online

By Pratik Khandelwal and Scott Addie

This tutorial creates a web API that performs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

- Configure MongoDB
- Create a MongoDB database
- Define a MongoDB collection and schema
- Perform MongoDB CRUD operations from a web API
- Customize JSON serialization

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- .NET Core SDK 3.0 or later
- Visual Studio 2019 with the **ASP.NET and web development** workload
- MongoDB

Configure MongoDB

If using Windows, MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `Path` environment variable. This change enables MongoDB access from anywhere on your development machine.

Use the mongo Shell in the following steps to create a database, make collections, and store documents. For more information on mongo Shell commands, see Working with the mongo Shell.

1. Choose a directory on your development machine for storing the data. For example, *C:\BooksData* on Windows. Create the directory if it doesn't exist. The mongo Shell doesn't create new directories.

2. Open a command shell. Run the following command to connect to MongoDB on default port 27017. Remember to replace `<data_directory_path>` with the directory you chose in the previous step.

   ```
   mongod --dbpath <data_directory_path>
   ```

3. Open another command shell instance. Connect to the default test database by running the following command:

   ```
   mongo
   ```

4. Run the following in a command shell:

```
use BookstoreDb
```

   If it doesn't already exist, a database named *BookstoreDb* is created. If the database does exist, its connection is opened for transactions.

5. Create a `Books` collection using following command:

```
db.createCollection('Books')
```

   The following result is displayed:

```
{ "ok" : 1 }
```

6. Define a schema for the `Books` collection and insert two documents using the following command:

```
db.Books.insertMany([{'Name':'Design Patterns','Price':54.93,'Category':'Computers','Author':'Ralph
Johnson'}, {'Name':'Clean Code','Price':43.15,'Category':'Computers','Author':'Robert C. Martin'}])
```

   The following result is displayed:

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5bfd996f7b8e48dc15ff215d"),
    ObjectId("5bfd996f7b8e48dc15ff215e")
  ]
}
```

   NOTE

   The ID's shown in this article will not match the IDs when you run this sample.

7. View the documents in the database using the following command:

```
db.Books.find({}).pretty()
```

   The following result is displayed:

```
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215d"),
  "Name" : "Design Patterns",
  "Price" : 54.93,
  "Category" : "Computers",
  "Author" : "Ralph Johnson"
}
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215e"),
  "Name" : "Clean Code",
  "Price" : 43.15,
  "Category" : "Computers",
  "Author" : "Robert C. Martin"
}
```

The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

The database is ready. You can start creating the ASP.NET Core web API.

Create the ASP.NET Core web API project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web Application** project type, and select **Next**.
3. Name the project *BooksApi*, and select **Create**.
4. Select the **.NET Core** target framework and **ASP.NET Core 3.0**. Select the **API** project template, and select **Create**.
5. Visit the NuGet Gallery: MongoDB.Driver to determine the latest stable version of the .NET driver for MongoDB. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

```
Install-Package MongoDB.Driver -Version {VERSION}
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BooksApi.Models
{
    public class Book
    {
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        [BsonElement("Name")]
        public string BookName { get; set; }

        public decimal Price { get; set; }

        public string Category { get; set; }

        public string Author { get; set; }
    }
}
```

In the preceding class, the `Id` property:

- Is required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Is annotated with `[BsonId]` to designate this property as the document's primary key.
- Is annotated with `[BsonRepresentation(BsonType.ObjectId)]` to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the `[BsonElement]` attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to *appsettings.json*:

```
{
  "BookstoreDatabaseSettings": {
    "BooksCollectionName": "Books",
    "ConnectionString": "mongodb://localhost:27017",
    "DatabaseName": "BookstoreDb"
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

2. Add a *BookstoreDatabaseSettings.cs* file to the *Models* directory with the following code:

```
namespace BooksApi.Models
{
    public class BookstoreDatabaseSettings : IBookstoreDatabaseSettings
    {
        public string BooksCollectionName { get; set; }
        public string ConnectionString { get; set; }
        public string DatabaseName { get; set; }
    }

    public interface IBookstoreDatabaseSettings
    {
        string BooksCollectionName { get; set; }
        string ConnectionString { get; set; }
        string DatabaseName { get; set; }
    }
}
```

The preceding `BookstoreDatabaseSettings` class is used to store the *appsettings.json* file's `BookstoreDatabaseSettings` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    // requires using Microsoft.Extensions.Options
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddControllers();
}
```

In the preceding code:

- The configuration instance to which the *appsettings.json* file's `BookstoreDatabaseSettings` section binds is registered in the Dependency Injection (DI) container. For example, a `BookstoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookstoreDatabaseSettings:ConnectionString` property in *appsettings.json*.
- The `IBookstoreDatabaseSettings` interface is registered in DI with a singleton service lifetime. When injected, the interface instance resolves to a `BookstoreDatabaseSettings` object.

4. Add the following code to the top of *Startup.cs* to resolve the `BookstoreDatabaseSettings` and `IBookstoreDatabaseSettings` references:

```
using BooksApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.
2. Add a `BookService` class to the *Services* directory with the following code:

```
using BooksApi.Models;
using MongoDB.Driver;
using System.Collections.Generic;
using System.Linq;

namespace BooksApi.Services
{
    public class BookService
    {
        private readonly IMongoCollection<Book> _books;

        public BookService(IBookstoreDatabaseSettings settings)
        {
            var client = new MongoClient(settings.ConnectionString);
            var database = client.GetDatabase(settings.DatabaseName);

            _books = database.GetCollection<Book>(settings.BooksCollectionName);
        }

        public List<Book> Get() =>
            _books.Find(book => true).ToList();

        public Book Get(string id) =>
            _books.Find<Book>(book => book.Id == id).FirstOrDefault();

        public Book Create(Book book)
        {
            _books.InsertOne(book);
            return book;
        }

        public void Update(string id, Book bookIn) =>
            _books.ReplaceOne(book => book.Id == id, bookIn);

        public void Remove(Book bookIn) =>
            _books.DeleteOne(book => book.Id == bookIn.Id);

        public void Remove(string id) =>
            _books.DeleteOne(book => book.Id == id);
    }
}
```

In the preceding code, an `IBookstoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the *appsettings.json* configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddControllers();
}
```

In the preceding code, the `BookService` class is registered with DI to support constructor injection in consuming classes. The singleton service lifetime is most appropriate because `BookService` takes a direct

dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of *Startup.cs* to resolve the `BookService` reference:

```
using BooksApi.Services;
```

The `BookService` class uses the following `MongoDB.Driver` members to perform CRUD operations against the database:

- MongoClient: Reads the server instance for performing database operations. The constructor of this class is provided the MongoDB connection string:

```
public BookService(IBookstoreDatabaseSettings settings)
{
    var client = new MongoClient(settings.ConnectionString);
    var database = client.GetDatabase(settings.DatabaseName);

    _books = database.GetCollection<Book>(settings.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for performing operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Perform CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOne: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOne: Inserts the provided object as a new document in the collection.
- ReplaceOne: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

```
using BooksApi.Models;
using BooksApi.Services;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace BooksApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly BookService _bookService;

        public BooksController(BookService bookService)
        {
            _bookService = bookService;
        }
```

```
        [HttpGet]
        public ActionResult<List<Book>> Get() =>
            _bookService.Get();

        [HttpGet("{id:length(24)}", Name = "GetBook")]
        public ActionResult<Book> Get(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            return book;
        }

        [HttpPost]
        public ActionResult<Book> Create(Book book)
        {
            _bookService.Create(book);

            return CreatedAtRoute("GetBook", new { id = book.Id.ToString() }, book);
        }

        [HttpPut("{id:length(24)}")]
        public IActionResult Update(string id, Book bookIn)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Update(id, bookIn);

            return NoContent();
        }

        [HttpDelete("{id:length(24)}")]
        public IActionResult Delete(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Remove(book.Id);

            return NoContent();
        }
    }
}
```

The preceding web API controller:

- Uses the `BookService` class to perform CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtRoute in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.
2. Navigate to `http://localhost:<port>/api/books` to test the controller's parameterless `Get` action method. The following JSON response is displayed:

```
[
  {
    "id":"5bfd996f7b8e48dc15ff215d",
    "bookName":"Design Patterns",
    "price":54.93,
    "category":"Computers",
    "author":"Ralph Johnson"
  },
  {
    "id":"5bfd996f7b8e48dc15ff215e",
    "bookName":"Clean Code",
    "price":43.15,
    "category":"Computers",
    "author":"Robert C. Martin"
  }
]
```

3. Navigate to `http://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. The following JSON response is displayed:

```
{
  "id":"{ID}",
  "bookName":"Clean Code",
  "price":43.15,
  "category":"Computers",
  "author":"Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. JSON.NET has been removed from ASP.NET shared framework. Add a package reference to `Microsoft.AspNetCore.Mvc.NewtonsoftJson`.
2. In `Startup.ConfigureServices`, chain the following highlighted code on to the `AddControllers` method call:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddControllers()
        .AddNewtonsoftJson(options => options.UseMemberCasing());
}
```

With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author`.

3. In *Models/Book.cs*, annotate the `BookName` property with the following `[JsonProperty]` attribute:

```
[BsonElement("Name")]
[JsonProperty("Name")]
public string BookName { get; set; }
```

The `[JsonProperty]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

4. Add the following code to the top of *Models/Book.cs* to resolve the `[JsonProperty]` attribute reference:

```
using Newtonsoft.Json;
```

5. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

This tutorial creates a web API that performs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

- Configure MongoDB
- Create a MongoDB database
- Define a MongoDB collection and schema
- Perform MongoDB CRUD operations from a web API
- Customize JSON serialization

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- .NET Core SDK 2.2
- Visual Studio 2019 with the **ASP.NET and web development** workload
- MongoDB

Configure MongoDB

If using Windows, MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `Path` environment variable. This change enables MongoDB access from anywhere on your development machine.

Use the mongo Shell in the following steps to create a database, make collections, and store documents. For more information on mongo Shell commands, see Working with the mongo Shell.

1. Choose a directory on your development machine for storing the data. For example, *C:\BooksData* on Windows. Create the directory if it doesn't exist. The mongo Shell doesn't create new directories.

2. Open a command shell. Run the following command to connect to MongoDB on default port 27017. Remember to replace `<data_directory_path>` with the directory you chose in the previous step.

   ```
   mongod --dbpath <data_directory_path>
   ```

3. Open another command shell instance. Connect to the default test database by running the following command:

   ```
   mongo
   ```

4. Run the following in a command shell:

   ```
   use BookstoreDb
   ```

   If it doesn't already exist, a database named *BookstoreDb* is created. If the database does exist, its connection is opened for transactions.

5. Create a `Books` collection using following command:

   ```
   db.createCollection('Books')
   ```

   The following result is displayed:

   ```
   { "ok" : 1 }
   ```

6. Define a schema for the `Books` collection and insert two documents using the following command:

   ```
   db.Books.insertMany([{'Name':'Design Patterns','Price':54.93,'Category':'Computers','Author':'Ralph
   Johnson'}, {'Name':'Clean Code','Price':43.15,'Category':'Computers','Author':'Robert C. Martin'}])
   ```

   The following result is displayed:

   ```
   {
     "acknowledged" : true,
     "insertedIds" : [
       ObjectId("5bfd996f7b8e48dc15ff215d"),
       ObjectId("5bfd996f7b8e48dc15ff215e")
     ]
   }
   ```

> **NOTE**
> The ID's shown in this article will not match the IDs when you run this sample.

7. View the documents in the database using the following command:

```
db.Books.find({}).pretty()
```

   The following result is displayed:

```
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215d"),
  "Name" : "Design Patterns",
  "Price" : 54.93,
  "Category" : "Computers",
  "Author" : "Ralph Johnson"
}
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215e"),
  "Name" : "Clean Code",
  "Price" : 43.15,
  "Category" : "Computers",
  "Author" : "Robert C. Martin"
}
```

   The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

The database is ready. You can start creating the ASP.NET Core web API.

Create the ASP.NET Core web API project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web Application** project type, and select **Next**.
3. Name the project *BooksApi*, and select **Create**.
4. Select the **.NET Core** target framework and **ASP.NET Core 2.2**. Select the **API** project template, and select **Create**.
5. Visit the NuGet Gallery: MongoDB.Driver to determine the latest stable version of the .NET driver for MongoDB. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

```
Install-Package MongoDB.Driver -Version {VERSION}
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BooksApi.Models
{
    public class Book
    {
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        [BsonElement("Name")]
        public string BookName { get; set; }

        public decimal Price { get; set; }

        public string Category { get; set; }

        public string Author { get; set; }
    }
}
```

In the preceding class, the `Id` property:

- Is required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Is annotated with `[BsonId]` to designate this property as the document's primary key.
- Is annotated with `[BsonRepresentation(BsonType.ObjectId)]` to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the `[BsonElement]` attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to *appsettings.json*:

```
{
  "BookstoreDatabaseSettings": {
    "BooksCollectionName": "Books",
    "ConnectionString": "mongodb://localhost:27017",
    "DatabaseName": "BookstoreDb"
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

2. Add a *BookstoreDatabaseSettings.cs* file to the *Models* directory with the following code:

```
namespace BooksApi.Models
{
    public class BookstoreDatabaseSettings : IBookstoreDatabaseSettings
    {
        public string BooksCollectionName { get; set; }
        public string ConnectionString { get; set; }
        public string DatabaseName { get; set; }
    }

    public interface IBookstoreDatabaseSettings
    {
        string BooksCollectionName { get; set; }
        string ConnectionString { get; set; }
        string DatabaseName { get; set; }
    }
}
```

The preceding `BookstoreDatabaseSettings` class is used to store the *appsettings.json* file's `BookstoreDatabaseSettings` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

In the preceding code:

- The configuration instance to which the *appsettings.json* file's `BookstoreDatabaseSettings` section binds is registered in the Dependency Injection (DI) container. For example, a `BookstoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookstoreDatabaseSettings:ConnectionString` property in *appsettings.json*.
- The `IBookstoreDatabaseSettings` interface is registered in DI with a singleton service lifetime. When injected, the interface instance resolves to a `BookstoreDatabaseSettings` object.

4. Add the following code to the top of *Startup.cs* to resolve the `BookstoreDatabaseSettings` and `IBookstoreDatabaseSettings` references:

```
using BooksApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.
2. Add a `BookService` class to the *Services* directory with the following code:

```
using BooksApi.Models;
using MongoDB.Driver;
using System.Collections.Generic;
using System.Linq;

namespace BooksApi.Services
{
    public class BookService
    {
        private readonly IMongoCollection<Book> _books;

        public BookService(IBookstoreDatabaseSettings settings)
        {
            var client = new MongoClient(settings.ConnectionString);
            var database = client.GetDatabase(settings.DatabaseName);

            _books = database.GetCollection<Book>(settings.BooksCollectionName);
        }

        public List<Book> Get() =>
            _books.Find(book => true).ToList();

        public Book Get(string id) =>
            _books.Find<Book>(book => book.Id == id).FirstOrDefault();

        public Book Create(Book book)
        {
            _books.InsertOne(book);
            return book;
        }

        public void Update(string id, Book bookIn) =>
            _books.ReplaceOne(book => book.Id == id, bookIn);

        public void Remove(Book bookIn) =>
            _books.DeleteOne(book => book.Id == bookIn.Id);

        public void Remove(string id) =>
            _books.DeleteOne(book => book.Id == id);
    }
}
```

In the preceding code, an `IBookstoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the *appsettings.json* configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

In the preceding code, the `BookService` class is registered with DI to support constructor injection in

consuming classes. The singleton service lifetime is most appropriate because `BookService` takes a direct dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of *Startup.cs* to resolve the `BookService` reference:

```
using BooksApi.Services;
```

The `BookService` class uses the following `MongoDB.Driver` members to perform CRUD operations against the database:

- MongoClient: Reads the server instance for performing database operations. The constructor of this class is provided the MongoDB connection string:

```
public BookService(IBookstoreDatabaseSettings settings)
{
    var client = new MongoClient(settings.ConnectionString);
    var database = client.GetDatabase(settings.DatabaseName);

    _books = database.GetCollection<Book>(settings.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for performing operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Perform CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOne: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOne: Inserts the provided object as a new document in the collection.
- ReplaceOne: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

```
using BooksApi.Models;
using BooksApi.Services;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace BooksApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly BookService _bookService;

        public BooksController(BookService bookService)
        {
            _bookService = bookService;
```

```
        }

        [HttpGet]
        public ActionResult<List<Book>> Get() =>
            _bookService.Get();

        [HttpGet("{id:length(24)}", Name = "GetBook")]
        public ActionResult<Book> Get(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            return book;
        }

        [HttpPost]
        public ActionResult<Book> Create(Book book)
        {
            _bookService.Create(book);

            return CreatedAtRoute("GetBook", new { id = book.Id.ToString() }, book);
        }

        [HttpPut("{id:length(24)}")]
        public IActionResult Update(string id, Book bookIn)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Update(id, bookIn);

            return NoContent();
        }

        [HttpDelete("{id:length(24)}")]
        public IActionResult Delete(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Remove(book.Id);

            return NoContent();
        }
    }
}
```

The preceding web API controller:

- Uses the `BookService` class to perform CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtRoute in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also

adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.
2. Navigate to `http://localhost:<port>/api/books` to test the controller's parameterless `Get` action method. The following JSON response is displayed:

```
[
  {
    "id":"5bfd996f7b8e48dc15ff215d",
    "bookName":"Design Patterns",
    "price":54.93,
    "category":"Computers",
    "author":"Ralph Johnson"
  },
  {
    "id":"5bfd996f7b8e48dc15ff215e",
    "bookName":"Clean Code",
    "price":43.15,
    "category":"Computers",
    "author":"Robert C. Martin"
  }
]
```

3. Navigate to `http://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. The following JSON response is displayed:

```
{
  "id":"{ID}",
  "bookName":"Clean Code",
  "price":43.15,
  "category":"Computers",
  "author":"Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. In `Startup.ConfigureServices`, chain the following highlighted code on to the `AddMvc` method call:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddMvc()
            .AddJsonOptions(options => options.UseMemberCasing())
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author`.

2. In *Models/Book.cs*, annotate the `BookName` property with the following `[JsonProperty]` attribute:

```
[BsonElement("Name")]
[JsonProperty("Name")]
public string BookName { get; set; }
```

The `[JsonProperty]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

3. Add the following code to the top of *Models/Book.cs* to resolve the `[JsonProperty]` attribute reference:

```
using Newtonsoft.Json;
```

4. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- IdentityServer4

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see Welcome to IdentityServer4.

Next steps

For more information on building ASP.NET Core web APIs, see the following resources:

- YouTube version of this article
- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API

Tutorial: Call an ASP.NET Core web API with JavaScript

9/22/2020 • 5 minutes to read • Edit Online

By Rick Anderson

This tutorial shows how to call an ASP.NET Core web API with JavaScript, using the Fetch API.

For ASP.NET Core 2.2, see the 2.2 version of Call the web API with JavaScript.

Prerequisites

- Complete Tutorial: Create a web API
- Familiarity with CSS, HTML, and JavaScript

Call the web API with JavaScript

In this section, you'll add an HTML page containing forms for creating and managing to-do items. Event handlers are attached to elements on the page. The event handlers result in HTTP requests to the web API's action methods. The Fetch API's `fetch` function initiates each HTTP request.

The `fetch` function returns a Promise object, which contains an HTTP response represented as a `Response` object. A common pattern is to extract the JSON response body by invoking the `json` function on the `Response` object. JavaScript updates the page with the details from the web API's response.

The simplest `fetch` call accepts a single parameter representing the route. A second parameter, known as the `init` object, is optional. `init` is used to configure the HTTP request.

1. Configure the app to serve static files and enable default file mapping. The following highlighted code is needed in the `Configure` method of *Startup.cs*:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

2. Create a *wwwroot* folder in the project root.

3. Create a *js* folder inside of the *wwwroot* folder.

4. Add an HTML file named *index.html* to the *wwwroot* folder. Replace the contents of *index.html* with the following markup:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <link rel="stylesheet" href="css/site.css" />
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="editForm">
        <h3>Edit</h3>
        <form action="javascript:void(0);" onsubmit="updateItem()">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete?</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="js/site.js" asp-append-version="true"></script>
    <script type="text/javascript">
        getItems();
    </script>
</body>
</html>
```

5. Add a JavaScript file named *site.js* to the *wwwroot/js* folder. Replace the contents of *site.js* with the following code:

```
const uri = 'api/TodoItems';
let todos = [];

function getItems() {
  fetch(uri)
    .then(response => response.json())
    .then(data => _displayItems(data))
    .catch(error => console.error('Unable to get items.', error));
}

function addItem() {
  const addNameTextbox = document.getElementById('add-name');
```

```
  const item = {
    isComplete: false,
    name: addNameTextbox.value.trim()
  };

  fetch(uri, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
    .then(response => response.json())
    .then(() => {
      getItems();
      addNameTextbox.value = '';
    })
    .catch(error => console.error('Unable to add item.', error));
}

function deleteItem(id) {
  fetch(`${uri}/${id}`, {
    method: 'DELETE'
  })
  .then(() => getItems())
  .catch(error => console.error('Unable to delete item.', error));
}

function displayEditForm(id) {
  const item = todos.find(item => item.id === id);

  document.getElementById('edit-name').value = item.name;
  document.getElementById('edit-id').value = item.id;
  document.getElementById('edit-isComplete').checked = item.isComplete;
  document.getElementById('editForm').style.display = 'block';
}

function updateItem() {
  const itemId = document.getElementById('edit-id').value;
  const item = {
    id: parseInt(itemId, 10),
    isComplete: document.getElementById('edit-isComplete').checked,
    name: document.getElementById('edit-name').value.trim()
  };

  fetch(`${uri}/${itemId}`, {
    method: 'PUT',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
  .then(() => getItems())
  .catch(error => console.error('Unable to update item.', error));

  closeInput();

  return false;
}

function closeInput() {
  document.getElementById('editForm').style.display = 'none';
}

function _displayCount(itemCount) {
  const name = (itemCount === 1) ? 'to-do' : 'to-dos';
```

```
  document.getElementById('counter').innerText = `${itemCount} ${name}`;
}

function _displayItems(data) {
  const tBody = document.getElementById('todos');
  tBody.innerHTML = '';

  _displayCount(data.length);

  const button = document.createElement('button');

  data.forEach(item => {
    let isCompleteCheckbox = document.createElement('input');
    isCompleteCheckbox.type = 'checkbox';
    isCompleteCheckbox.disabled = true;
    isCompleteCheckbox.checked = item.isComplete;

    let editButton = button.cloneNode(false);
    editButton.innerText = 'Edit';
    editButton.setAttribute('onclick', `displayEditForm(${item.id})`);

    let deleteButton = button.cloneNode(false);
    deleteButton.innerText = 'Delete';
    deleteButton.setAttribute('onclick', `deleteItem(${item.id})`);

    let tr = tBody.insertRow();

    let td1 = tr.insertCell(0);
    td1.appendChild(isCompleteCheckbox);

    let td2 = tr.insertCell(1);
    let textNode = document.createTextNode(item.name);
    td2.appendChild(textNode);

    let td3 = tr.insertCell(2);
    td3.appendChild(editButton);

    let td4 = tr.insertCell(3);
    td4.appendChild(deleteButton);
  });

  todos = data;
}
```

A change to the ASP.NET Core project's launch settings may be required to test the HTML page locally:

1. Open *Properties\launchSettings.json*.
2. Remove the `launchUrl` property to force the app to open at *index.html*—the project's default file.

This sample calls all of the CRUD methods of the web API. Following are explanations of the web API requests.

Get a list of to-do items

In the following code, an HTTP GET request is sent to the *api/TodoItems* route:

```
fetch(uri)
  .then(response => response.json())
  .then(data => _displayItems(data))
  .catch(error => console.error('Unable to get items.', error));
```

When the web API returns a successful status code, the `_displayItems` function is invoked. Each to-do item in the array parameter accepted by `_displayItems` is added to a table with **Edit** and **Delete** buttons. If the web API request fails, an error is logged to the browser's console.

Add a to-do item

In the following code:

- An `item` variable is declared to construct an object literal representation of the to-do item.
- A Fetch request is configured with the following options:
  - `method` —specifies the POST HTTP action verb.
  - `body` —specifies the JSON representation of the request body. The JSON is produced by passing the object literal stored in `item` to the JSON.stringify function.
  - `headers` —specifies the `Accept` and `Content-Type` HTTP request headers. Both headers are set to `application/json` to specify the media type being received and sent, respectively.
- An HTTP POST request is sent to the *api/TodoItems* route.

```
function addItem() {
  const addNameTextbox = document.getElementById('add-name');

  const item = {
    isComplete: false,
    name: addNameTextbox.value.trim()
  };

  fetch(uri, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
    .then(response => response.json())
    .then(() => {
      getItems();
      addNameTextbox.value = '';
    })
    .catch(error => console.error('Unable to add item.', error));
}
```

When the web API returns a successful status code, the `getItems` function is invoked to update the HTML table. If the web API request fails, an error is logged to the browser's console.

Update a to-do item

Updating a to-do item is similar to adding one; however, there are two significant differences:

- The route is suffixed with the unique identifier of the item to update. For example, *api/TodoItems/1*.
- The HTTP action verb is PUT, as indicated by the `method` option.

```
fetch(`${uri}/${itemId}`, {
  method: 'PUT',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(item)
})
.then(() => getItems())
.catch(error => console.error('Unable to update item.', error));
```

Delete a to-do item

To delete a to-do item, set the request's `method` option to `DELETE` and specify the item's unique identifier in the

URL.

```
fetch(`${uri}/${id}`, {
  method: 'DELETE'
})
.then(() => getItems())
.catch(error => console.error('Unable to delete item.', error));
```

Advance to the next tutorial to learn how to generate web API help pages:

Get started with Swashbuckle and ASP.NET Core

Create backend services for native mobile apps with ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

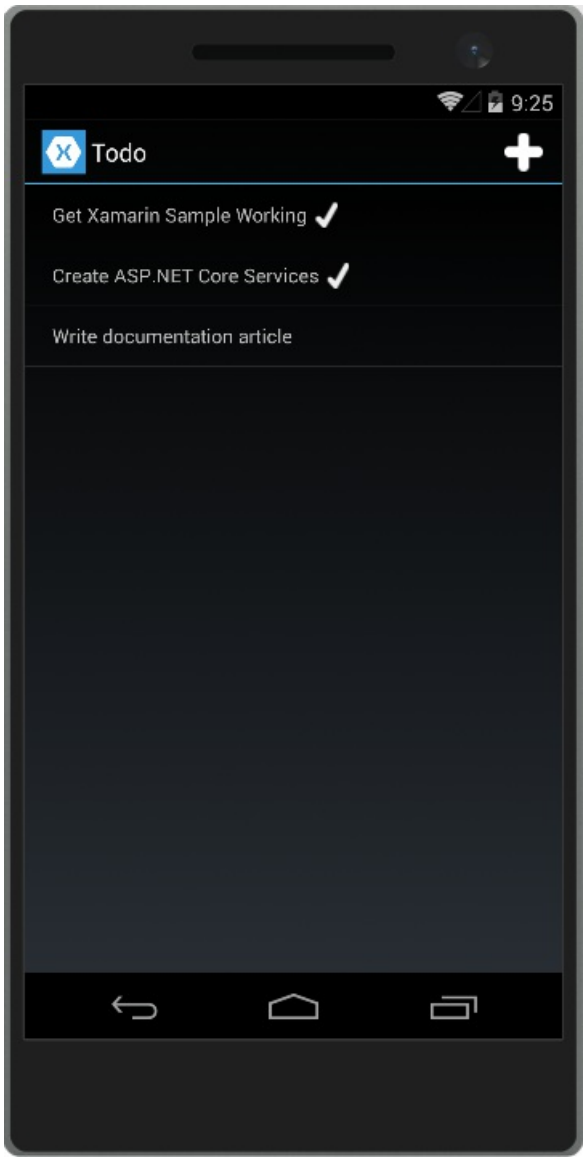
By Steve Smith

Mobile apps can communicate with ASP.NET Core backend services. For instructions on connecting local web services from iOS simulators and Android emulators, see Connect to Local Web Services from iOS Simulators and Android Emulators.

View or download sample backend services code

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core MVC to support native mobile apps. It uses the Xamarin Forms ToDoRest app as its native client, which includes separate native clients for Android, iOS, Windows Universal, and Window Phone devices. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), as well as download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Web API 2 services project, which this article's ASP.NET Core app replaces (with no changes required by the client).

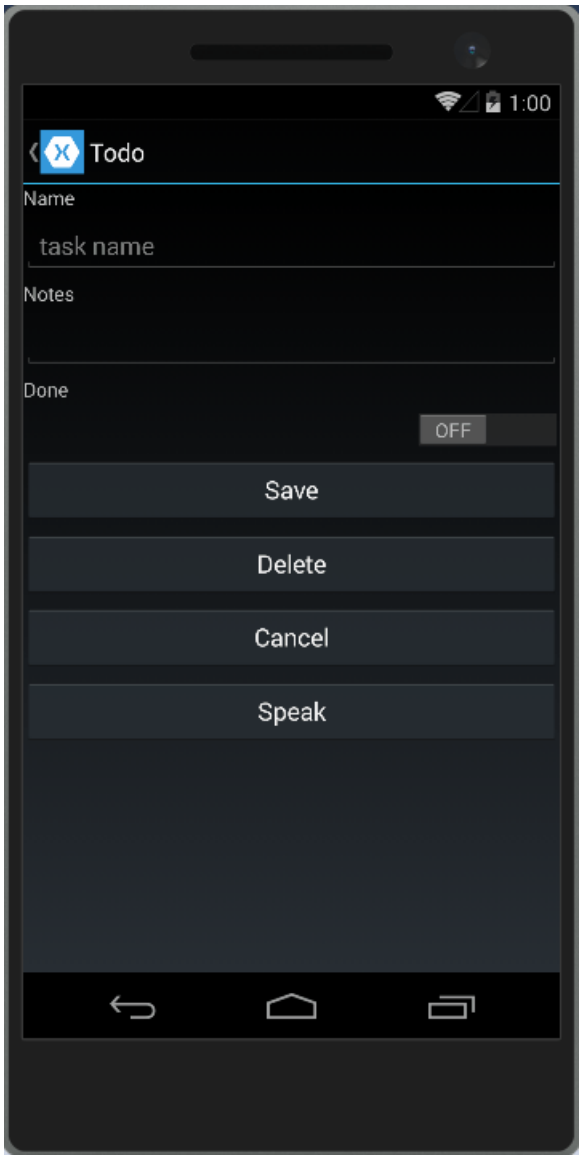


Features

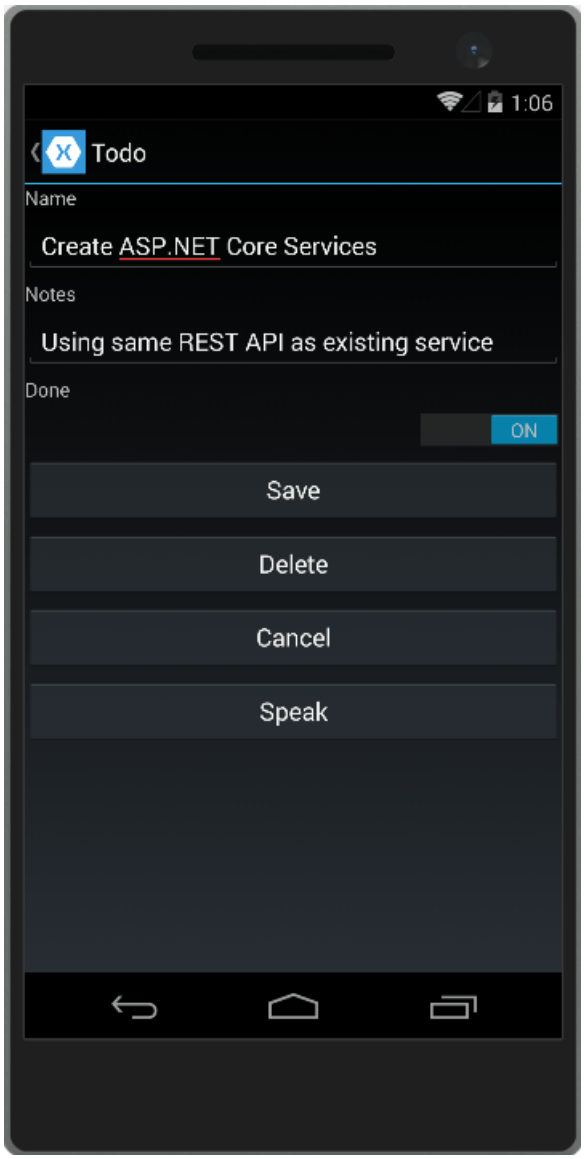
The ToDoRest app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

The main view of the items, as shown above, lists each item's name and indicates if it's done with a checkmark.

Tapping the `+` icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:



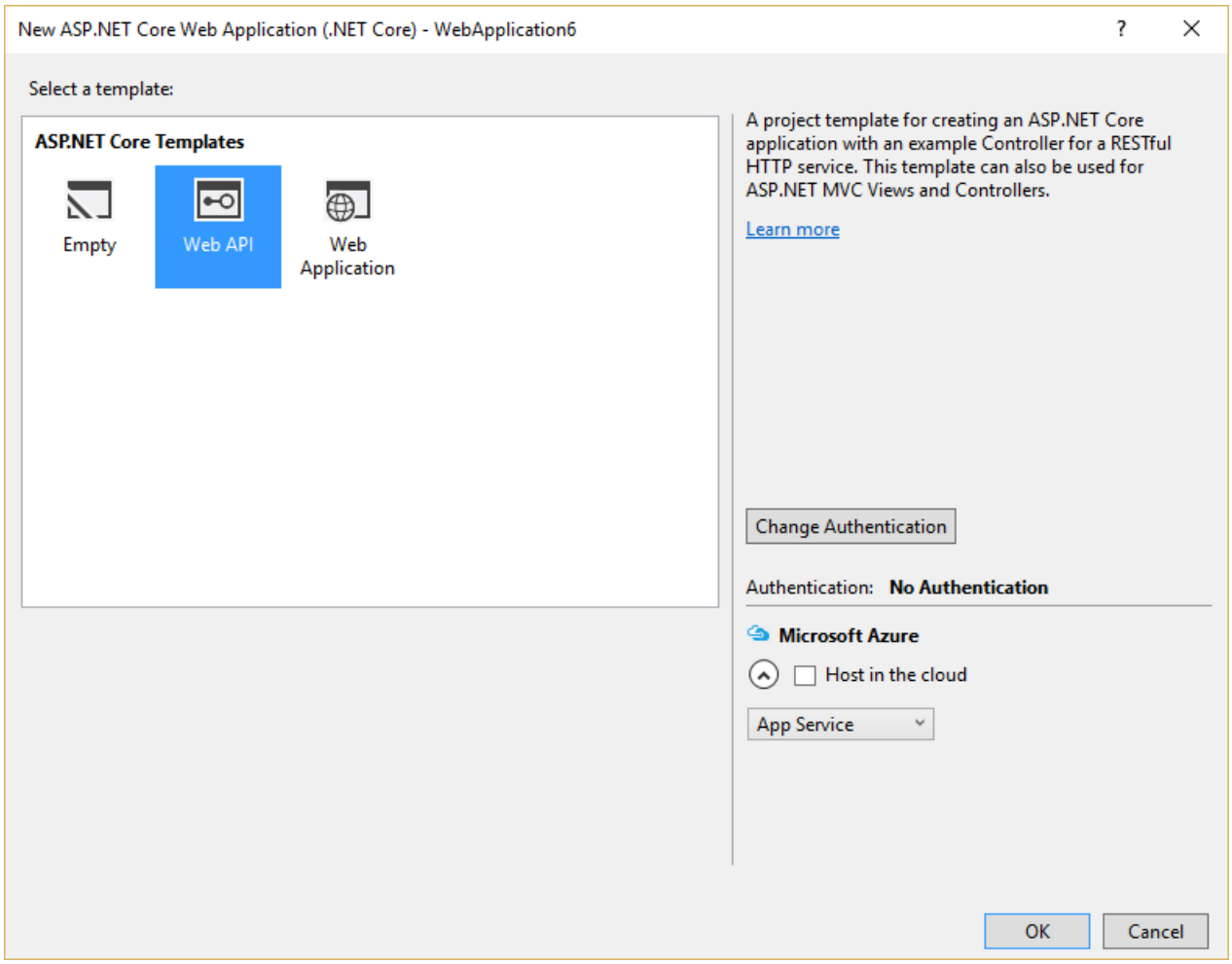
This sample is configured by default to use backend services hosted at developer.xamarin.com, which allow read-only operations. To test it out yourself against the ASP.NET Core app created in the next section running on your computer, you'll need to update the app's `RestUrl` constant. Navigate to the `ToDoREST` project and open the *Constants.cs* file. Replace the `RestUrl` with a URL that includes your machine's IP address (not localhost or 127.0.0.1, since this address is used from the device emulator, not from your machine). Include the port number as well (5000). In order to test that your services work with a device, ensure you don't have an active firewall blocking access to this port.

```
// URL of REST service (Xamarin ReadOnly Service)
//public static string RestUrl = "http://developer.xamarin.com:8081/api/todoitems{0}";

// use your machine's IP address
public static string RestUrl = "http://192.168.1.207:5000/api/todoitems/{0}";
```

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template and No Authentication. Name the project *ToDoApi*.



The application should respond to all requests made to port 5000. Update *Program.cs* to include `.UseUrls("http://*:5000")` to achieve this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:5000")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

NOTE

Make sure you run the application directly, rather than behind IIS Express, which ignores non-local requests by default. Run dotnet run from a command prompt, or choose the application name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields with the `[Required]` attribute:

```
using System.ComponentModel.DataAnnotations;

namespace ToDoApi.Models
{
    public class ToDoItem
    {
        [Required]
        public string ID { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        public string Notes { get; set; }

        public bool Done { get; set; }
    }
}
```

The API methods require some way to work with data. Use the same `IToDoRepository` interface the original Xamarin sample uses:

```
using System.Collections.Generic;
using ToDoApi.Models;

namespace ToDoApi.Interfaces
{
    public interface IToDoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<ToDoItem> All { get; }
        ToDoItem Find(string id);
        void Insert(ToDoItem item);
        void Update(ToDoItem item);
        void Delete(string id);
    }
}
```

For this sample, the implementation just uses a private collection of items:

```
using System.Collections.Generic;
using System.Linq;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Services
{
    public class ToDoRepository : IToDoRepository
    {
        private List<ToDoItem> _toDoList;

        public ToDoRepository()
        {
            InitializeData();
        }

        public IEnumerable<ToDoItem> All
        {
            get { return _toDoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _toDoList.Any(item => item.ID == id);
```

```
        }

        public ToDoItem Find(string id)
        {
            return _toDoList.FirstOrDefault(item => item.ID == id);
        }

        public void Insert(ToDoItem item)
        {
            _toDoList.Add(item);
        }

        public void Update(ToDoItem item)
        {
            var todoItem = this.Find(item.ID);
            var index = _toDoList.IndexOf(todoItem);
            _toDoList.RemoveAt(index);
            _toDoList.Insert(index, item);
        }

        public void Delete(string id)
        {
            _toDoList.Remove(this.Find(id));
        }

        private void InitializeData()
        {
            _toDoList = new List<ToDoItem>();

            var todoItem1 = new ToDoItem
            {
                ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
                Name = "Learn app development",
                Notes = "Attend Xamarin University",
                Done = true
            };

            var todoItem2 = new ToDoItem
            {
                ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
                Name = "Develop apps",
                Notes = "Use Xamarin Studio/Visual Studio",
                Done = false
            };

            var todoItem3 = new ToDoItem
            {
                ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
                Name = "Publish apps",
                Notes = "All app stores",
                Done = false,
            };

            _toDoList.Add(todoItem1);
            _toDoList.Add(todoItem2);
            _toDoList.Add(todoItem3);
        }
    }
}
```

Configure the implementation in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();

    services.AddSingleton<IToDoRepository,ToDoRepository>();
}
```

At this point, you're ready to create the *ToDoItemsController*.

TIP

Learn more about creating web APIs in Build your first Web API with ASP.NET Core MVC and Visual Studio.

Creating the Controller

Add a new controller to the project, *ToDoItemsController*. It should inherit from Microsoft.AspNetCore.Mvc.Controller. Add a `Route` attribute to indicate that the controller will handle requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about routing.

The controller requires an `IToDoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance will be provided using the framework's support for dependency injection.

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using ToDoApi.Interfaces;
using ToDoApi.Models;

namespace ToDoApi.Controllers
{
    [Route("api/[controller]")]
    public class ToDoItemsController : Controller
    {
        private readonly IToDoRepository _toDoRepository;

        public ToDoItemsController(IToDoRepository toDoRepository)
        {
            _toDoRepository = toDoRepository;
        }
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP GET request.

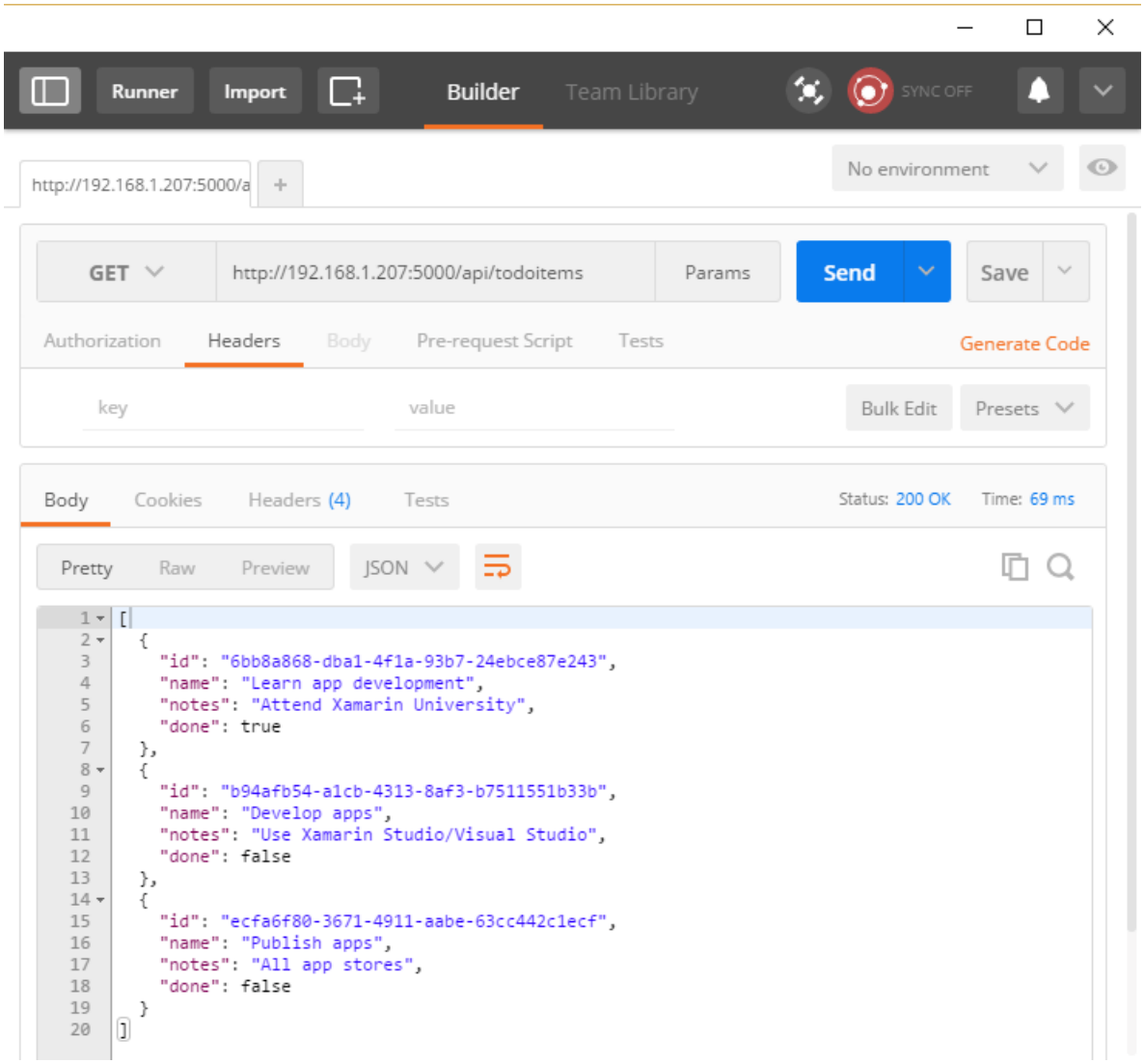
Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

```
[HttpGet]
public IActionResult List()
{
    return Ok(_toDoRepository.All);
}
```

The `List` method returns a 200 OK response code and all of the ToDo items, serialized as JSON.

You can test your new API method using a variety of tools, such as Postman, shown here:



Creating Items

By convention, creating new data items is mapped to the HTTP POST verb. The `Create` method has an `[HttpPost]` attribute applied to it and accepts a `ToDoItem` instance. Since the `item` argument is passed in the body of the POST, this parameter specifies the `[FromBody]` attribute.

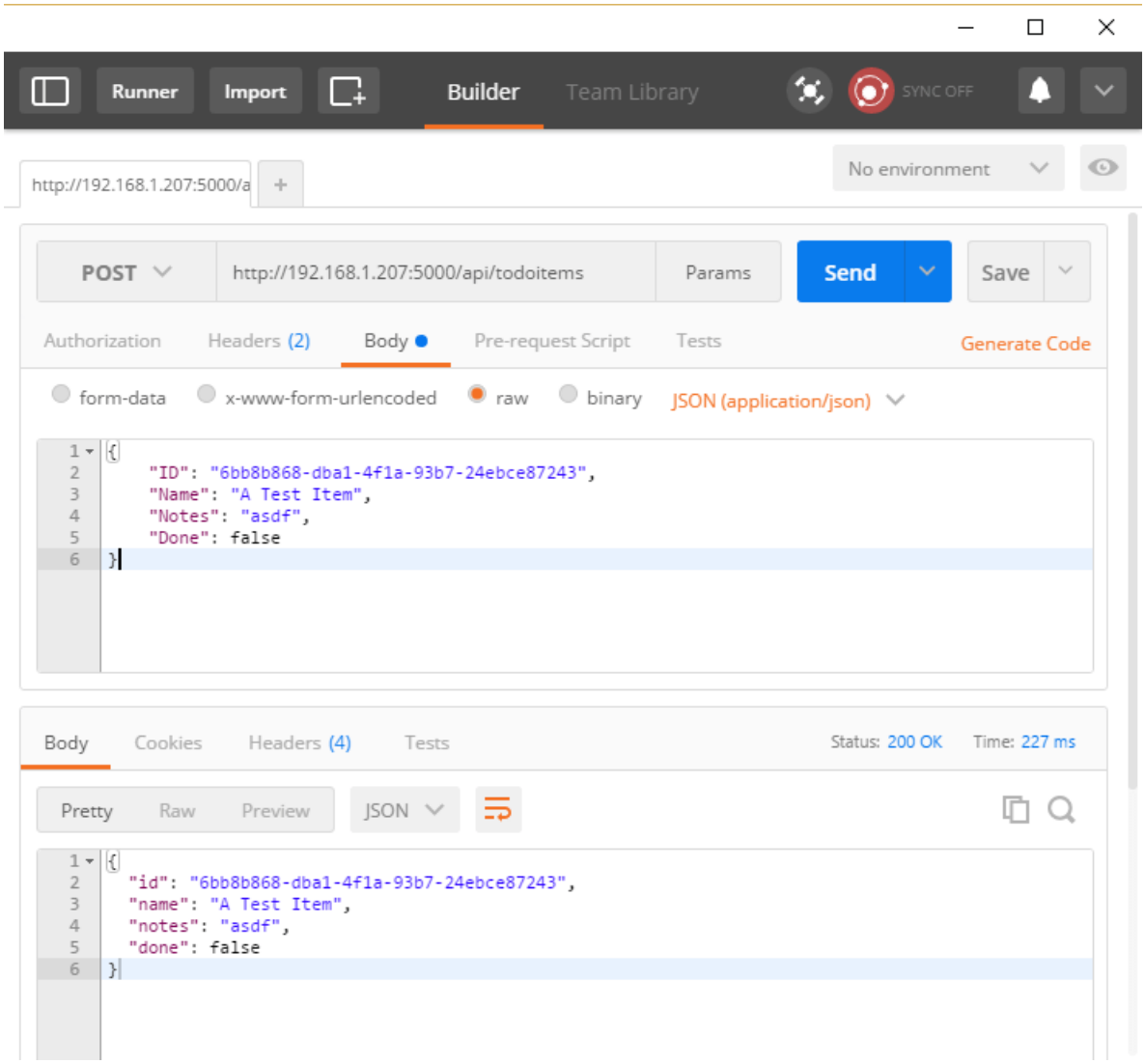
Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it's added using the repository. Checking `ModelState.IsValid` performs model validation, and should be done in every API method that accepts user input.

```
[HttpPost]
public IActionResult Create([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _toDoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict, ErrorCode.TodoItemIDInUse.ToString());
        }
        _toDoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}
```

The sample uses an enum containing error codes that are passed to the mobile client:

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

Test adding new items using Postman by choosing the POST verb providing the new object in JSON format in the Body of the request. You should also add a request header specifying a `Content-Type` of `application/json`.



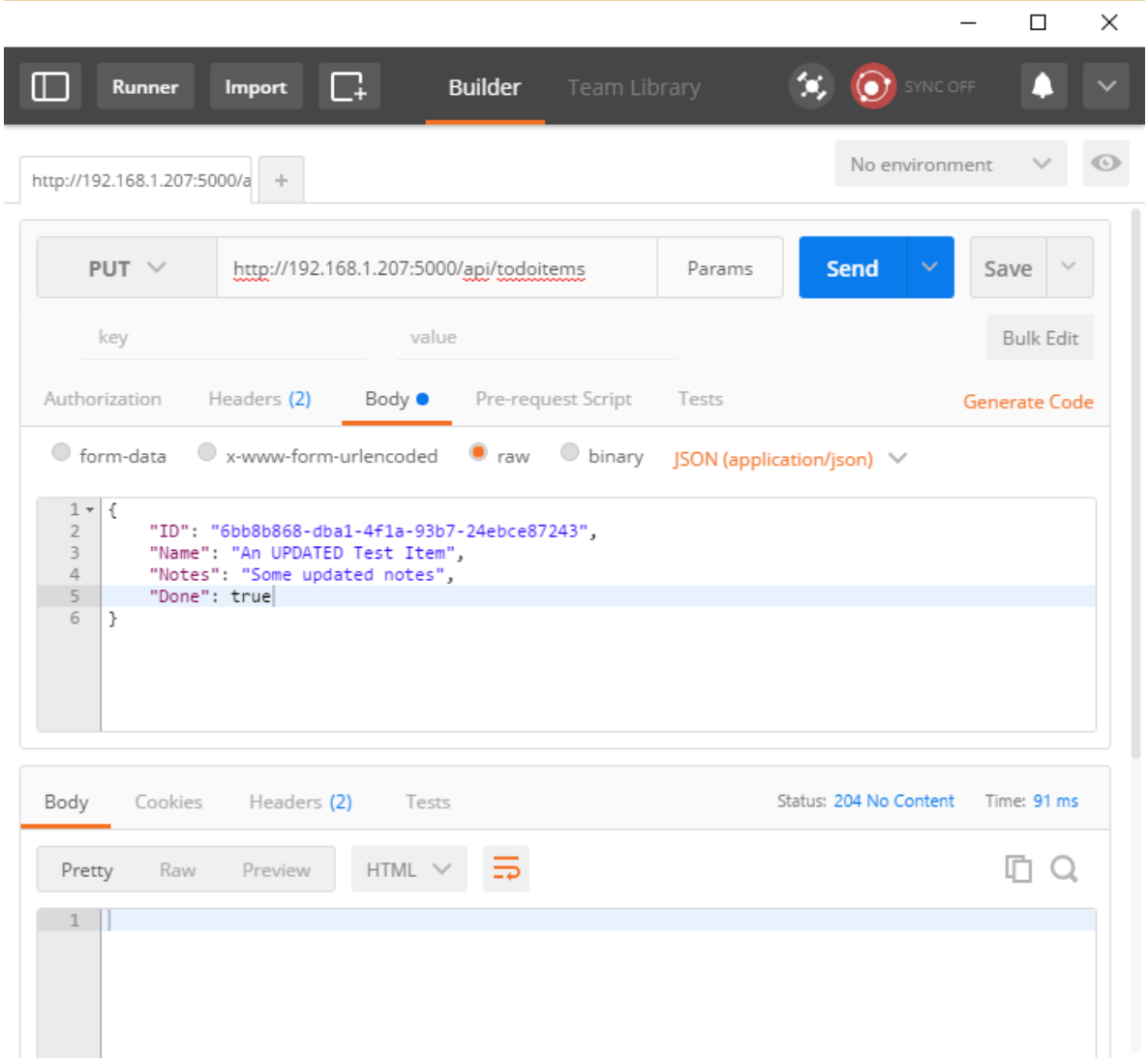
The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP PUT requests. Other than this change, the `Edit` method is almost identical to `Create`. Note that if the record isn't found, the `Edit` action will return a `NotFound` (404) response.

```
[HttpPut]
public IActionResult Edit([FromBody] ToDoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _toDoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _toDoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with Postman, change the verb to PUT. Specify the updated object data in the Body of the request.



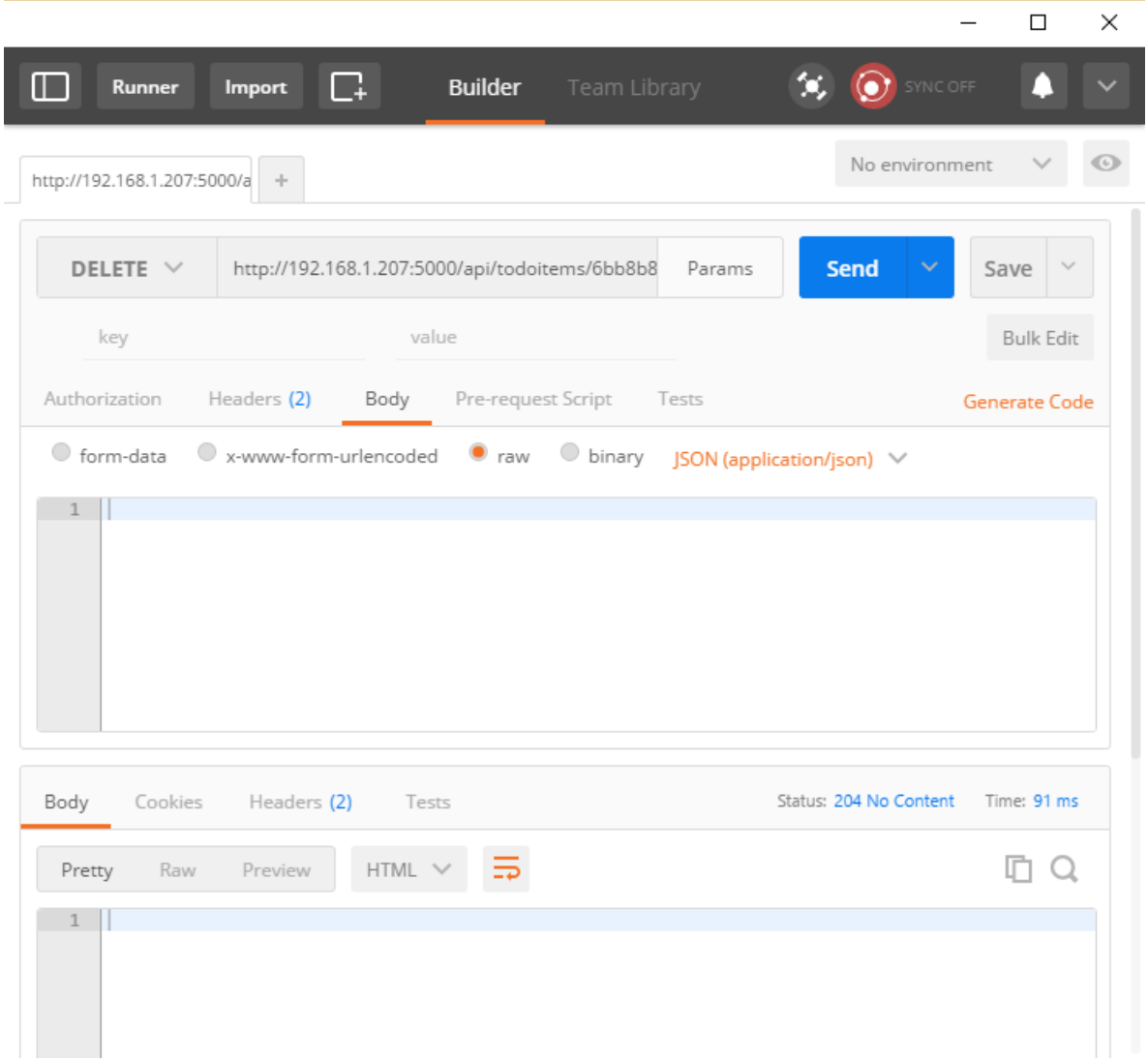
This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making DELETE requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist will receive `NotFound` responses. Otherwise, a successful request will get a `NoContent` (204) response.

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _toDoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _toDoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

Note that when testing the delete functionality, nothing is required in the Body of the request.



Common Web API Conventions

As you develop the backend services for your app, you will want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown above, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a filter. Learn more about how to encapsulate common API policies in ASP.NET Core MVC applications.

Additional resources

- Authentication and Authorization

Publish an ASP.NET Core web API to Azure API Management with Visual Studio

9/22/2020 • 4 minutes to read • Edit Online

By Matt Soucoup

Set up

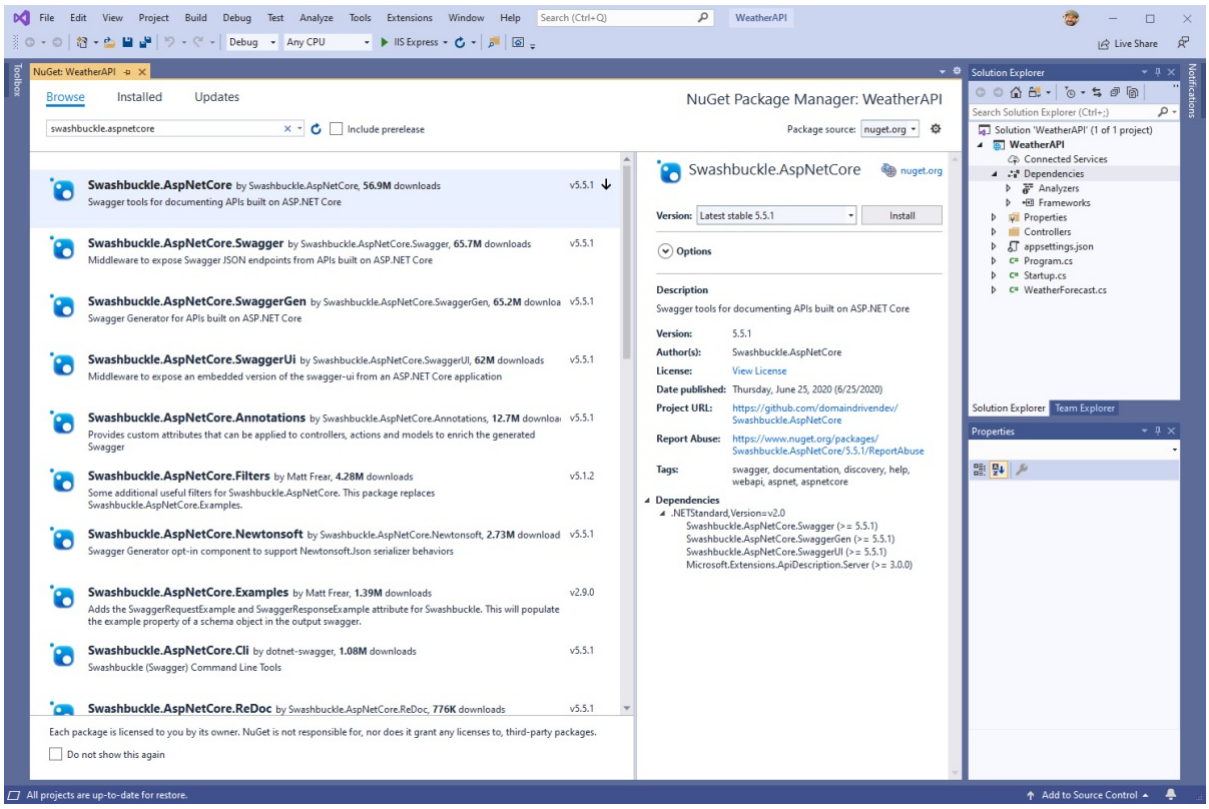
- Open a free Azure account if you don't have one.
- Create a new Azure API Management instance if you haven't already.
- Create a web API app project.

Configure the app

Adding Swagger definitions to the ASP.NET Core web API allows Azure API Management to read the app's API definitions. Complete the following steps.

Add Swagger

1. Add the Swashbuckle.AspNetCore NuGet package to the ASP.NET Core web API project:



2. Add the following line to the `Startup.ConfigureServices` method:

```
services.AddSwaggerGen();
```

3. Add the following line to the `Startup.Configure` method:

```
app.UseSwagger();
```

Change the API routing

Next, you'll change the URL structure needed to access the `Get` action of the `WeatherForecastController`. Complete the following steps:

1. Open the *WeatherForecastController.cs* file.
2. Delete the `[Route("[controller]")]` class-level attribute. The class definition will look like the following:

```
[ApiController]
public class WeatherForecastController : ControllerBase
```

3. Add a `[Route("/")]` attribute to the `Get()` action. The function definition will look like the following:

```
[HttpGet]
[Route("/")]
public IEnumerable<WeatherForecast> Get()
```

Publish the web API to Azure App Service

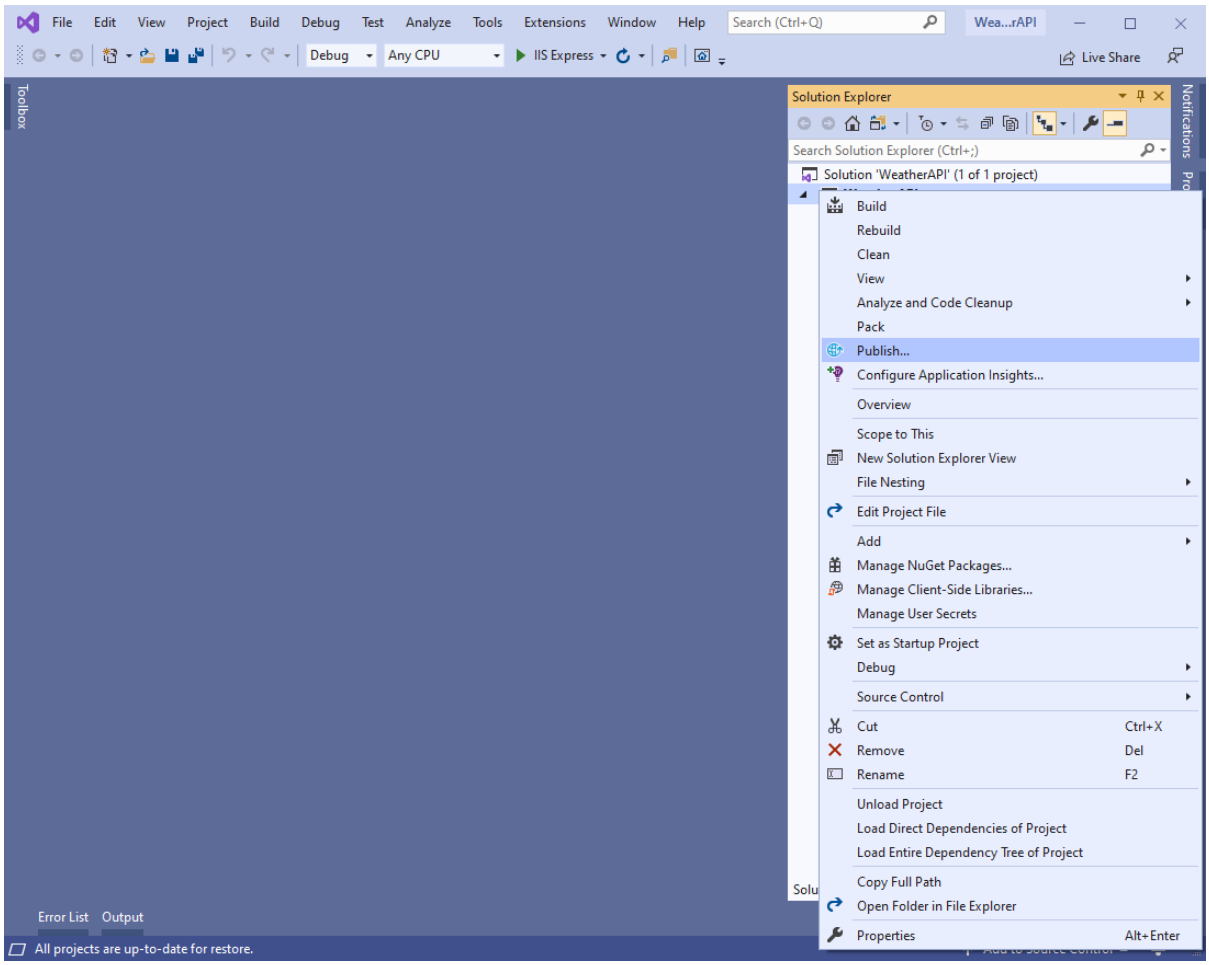
Complete the following steps to publish the ASP.NET Core web API to Azure API Management:

1. Publish the API app to Azure App Service.
2. Add a blank API to the Azure API Management service instance.
3. Publish the ASP.NET Core web API app to the Azure API Management service instance.

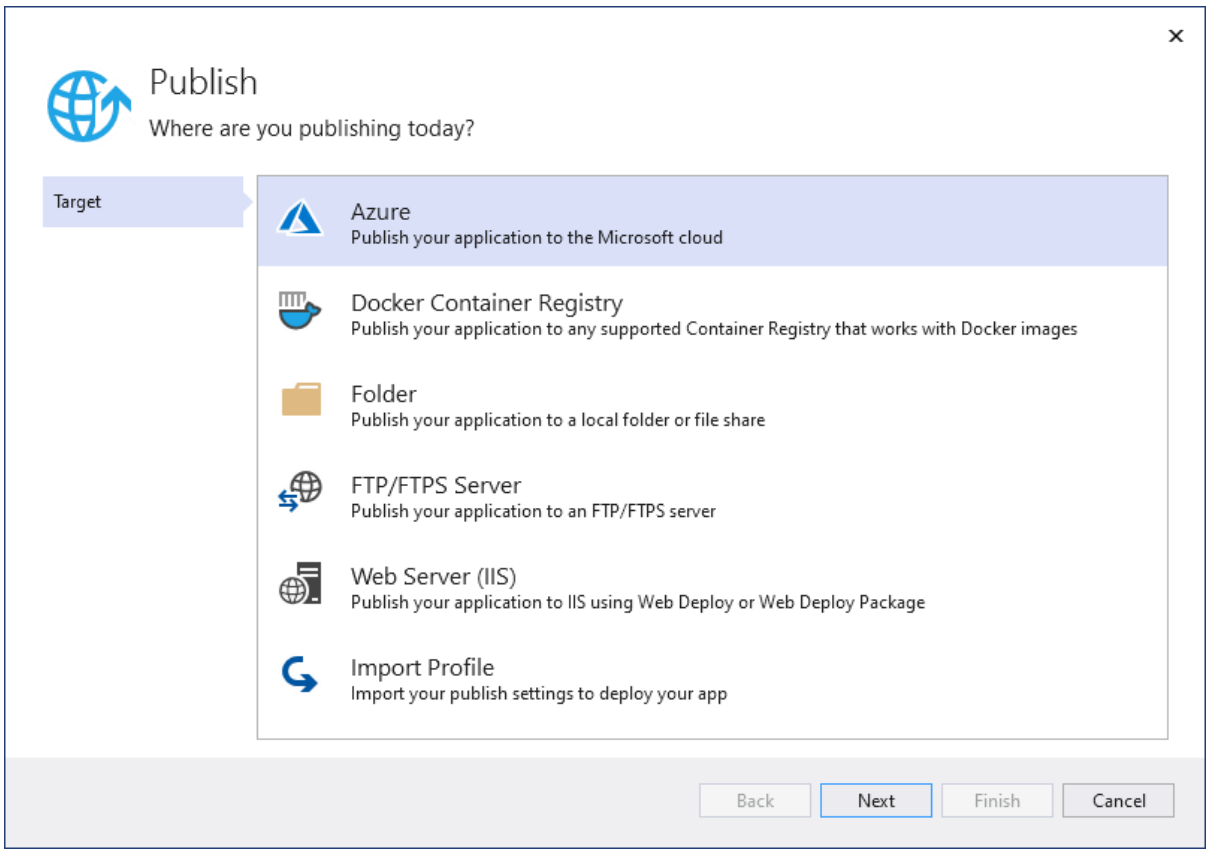
Publish the API app to Azure App Service

Complete the following steps to publish the ASP.NET Core web API to Azure API Management:

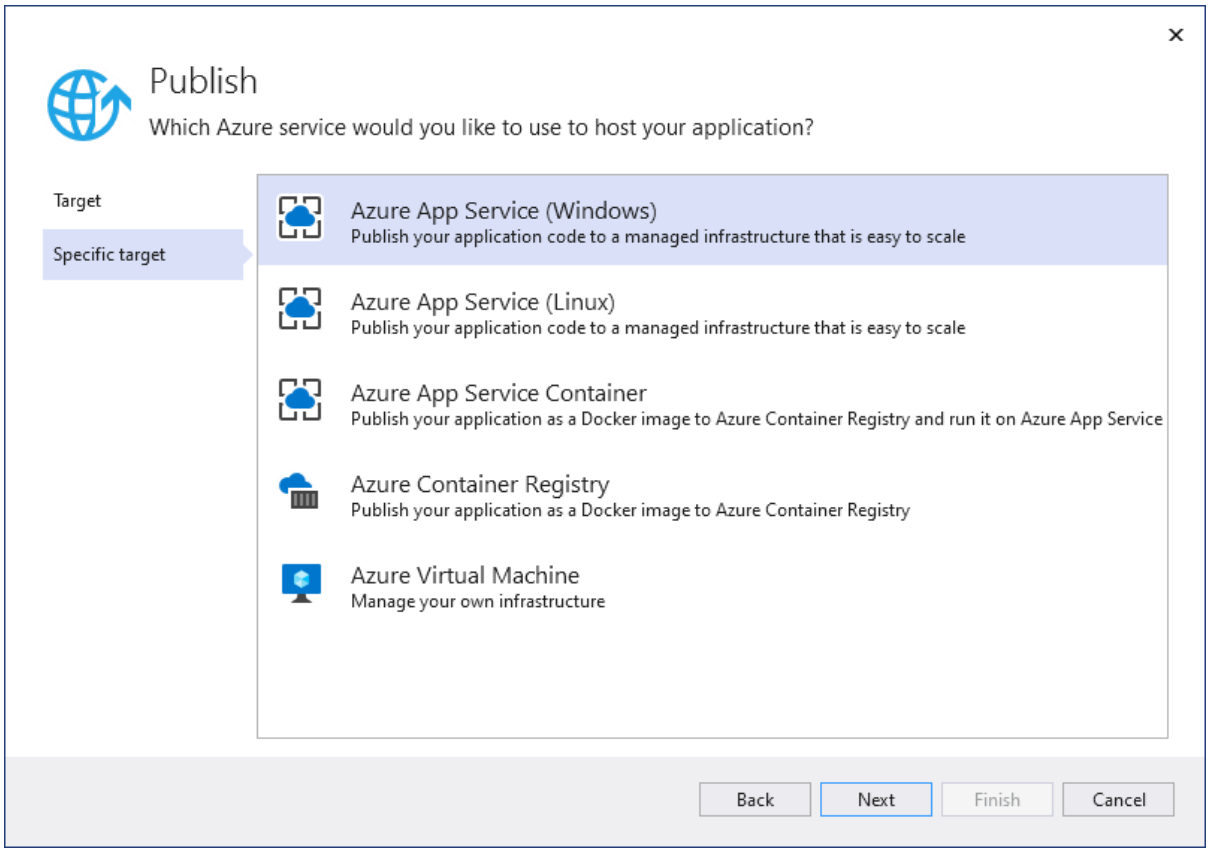
1. In **Solution Explorer**, right-click the project and select **Publish**:



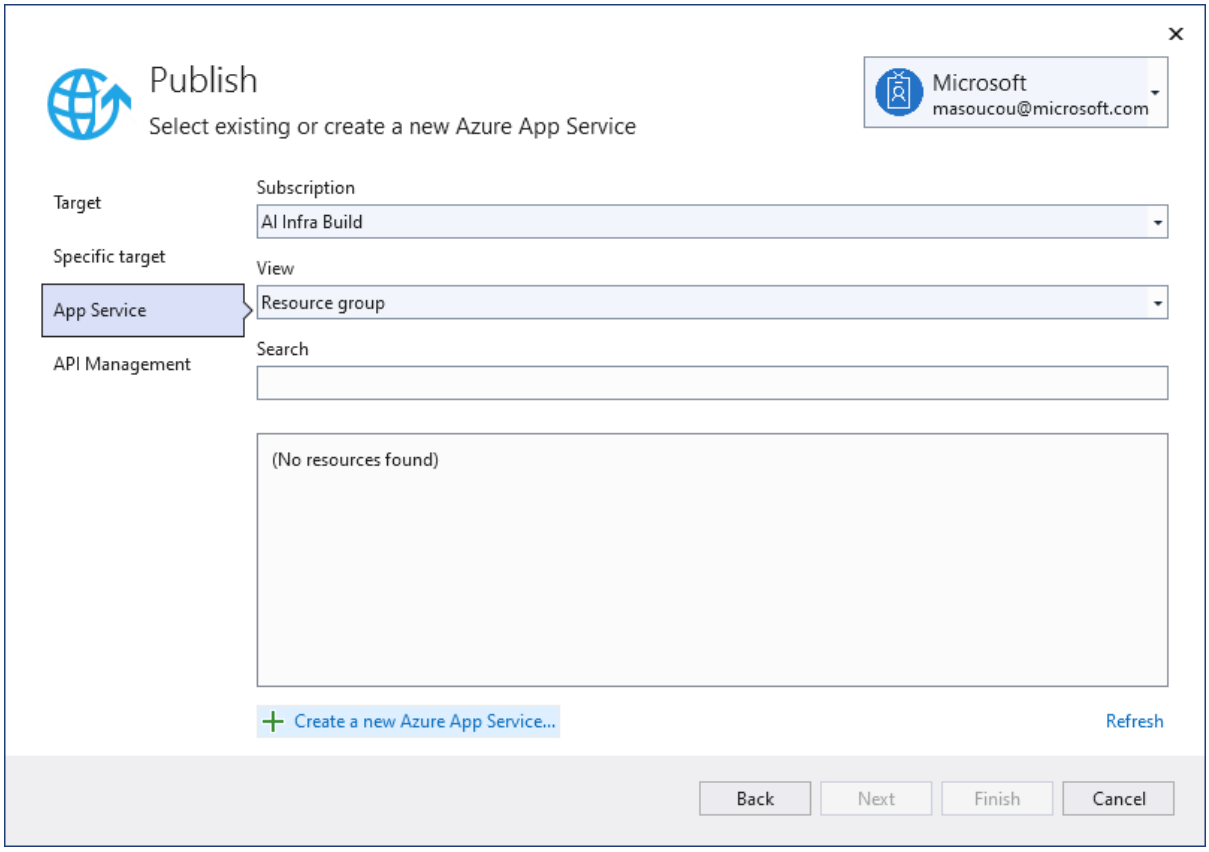
2. In the **Publish** dialog, select **Azure** and select the **Next** button:



3. Select **Azure App Service (Windows)** and select the **Next** button:

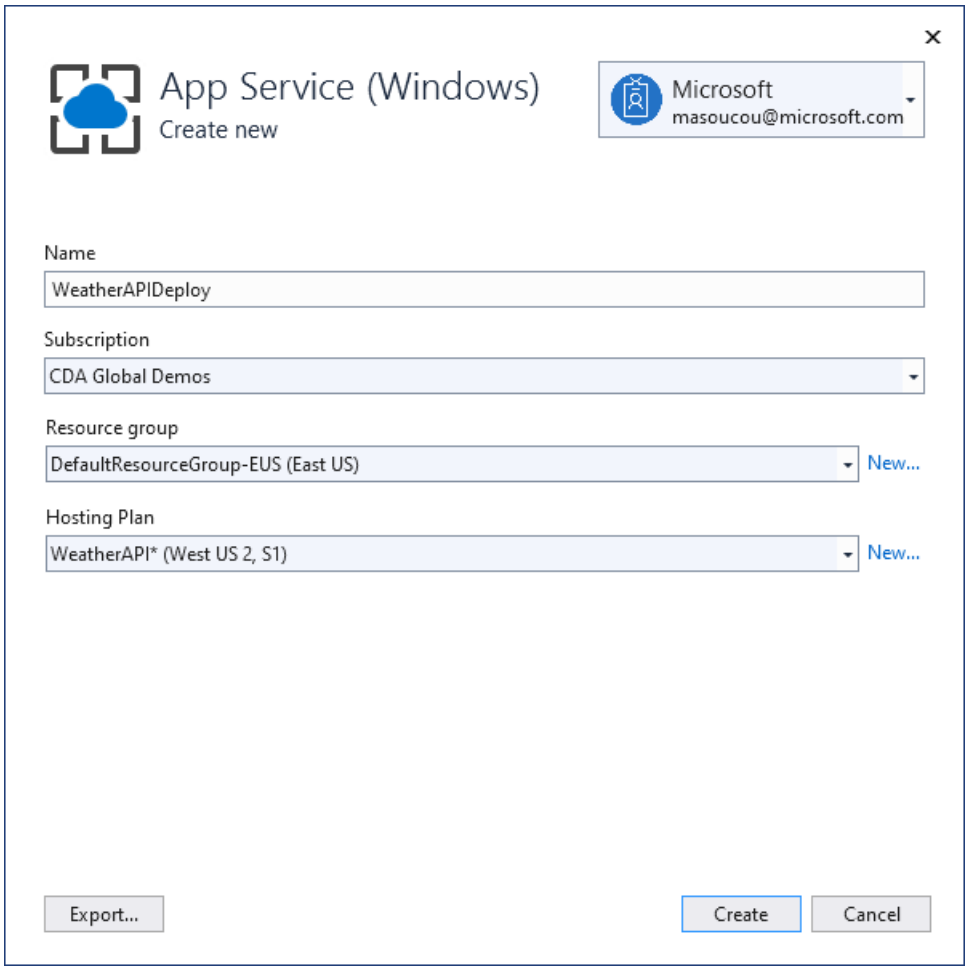


4. Select **Create a new Azure App Service**.

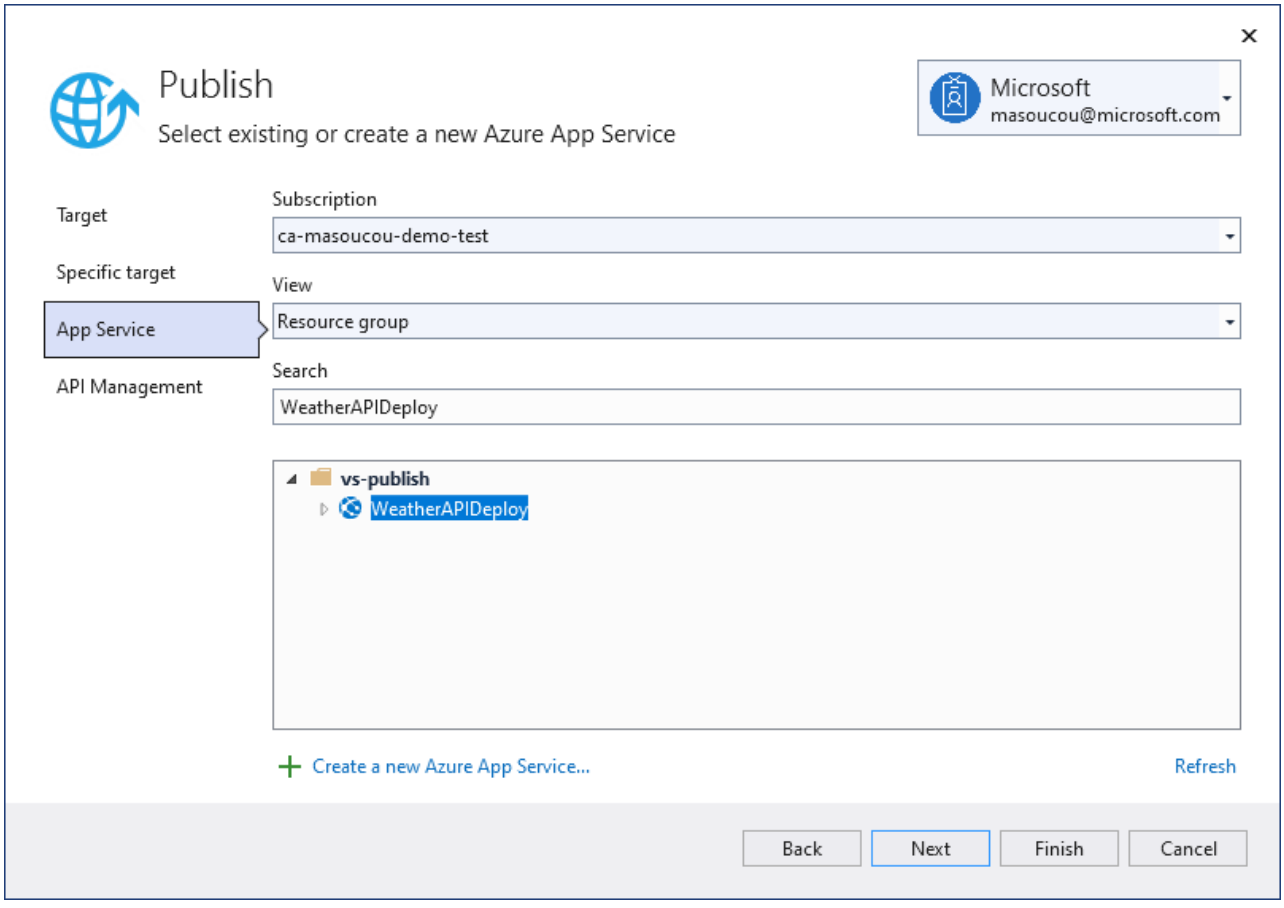


The **Create App Service** dialog appears. The **App Name**, **Resource Group**, and **App Service Plan** entry fields are populated. You can keep these names or change them.

5. Select the **Create** button.



After creation is completed, the dialog is automatically closed and the **Publish** dialog gets focus again. The instance that was created is automatically selected.

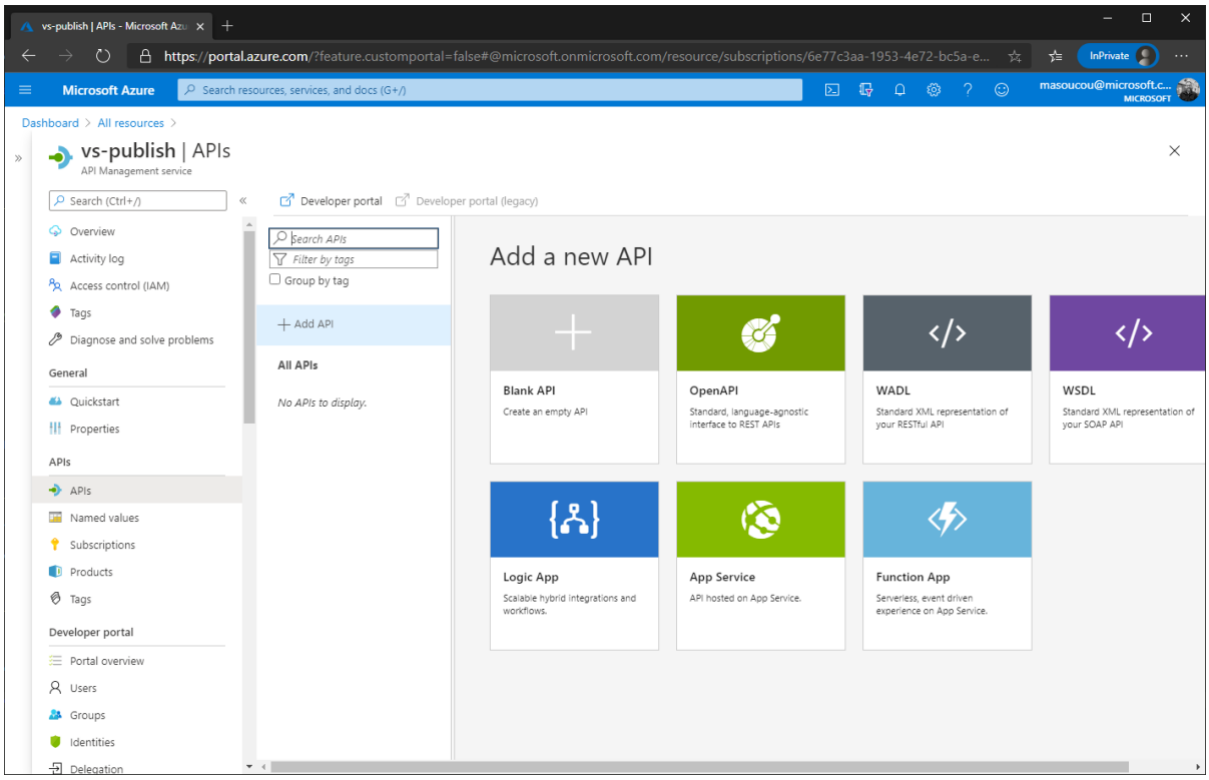


At this point, you need to add an API to the Azure API Management service. Leave Visual Studio open while you

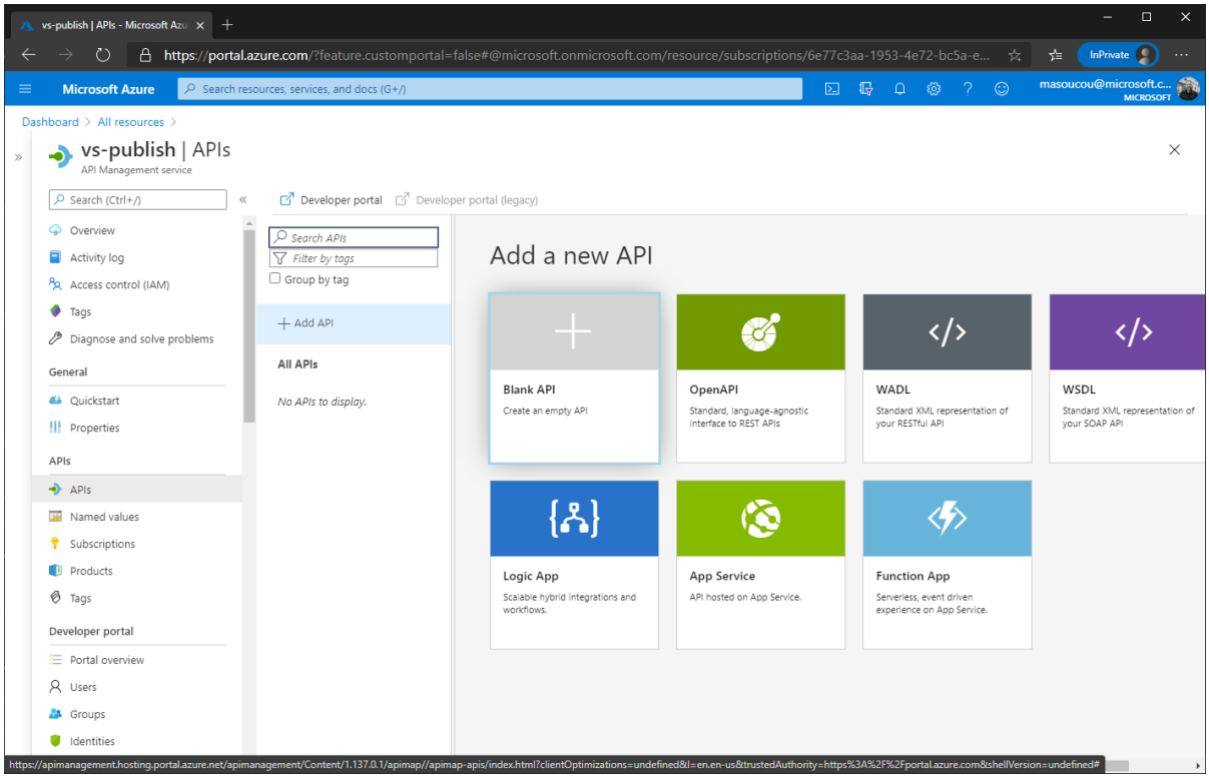
complete the following tasks.

Add an API to Azure API Management

1. Open the API Management Service instance created previously in the Azure portal. Select the **APIs** blade:



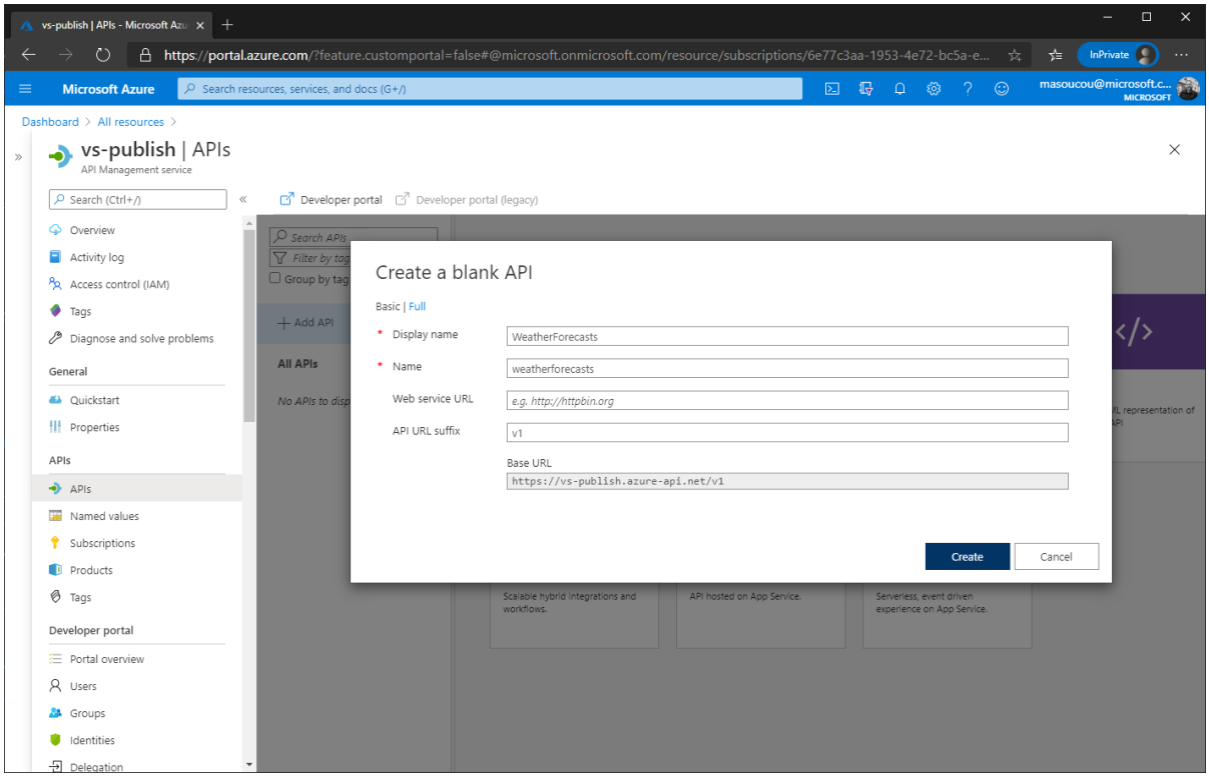
2. From the **Add a new API** panel, select the **Blank API** tile:



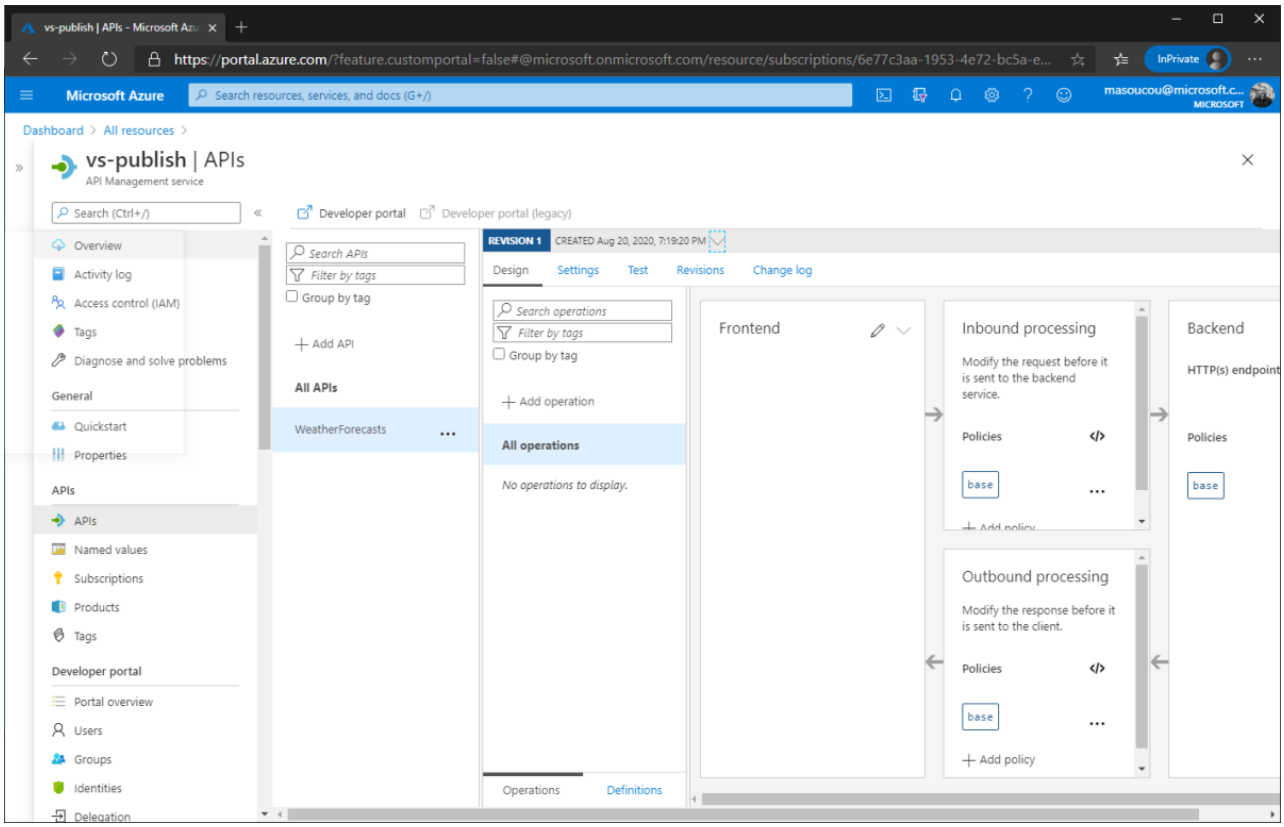
3. Enter the following values in the **Create a blank API** dialog that appears:

   - **Display Name**: *WeatherForecasts*
   - **Name**: *weatherforecasts*
   - **API Url suffix**: *v1*
   - Leave the **Web service URL** field empty.

4. Select the **Create** button.

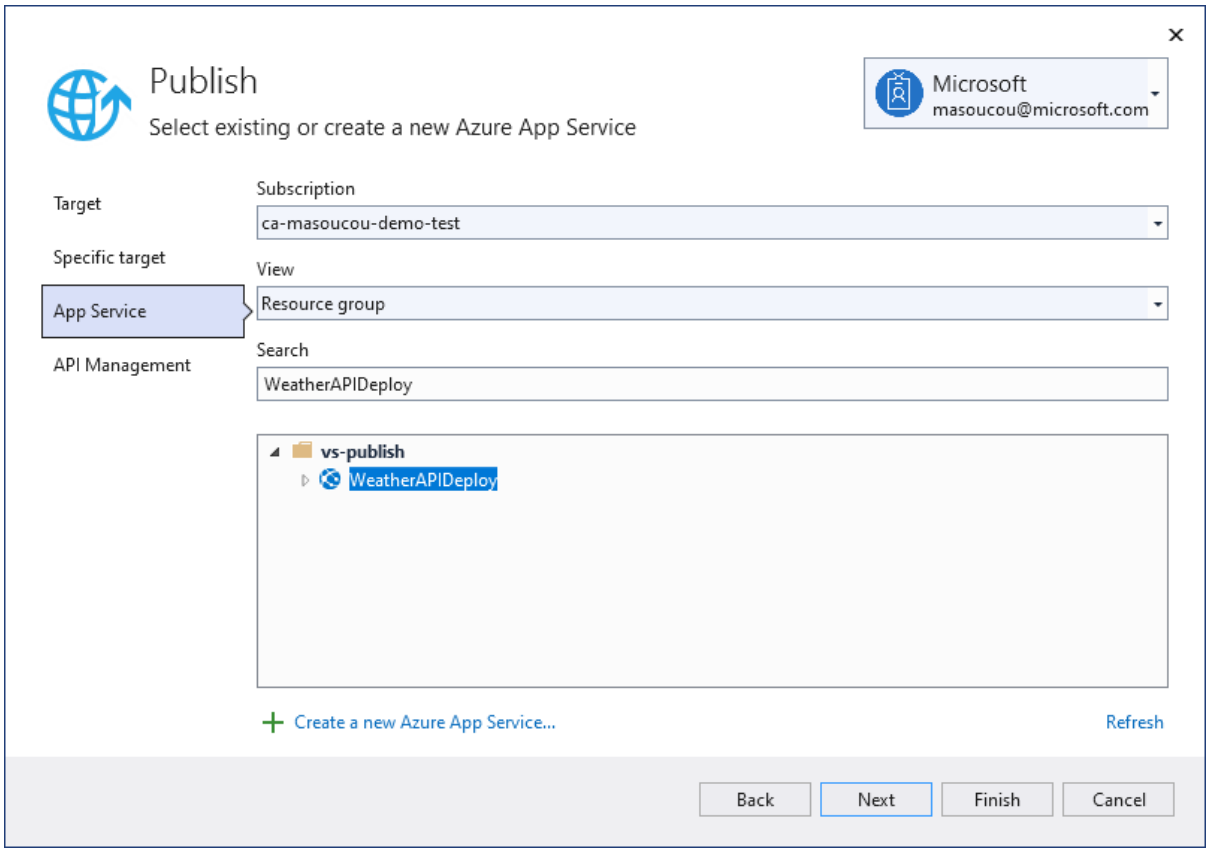


The blank API is created.



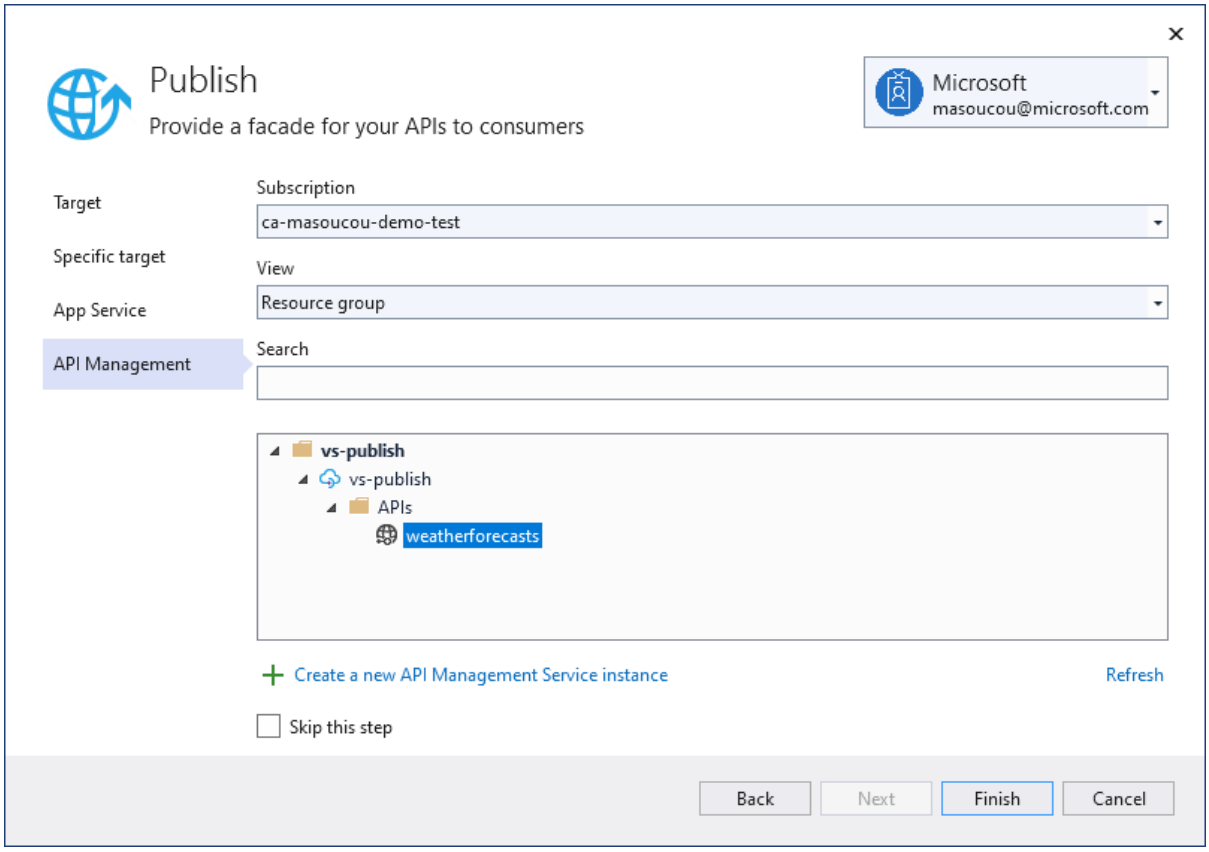
Publish the ASP.NET Core web API to Azure API Management

1. Switch back to Visual Studio. The **Publish** dialog should still be open where you left off before.
2. Select the newly published Azure App Service so it's highlighted.
3. Select the **Next** button.

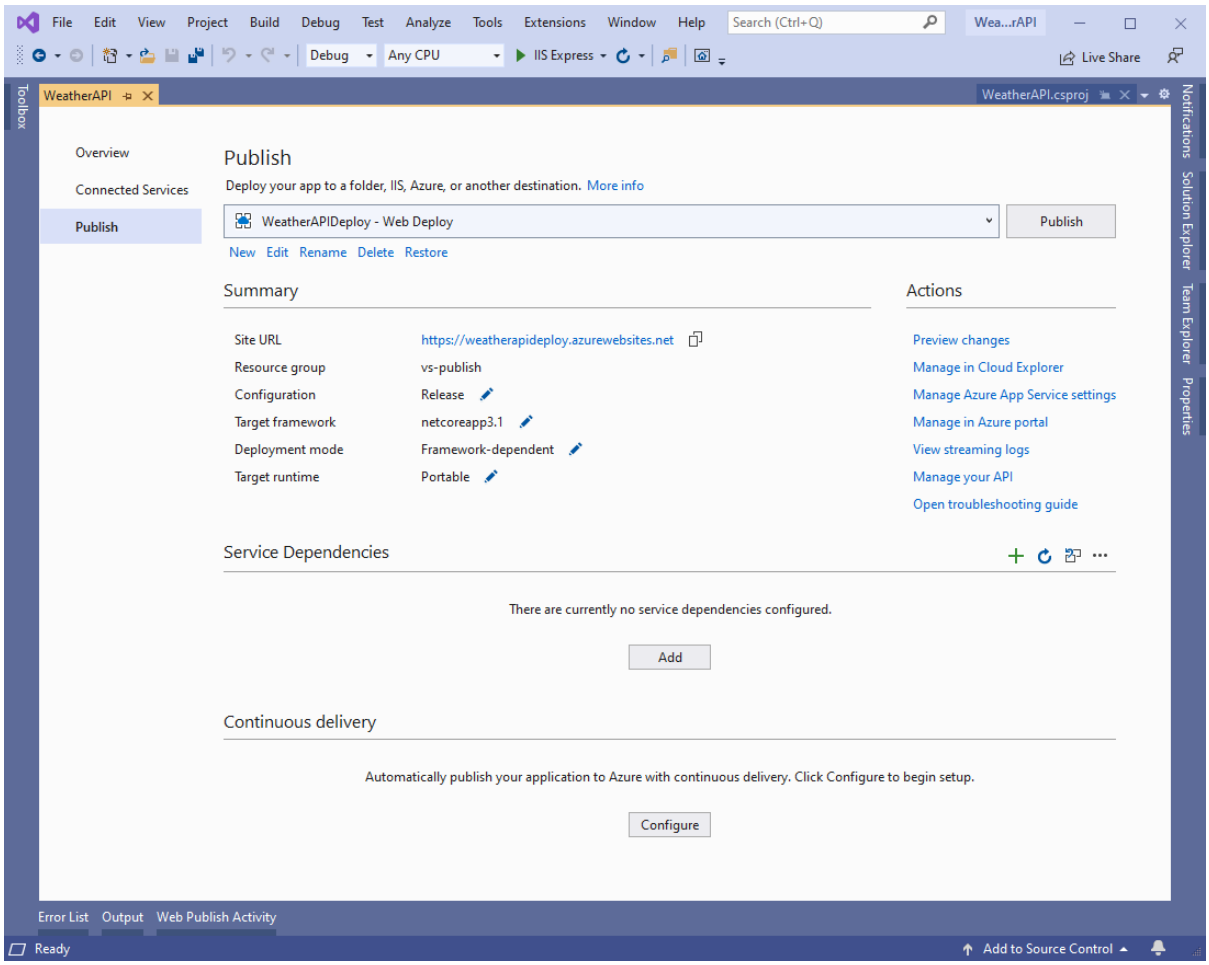


4. The dialog now shows the Azure API Management service created before. Expand it and the *APIs* folder to see the blank API you created.

5. Select the blank API's name and select the **Finish** button.



6. The dialog closes and a summary screen appears with information about the publish. Select the **Publish** button.

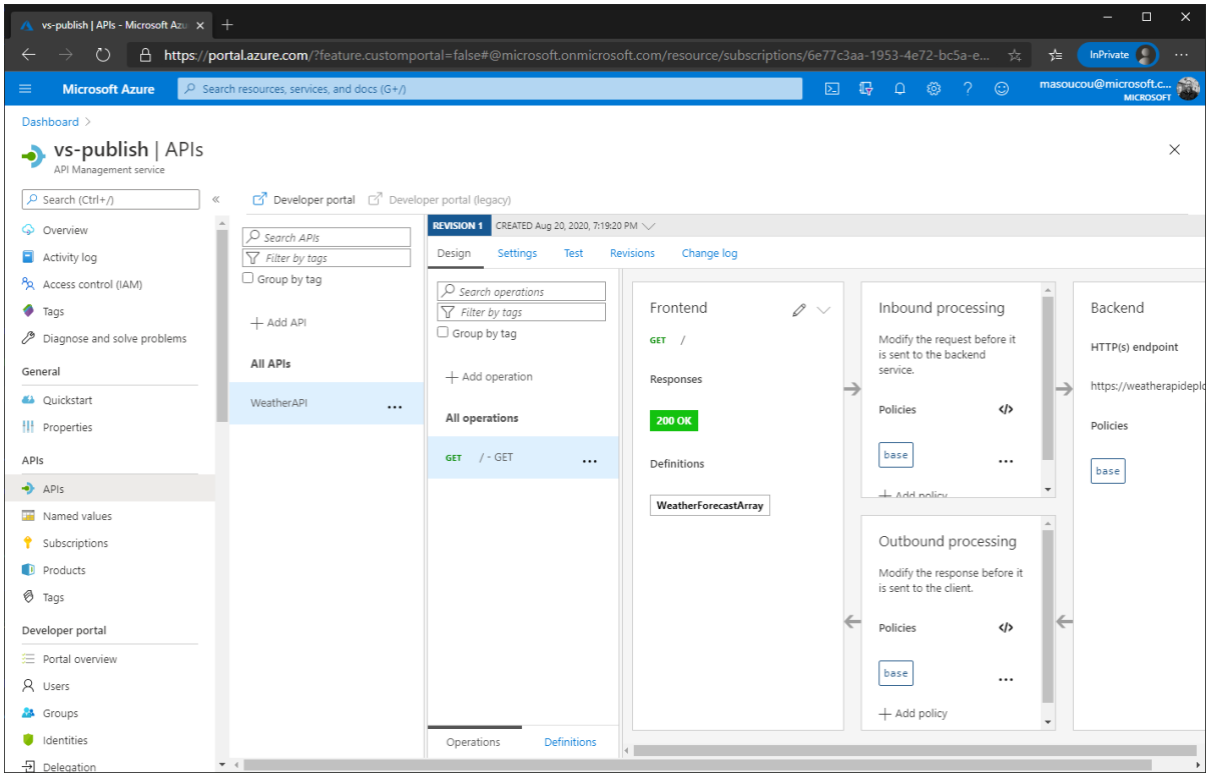


The web API will publish to both Azure App Service and Azure API Management. A new browser window will appear and show the API running in Azure App Service. You can close that window.

7. Switch back to the Azure API Management instance in the Azure portal.

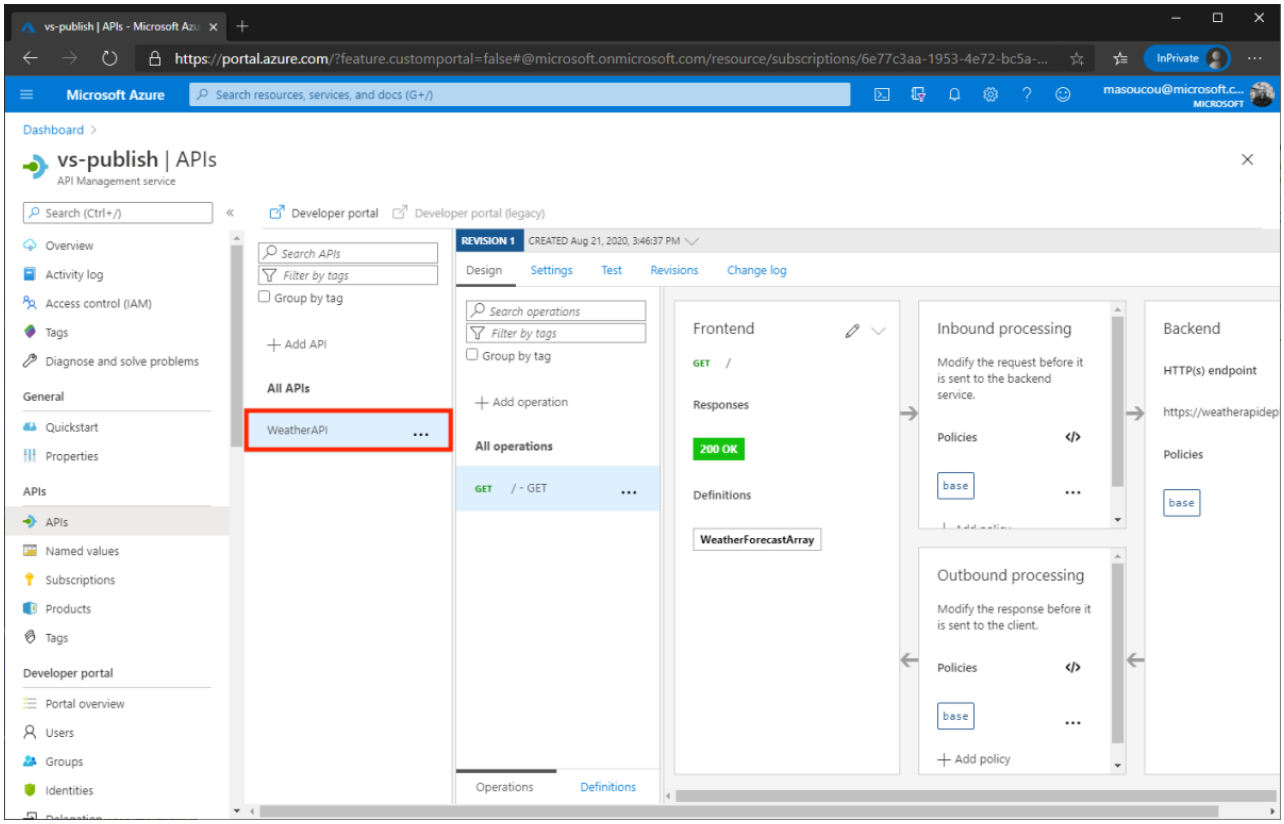
8. Refresh the browser window.

9. Select the blank API you created in the preceding steps. It's now populated and you can explore around.



Configure the published API name

Notice the name of the API is different than what you named it. The published API is named *WeatherAPI*, however, you named it *WeatherForecasts* when you created it. Complete the following steps to fix the name:



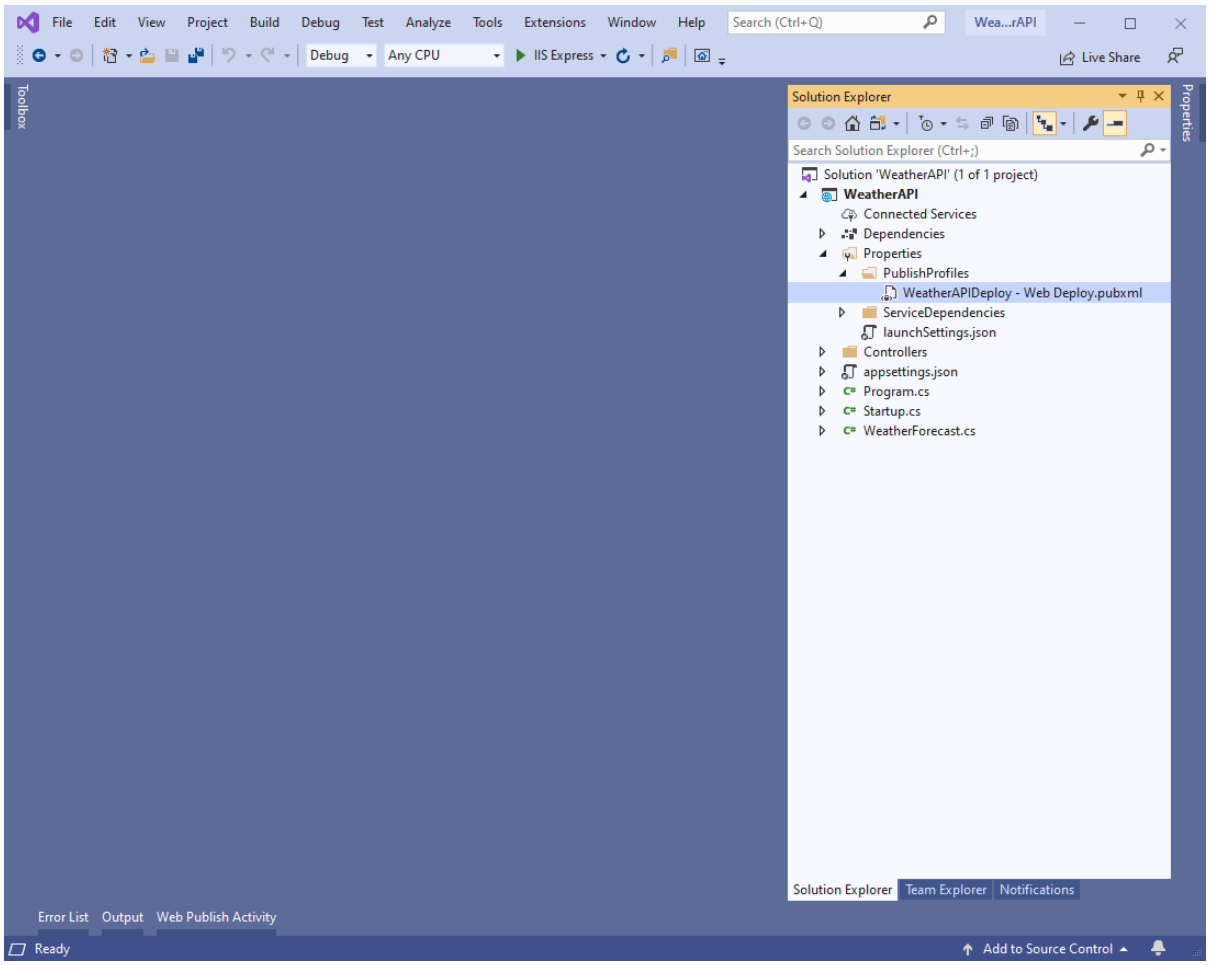
1. Delete the following line from the `Startup.ConfigureServices` method:

```
services.AddSwaggerGen();
```

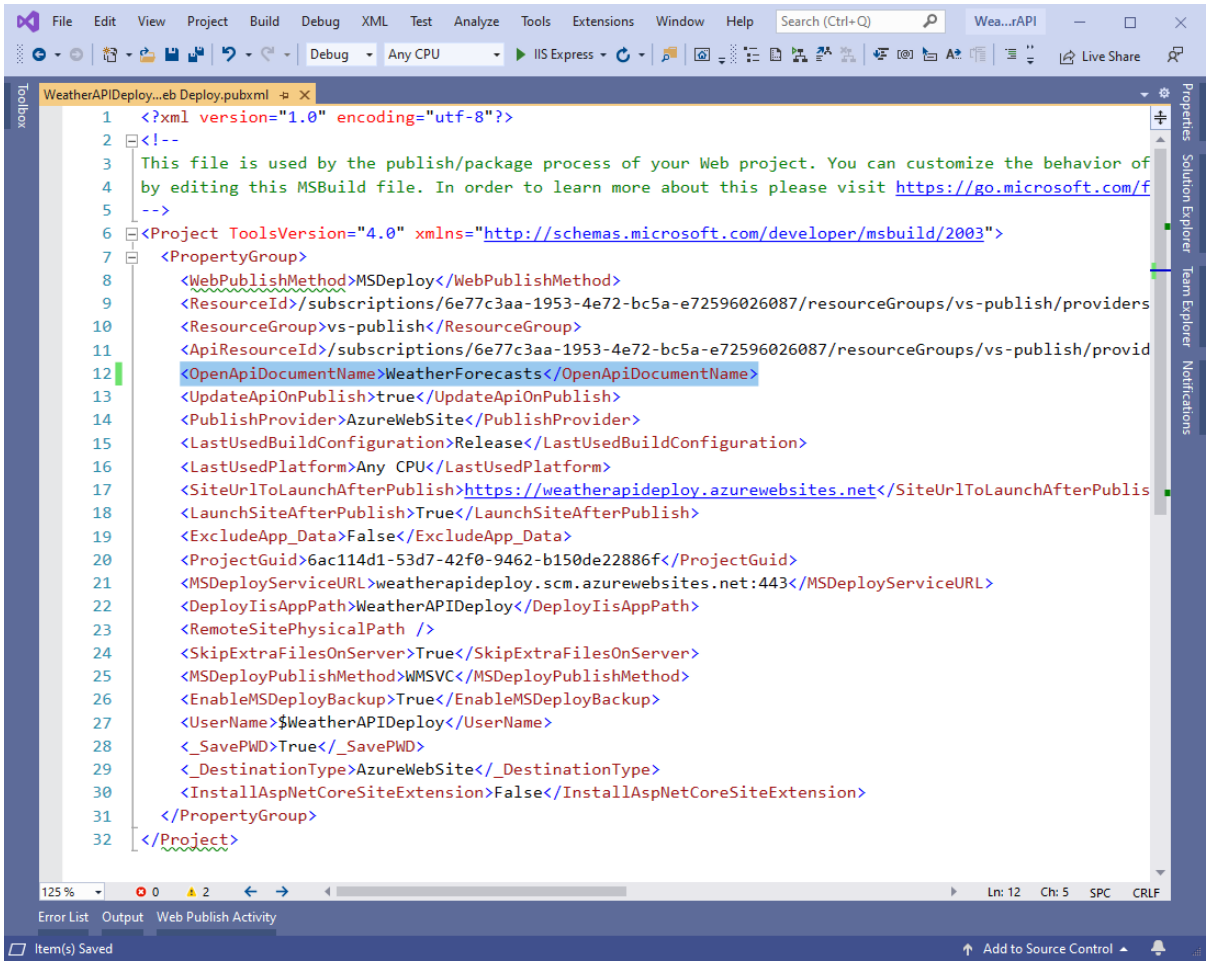
2. Add the following code to the `Startup.ConfigureServices` method:

```
services.AddSwaggerGen(config =>
{
    config.SwaggerDoc("WeatherForecasts", new Microsoft.OpenApi.Models.OpenApiInfo
    {
        Title = "Weather Forecasts",
        Version = "v1"
    });
});
```

3. Open the newly created publish profile. It can be found from **Solution Explorer** in the *Properties/PublishProfiles* folder.

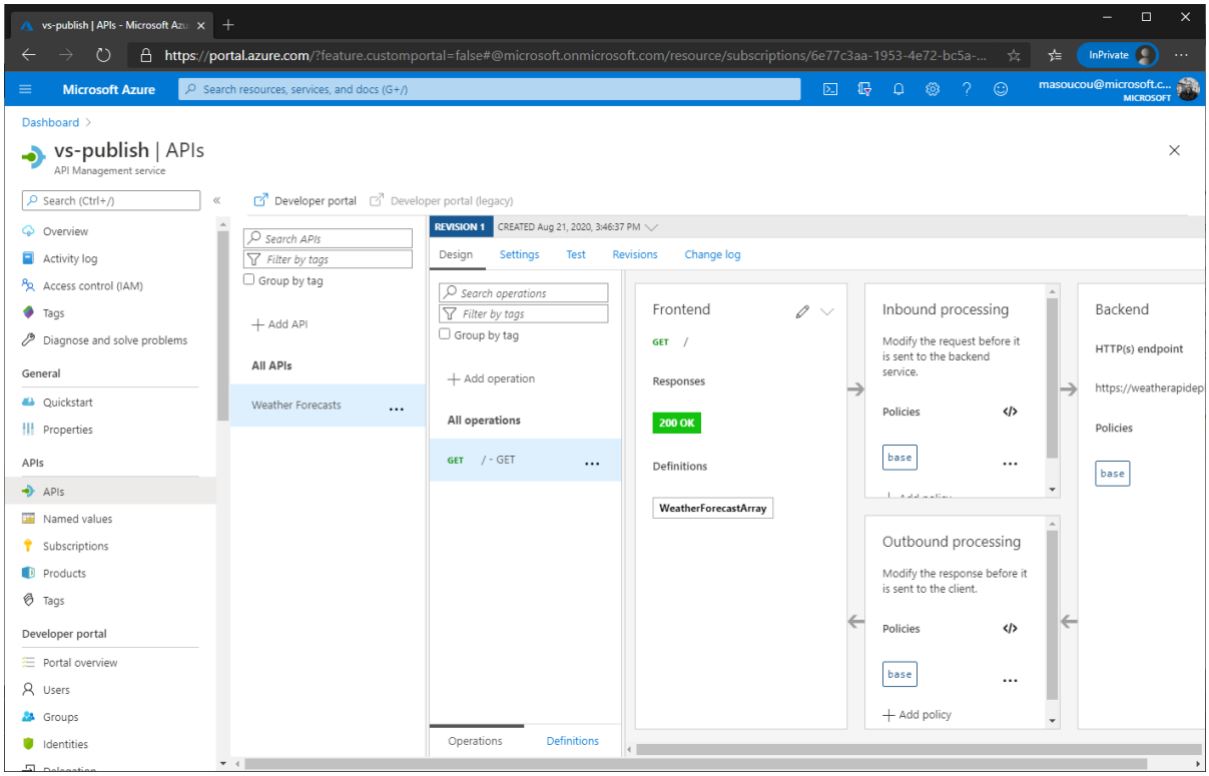


4. Change the `<OpenAPIDocumentName>` element's value from `v1` to `WeatherForecasts`.



5. Republish the ASP.NET Core web API and open the Azure API Management instance in the Azure portal.

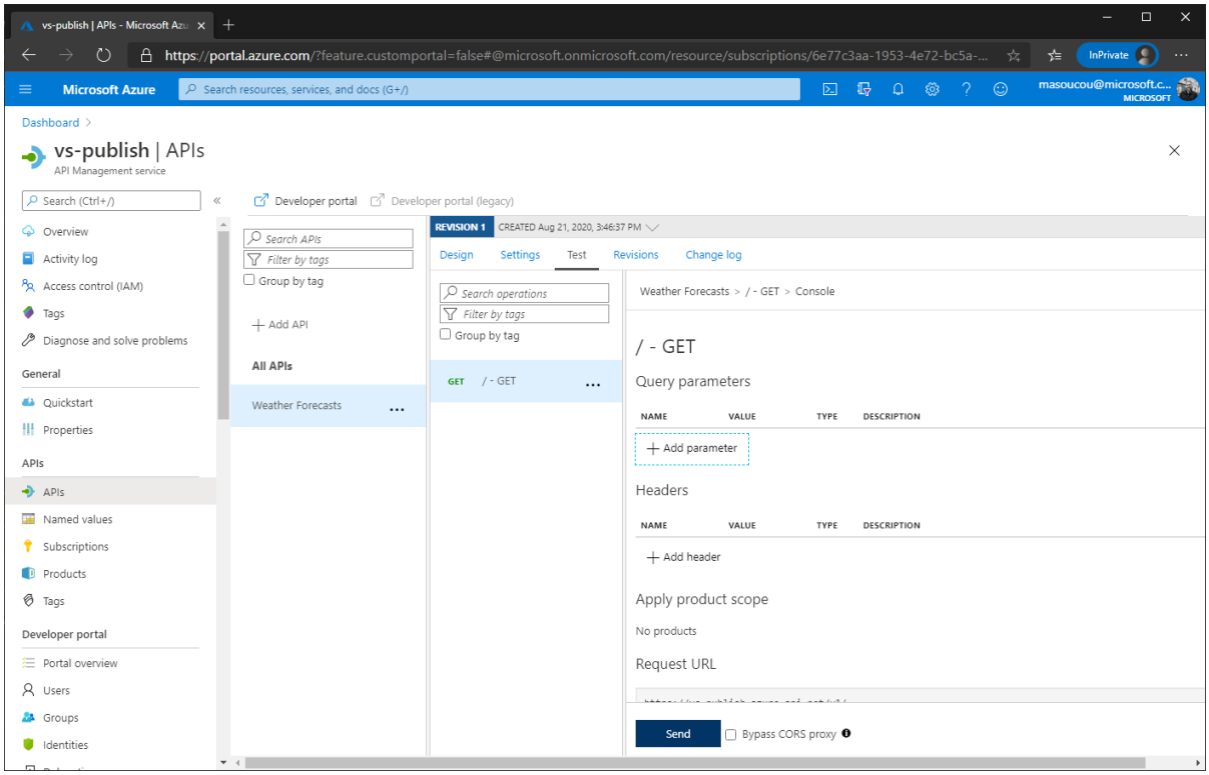
6. Refresh the page in your browser. You'll see the name of the API is now correct.



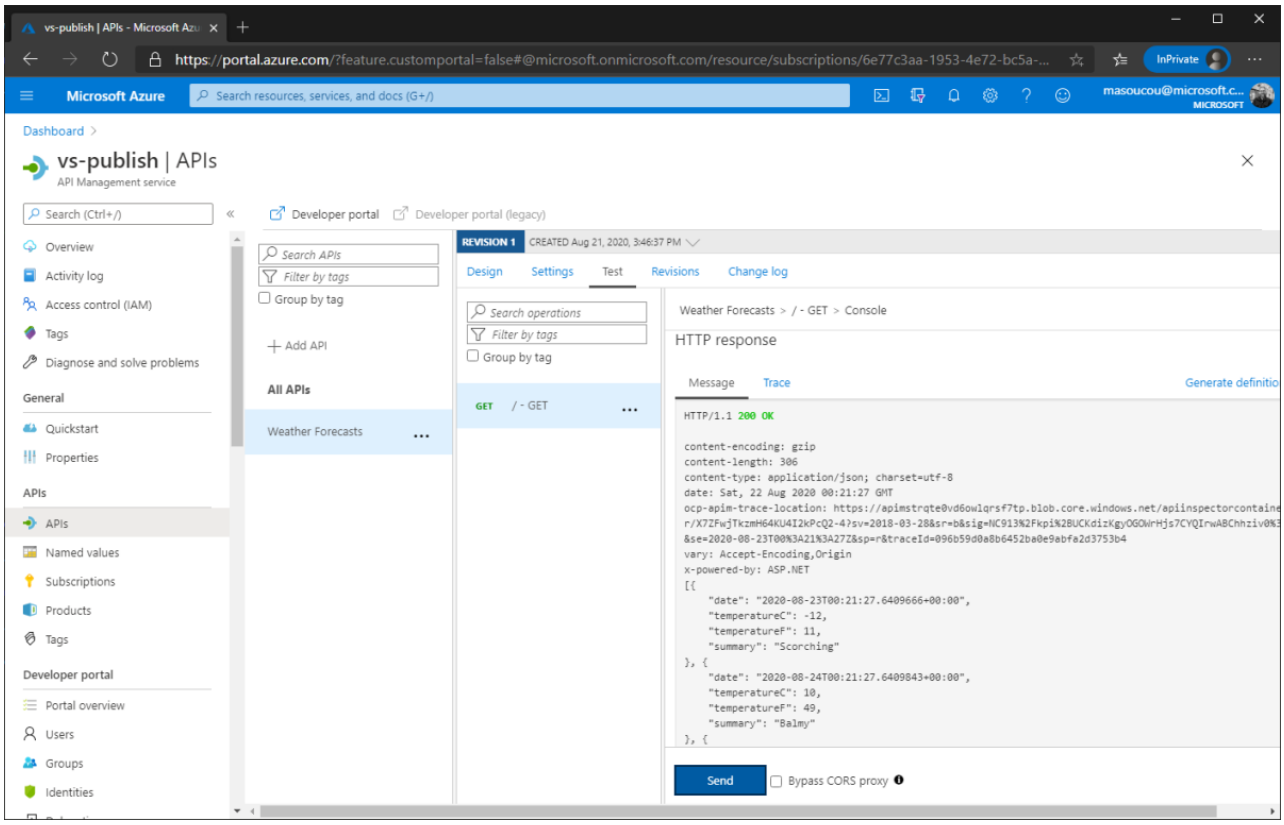
Verify the web API is working

You can test the deployed ASP.NET Core web API in Azure API Management from the Azure portal with the following steps:

1. Open the **Test** tab.
2. Select **/** or the **Get** operation.
3. Select **Send**.



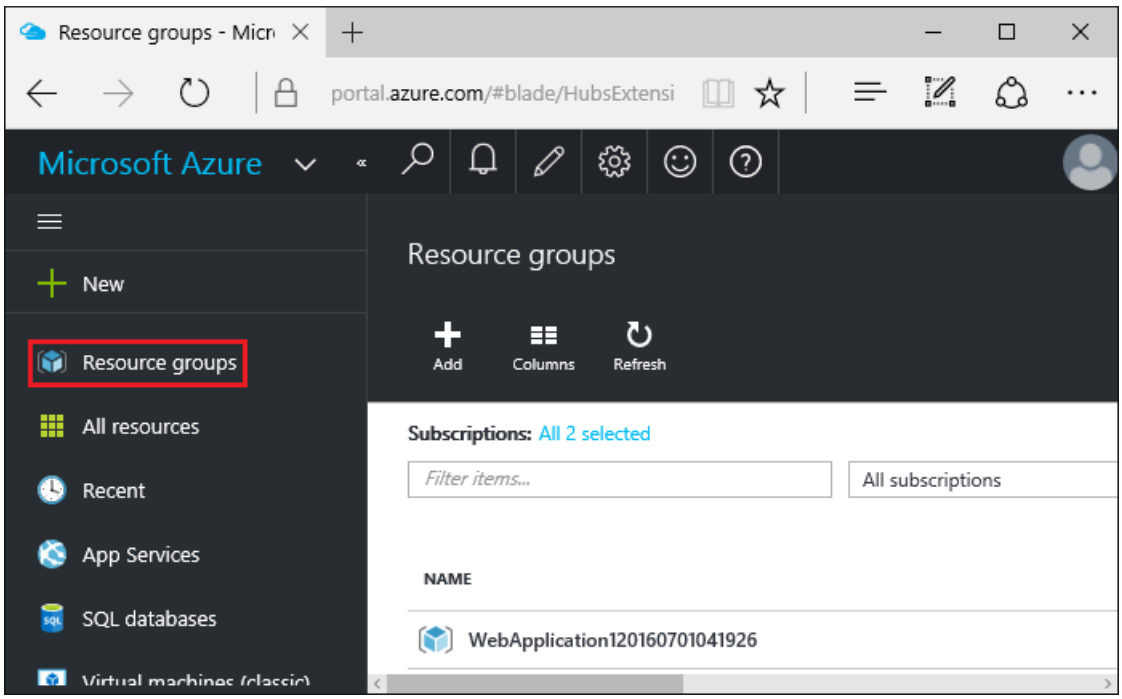
A successful response will look like the following:



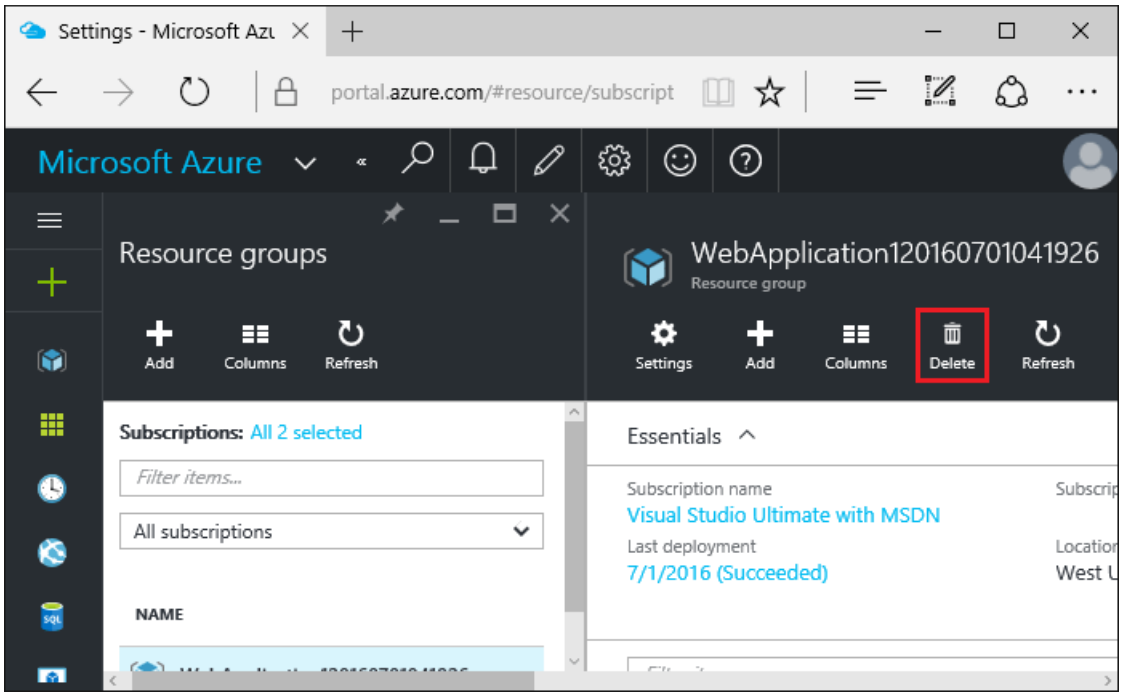
Clean up

When you've finished testing the app, go to the Azure portal and delete the app.

1. Select **Resource groups**, then select the resource group you created.



2. In the **Resource groups** page, select **Delete**.



3. Enter the name of the resource group and select **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure.

Next steps

Continuous deployment to Azure with Visual Studio and Git with ASP.NET Core

Additional resources

- Azure API Management
- Azure App Service

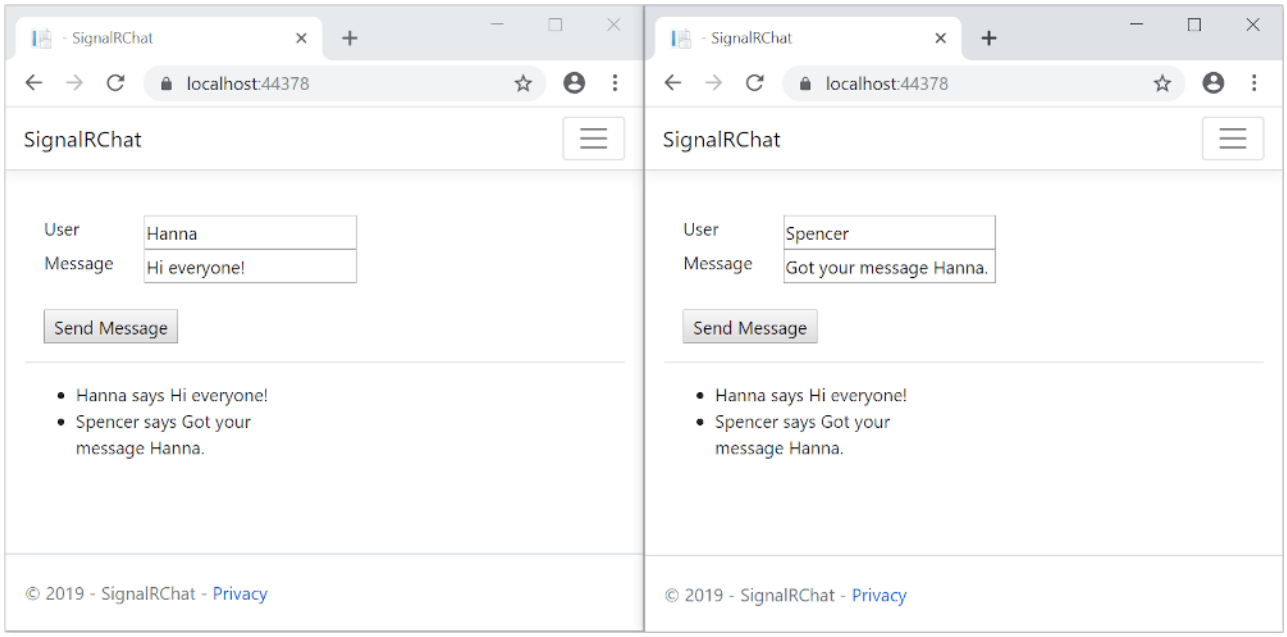
Tutorial: Get started with ASP.NET Core SignalR

9/22/2020 • 13 minutes to read • Edit Online

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- Create a web project.
- Add the SignalR client library.
- Create a SignalR hub.
- Configure the project to use SignalR.
- Add code that sends messages from any client to all connected clients.

At the end, you'll have a working chat app:



Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

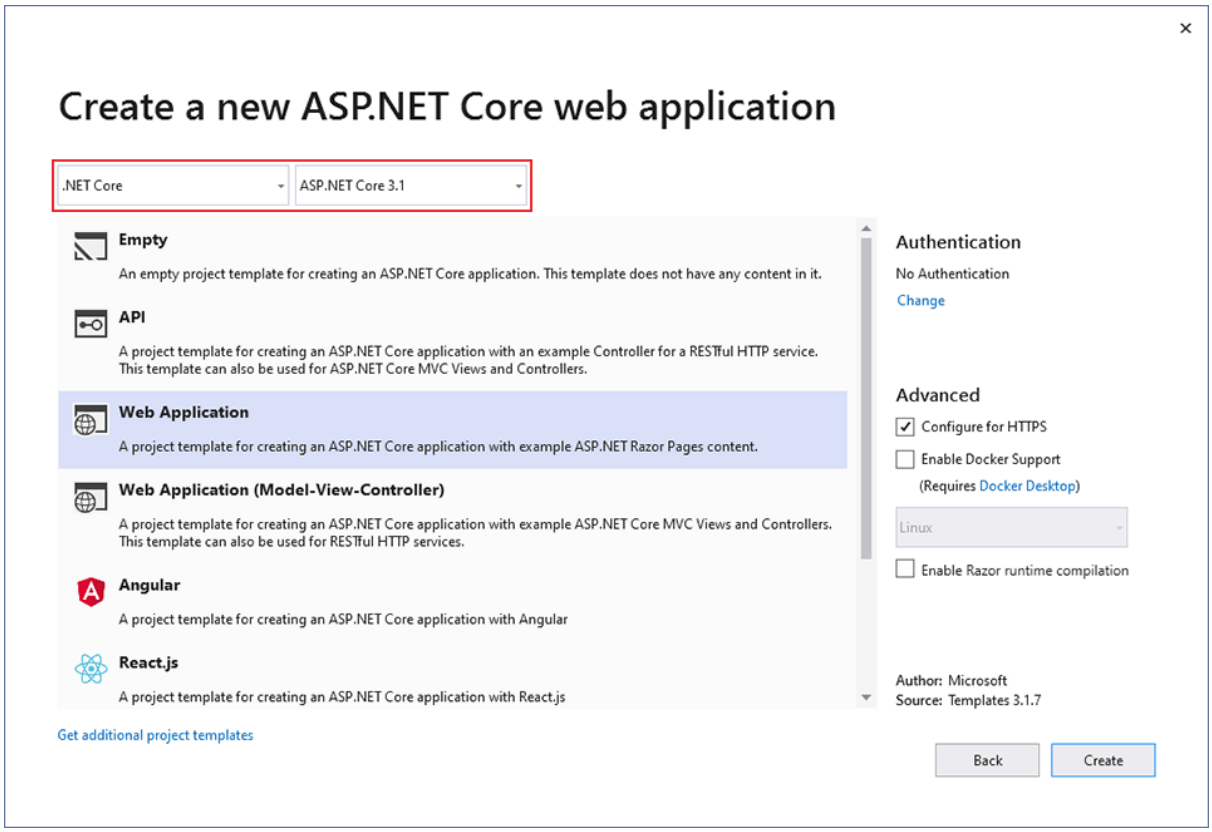
- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a web app project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the menu, select **File > New Project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web Application**, and then select **Next**.
- In the **Configure your new project** dialog, name the project *SignalRChat*, and then select **Create**.

- In the **Create a new ASP.NET Core web Application** dialog, select **.NET Core** and **ASP.NET Core 3.1**.
- Select **Web Application** to create a project that uses Razor Pages, and then select **Create**.

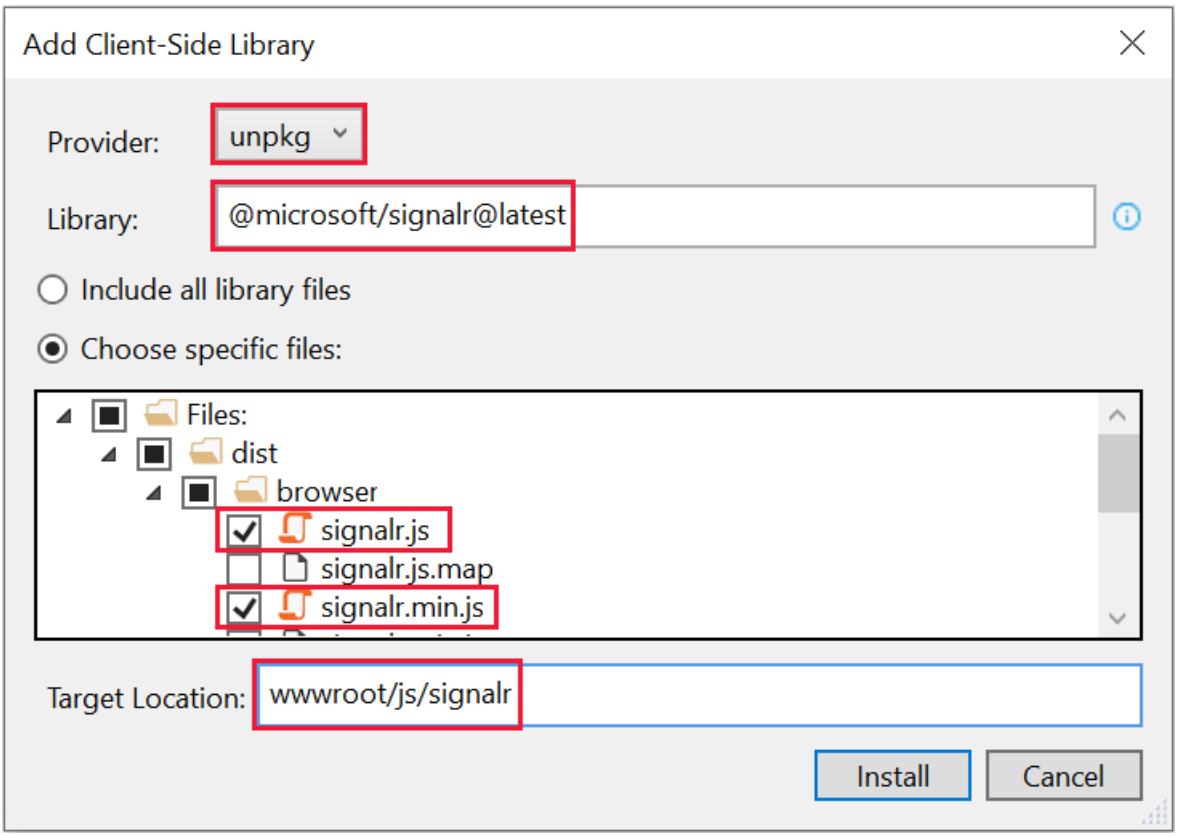


Add the SignalR client library

The SignalR server library is included in the ASP.NET Core 3.1 shared framework. The JavaScript client library isn't automatically included in the project. For this tutorial, you use Library Manager (LibMan) to get the client library from *unpkg*. unpkg is a content delivery network (CDN) that can deliver anything found in npm, the Node.js package manager.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.
- In the **Add Client-Side Library** dialog, for **Provider** select **unpkg**.
- For **Library**, enter `@microsoft/signalr@latest`.
- Select **Choose specific files**, expand the *dist/browser* folder, and select *signalr.js* and *signalr.min.js*.
- Set **Target Location** to *wwwroot/js/signalr/*, and select **Install**.



LibMan creates a *wwwroot/js/signalr* folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

- In the SignalRChat project folder, create a *Hubs* folder.
- In the *Hubs* folder, create a *ChatHub.cs* file with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

The `ChatHub` class inherits from the SignalR `Hub` class. The `Hub` class manages connections, groups, and messaging.

The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR.

- Add the following highlighted code to the *Startup.cs* file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
            services.AddSignalR();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
                endpoints.MapHub<ChatHub>("/chathub");
            });
        }
    }
}
```

These changes add SignalR to the ASP.NET Core dependency injection and routing systems.

Add SignalR client code

- Replace the content in *Pages\Index.cshtml* with the following code:

```
@page
    <div class="container">
        <div class="row"> </div>
        <div class="row">
            <div class="col-2">User</div>
            <div class="col-4"><input type="text" id="userInput" /></div>
        </div>
        <div class="row">
            <div class="col-2">Message</div>
            <div class="col-4"><input type="text" id="messageInput" /></div>
        </div>
        <div class="row"> </div>
        <div class="row">
            <div class="col-6">
                <input type="button" id="sendButton" value="Send Message" />
            </div>
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <hr />
        </div>
    </div>
    <div class="row">
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

  The preceding code:

  - Creates text boxes for name and message text, and a submit button.
  - Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
  - Includes script references to SignalR and the *chat.js* application code that you create in the next step.
- In the *wwwroot/js* folder, create a *chat.js* file with the following code:

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable send button until connection is established
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    var encodedMsg = user + " says " + msg;
    var li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding code:

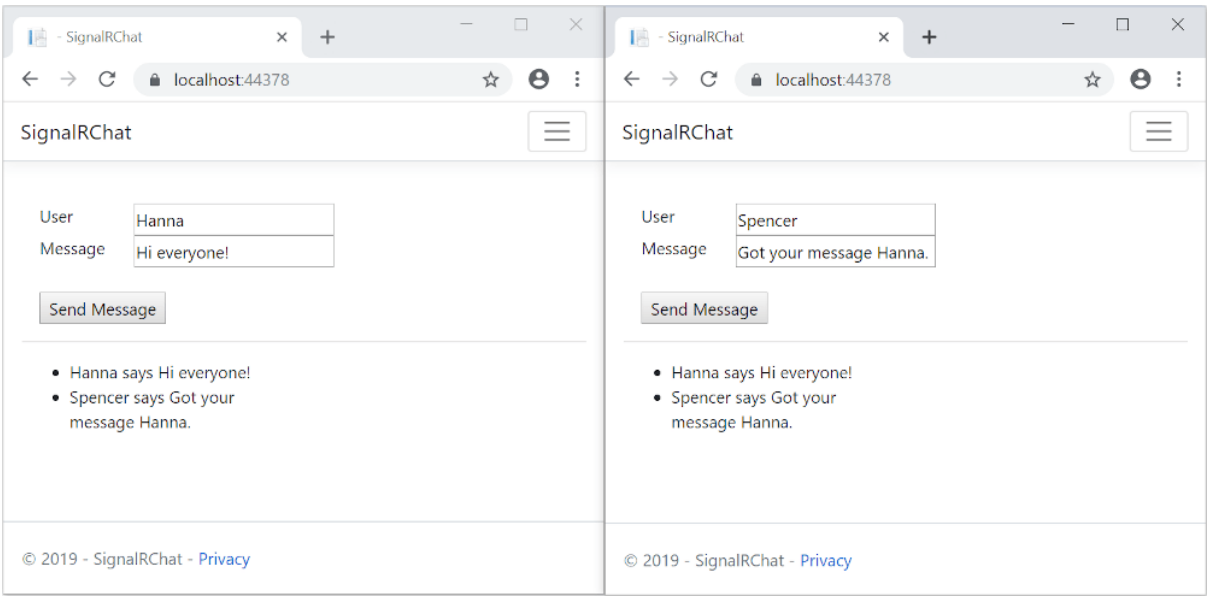
- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

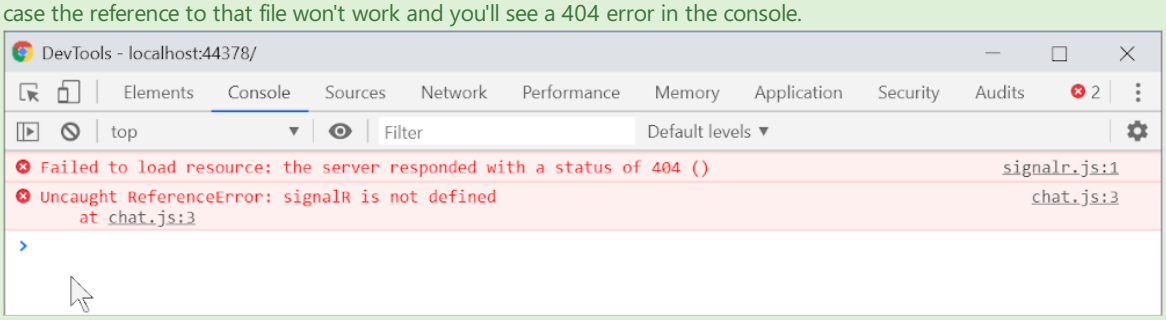
- Press **CTRL+F5** to run the app without debugging.
- Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.
- Choose either browser, enter a name and message, and select the **Send Message** button.

  The name and message are displayed on both pages instantly.



TIP

- If the app doesn't work, open your browser developer tools (F12) and go to the console. You might see errors related to your HTML and JavaScript code. For example, suppose you put *signalr.js* in a different folder than directed. In that case the reference to that file won't work and you'll see a 404 error in the console.

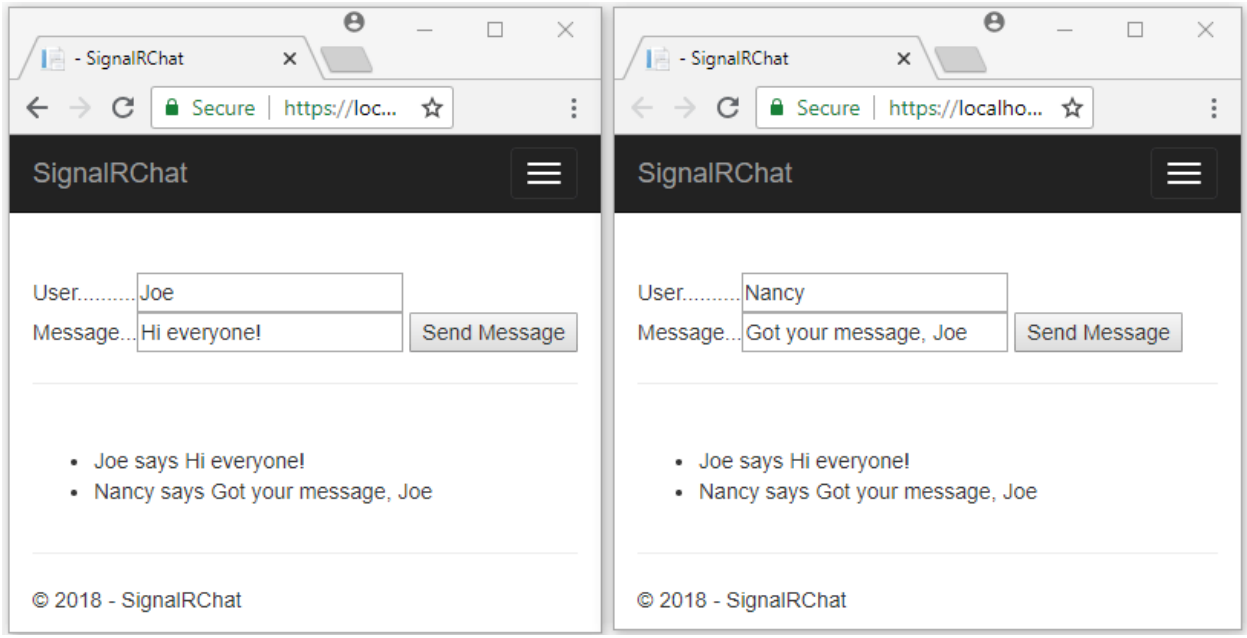


- If you get the error ERR_SPDY_INADEQUATE_TRANSPORT_SECURITY in Chrome, run these commands to update your development certificate:

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- Create a web project.
- Add the SignalR client library.
- Create a SignalR hub.
- Configure the project to use SignalR.
- Add code that sends messages from any client to all connected clients. At the end, you'll have a working chat app:



Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

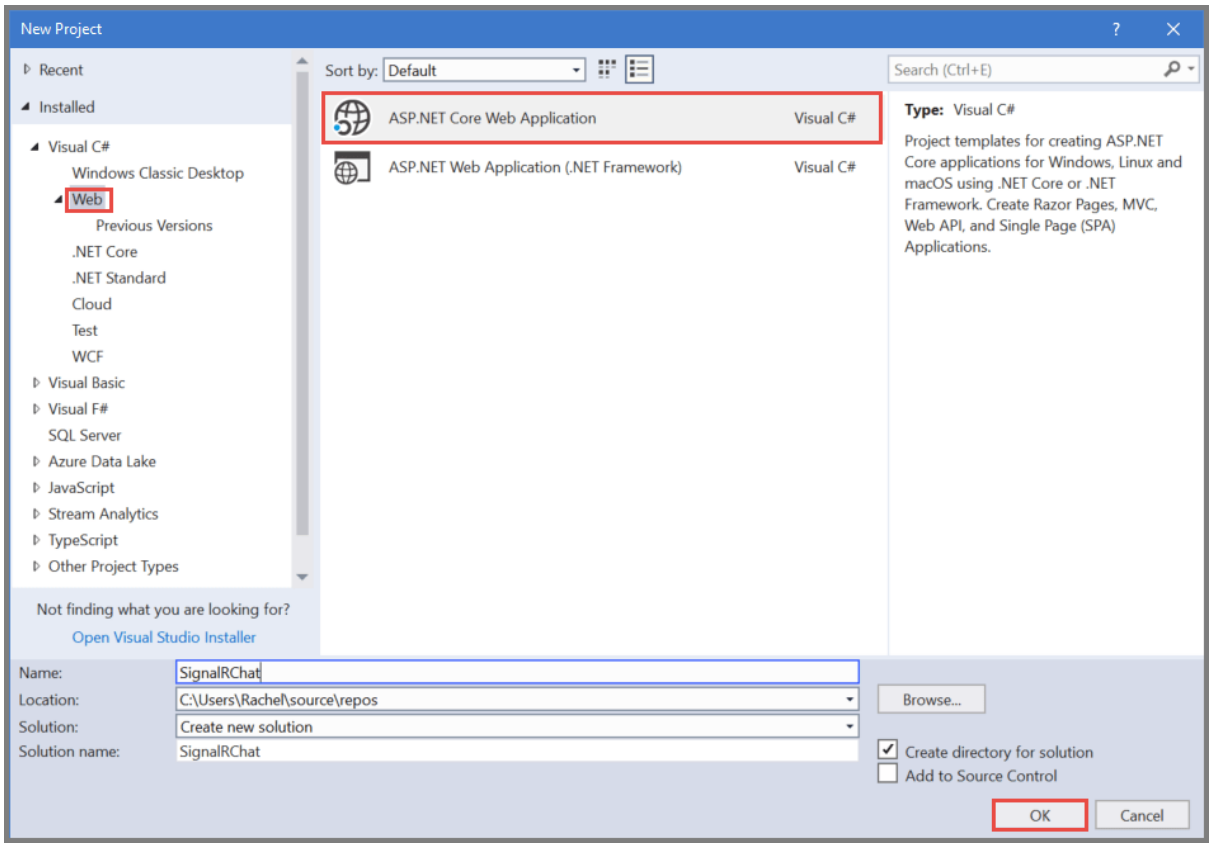
- Visual Studio 2017 version 15.9 or later with the **ASP.NET and web development** workload. You can use Visual Studio 2019, but some project creation steps differ from what's shown in the tutorial.
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

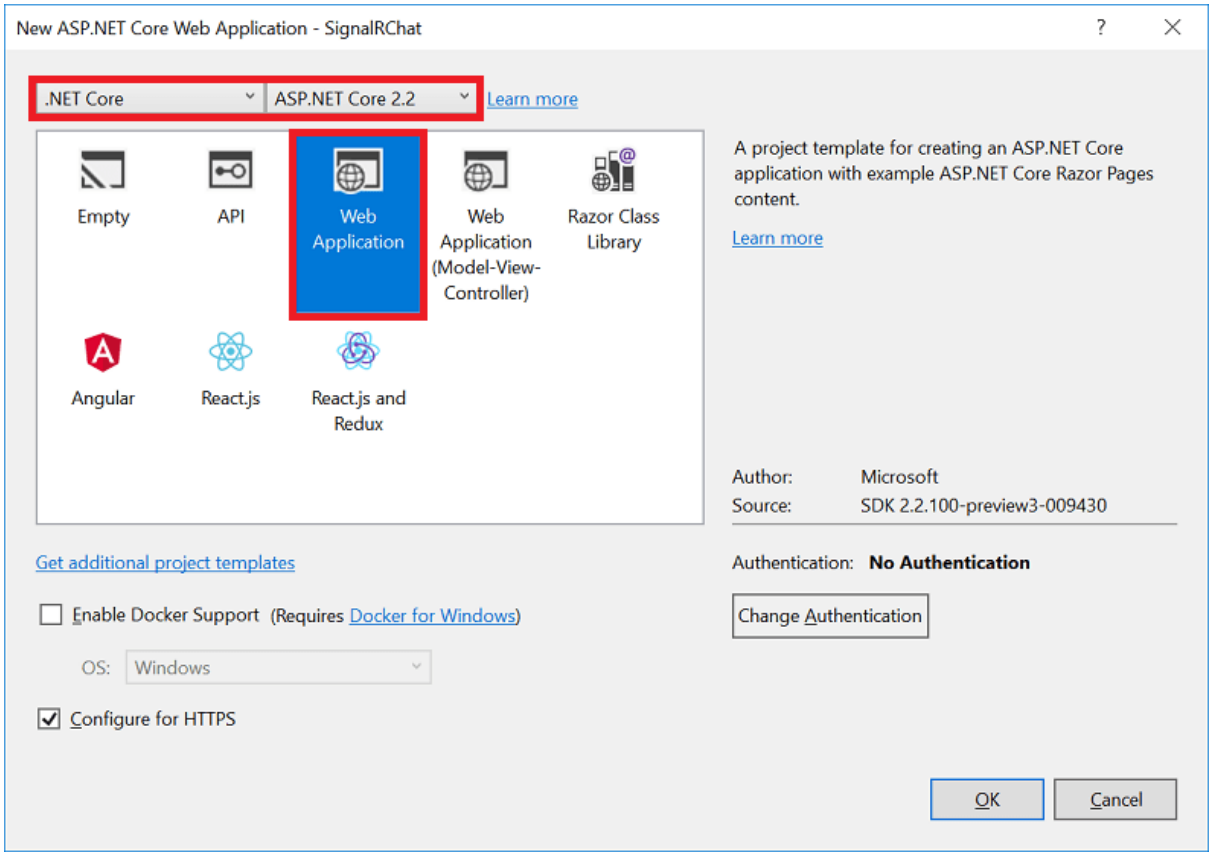
Create a web project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the menu, select **File > New Project**.
- In the **New Project** dialog, select **Installed > Visual C# > Web > ASP.NET Core Web Application**. Name the project *SignalRChat*.



- Select **Web Application** to create a project that uses Razor Pages.
- Select a target framework of **.NET Core**, select **ASP.NET Core 2.2**, and click **OK**.



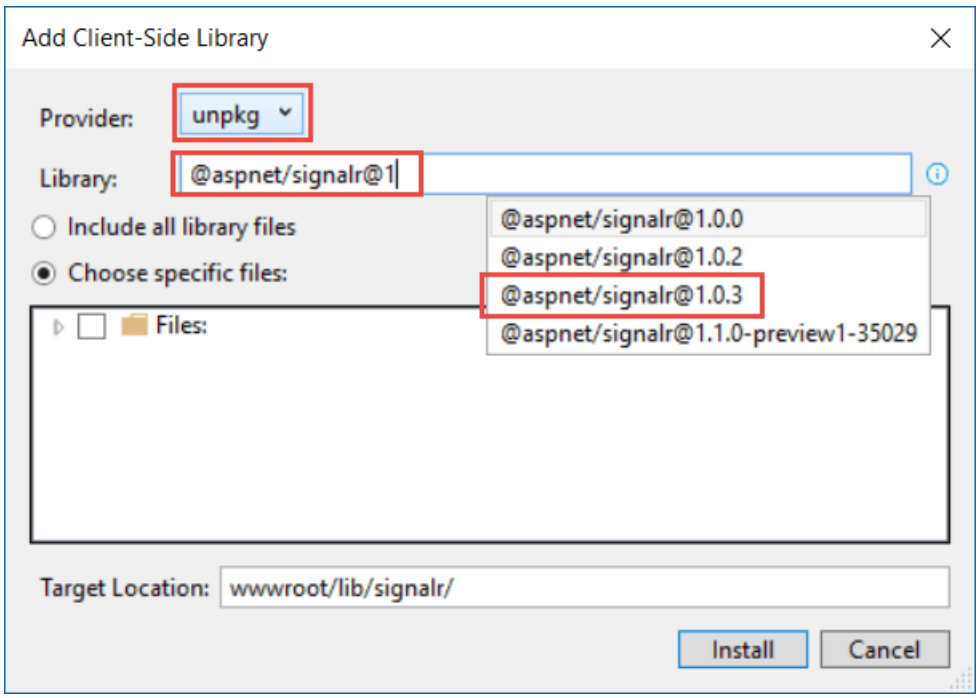
Add the SignalR client library

The SignalR server library is included in the `Microsoft.AspNetCore.App` metapackage. The JavaScript client library isn't automatically included in the project. For this tutorial, you use Library Manager (LibMan) to get the client library from *unpkg*. unpkg is a content delivery network (CDN) that can deliver anything found in npm, the Node.js

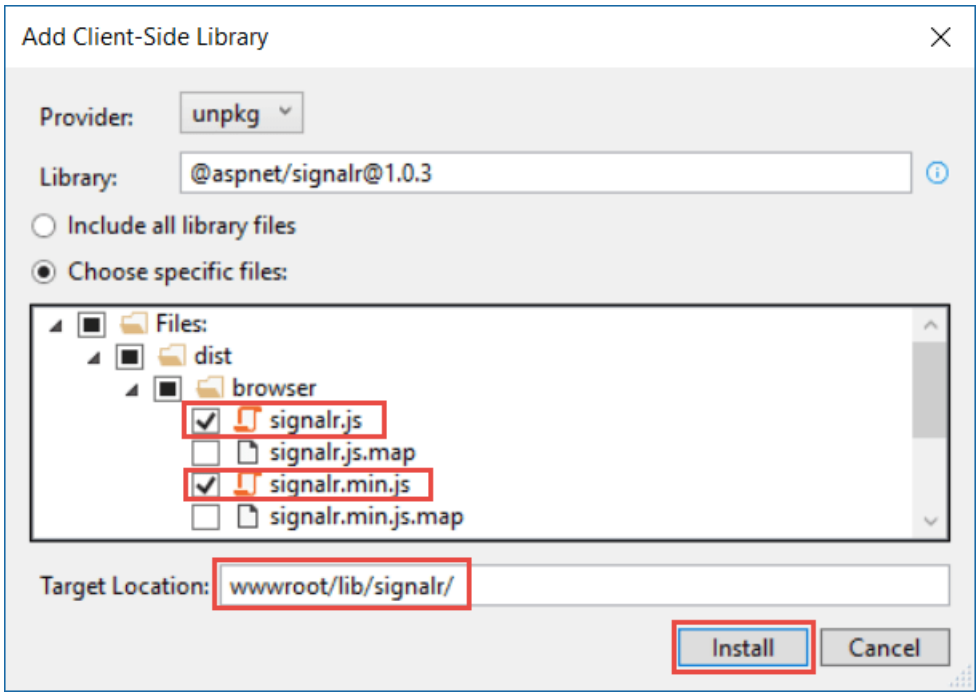
package manager.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.
- In the **Add Client-Side Library** dialog, for **Provider** select **unpkg**.
- For **Library**, enter `@microsoft/signalr@3`, and select the latest version that isn't preview.



- Select **Choose specific files**, expand the *dist/browser* folder, and select *signalr.js* and *signalr.min.js*.
- Set **Target Location** to *wwwroot/lib/signalr/*, and select **Install**.



LibMan creates a *wwwroot/lib/signalr* folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

- In the SignalRChat project folder, create a *Hubs* folder.
- In the *Hubs* folder, create a *ChatHub.cs* file with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

  The `ChatHub` class inherits from the SignalR `Hub` class. The `Hub` class manages connections, groups, and messaging.

  The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR.

- Add the following highlighted code to the *Startup.cs* file.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                // This lambda determines whether user consent for non-essential cookies is needed for a
given request.
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });


            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

            services.AddSignalR();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            app.UseSignalR(routes =>
            {
                routes.MapHub<ChatHub>("/chathub");
            });
            app.UseMvc();
        }
    }
}
```

These changes add SignalR to the ASP.NET Core dependency injection system and the middleware pipeline.

Add SignalR client code

- Replace the content in *Pages\Index.cshtml* with the following code:

```
@page
<div class="container">
    <div class="row"> </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            User.........<input type="text" id="userInput" />
            <br />
            Message...<input type="text" id="messageInput" />
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <hr />
        </div>
    </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/lib/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

  The preceding code:

  - Creates text boxes for name and message text, and a submit button.
  - Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
  - Includes script references to SignalR and the *chat.js* application code that you create in the next step.
- In the *wwwroot/js* folder, create a *chat.js* file with the following code:

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable send button until connection is established
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    var encodedMsg = user + " says " + msg;
    var li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().then(function(){
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding code:

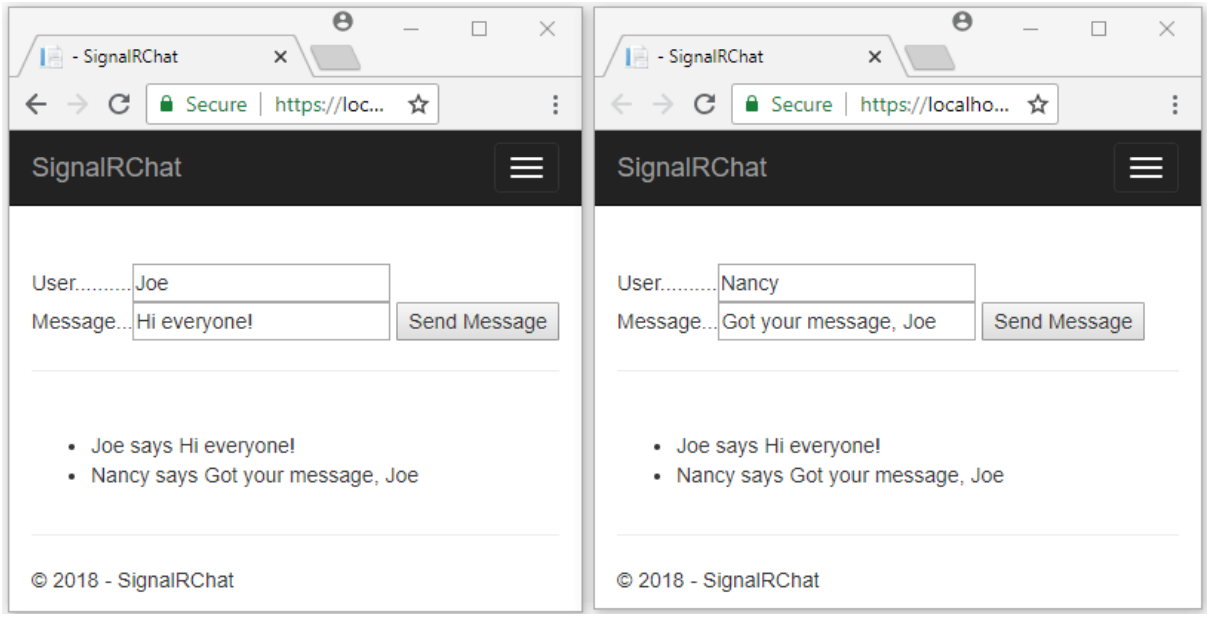
- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

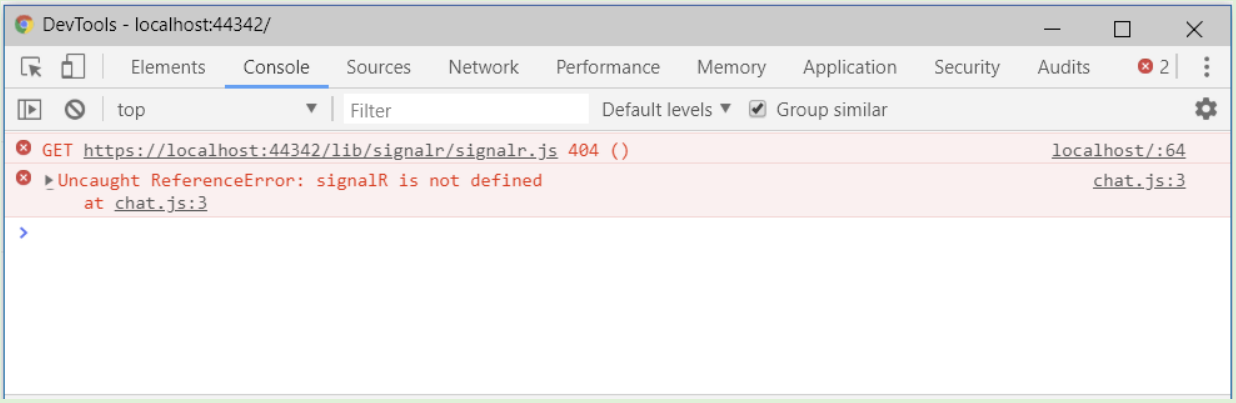
- Press **CTRL+F5** to run the app without debugging.
- Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.
- Choose either browser, enter a name and message, and select the **Send Message** button.

  The name and message are displayed on both pages instantly.



TIP

If the app doesn't work, open your browser developer tools (F12) and go to the console. You might see errors related to your HTML and JavaScript code. For example, suppose you put *signalr.js* in a different folder than directed. In that case the reference to that file won't work and you'll see a 404 error in the console.



Additional resources

- Youtube version of this tutorial

Use ASP.NET Core SignalR with TypeScript and Webpack

9/22/2020 • 21 minutes to read • Edit Online

By Sébastien Sougnez and Scott Addie

Webpack enables developers to bundle and build the client-side resources of a web app. This tutorial demonstrates using Webpack in an ASP.NET Core SignalR web app whose client is written in TypeScript.

In this tutorial, you learn how to:

- Scaffold a starter ASP.NET Core SignalR app
- Configure the SignalR TypeScript client
- Configure a build pipeline using Webpack
- Configure the SignalR server
- Enable communication between client and server

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code

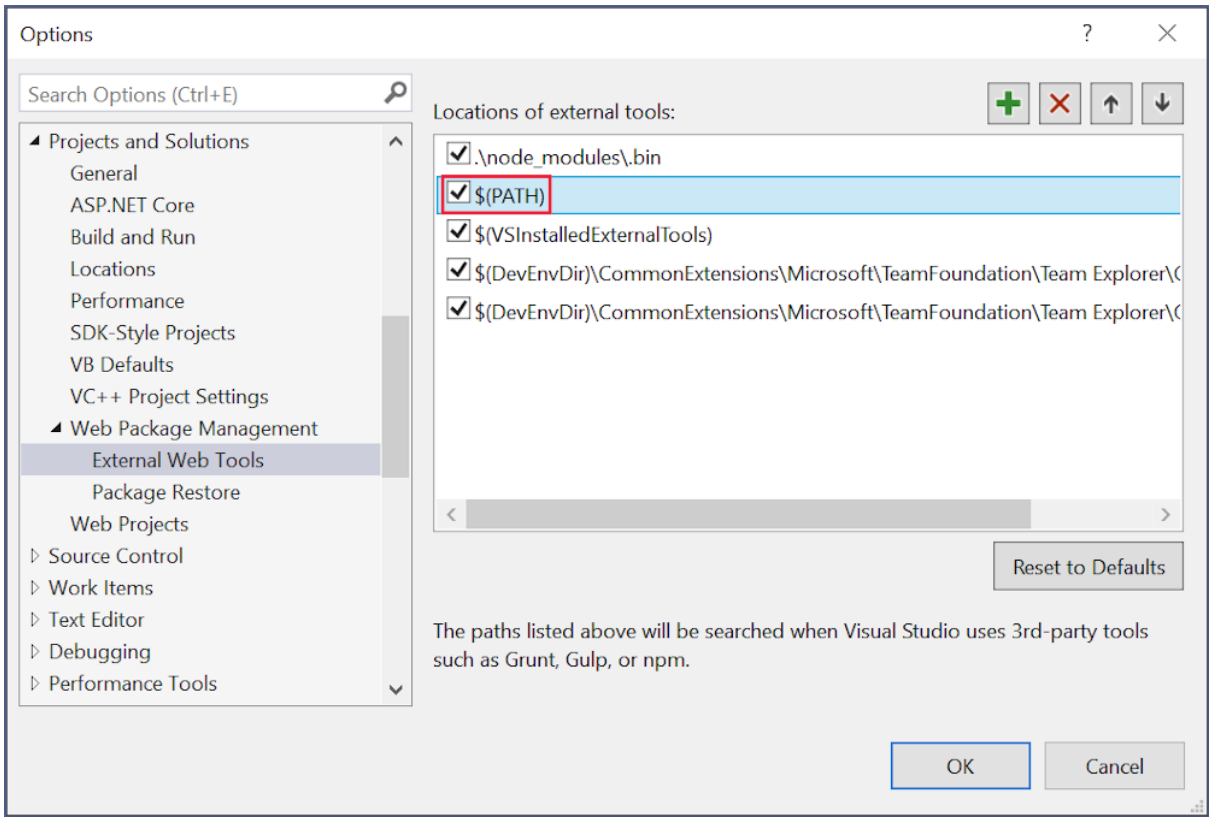
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 3.0 or later
- Node.js with npm

Create the ASP.NET Core web app

- Visual Studio
- Visual Studio Code

Configure Visual Studio to look for npm in the *PATH* environment variable. By default, Visual Studio uses the version of npm found in its installation directory. Follow these instructions in Visual Studio:

1. Launch Visual Studio. At the start window, select **Continue without code**.
2. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.
3. Select the *$(PATH)* entry from the list. Click the up arrow to move the entry to the second position in the list, and select **OK**.



Visual Studio configuration is complete.

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Web Application** template. Select **Next**.
2. Name the project *SignalRWebPack*, and select **Create**.
3. Select *.NET Core* from the target framework drop-down, and select *ASP.NET Core 3.1* from the framework selector drop-down. Select the **Empty** template, and select **Create**.

Add the `Microsoft.TypeScript.MSBuild` package to the project:

1. In **Solution Explorer** (right pane), right-click the project node and select **Manage NuGet Packages**. In the **Browse** tab, search for `Microsoft.TypeScript.MSBuild`, and then click **Install** on the right to install the package.

Visual Studio adds the NuGet package under the **Dependencies** node in **Solution Explorer**, enabling TypeScript compilation in the project.

Configure Webpack and TypeScript

The following steps configure the conversion of TypeScript to JavaScript and the bundling of client-side resources.

1. Run the following command in the project root to create a *package.json* file:

```
npm init -y
```

2. Add the highlighted property to the *package.json* file and save the file changes:

```
{
  "name": "SignalRWebPack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Setting the `private` property to `true` prevents package installation warnings in the next step.

3. Install the required npm packages. Run the following command from the project root:

```
npm i -D -E clean-webpack-plugin@3.0.0 css-loader@3.4.2 html-webpack-plugin@3.2.0 mini-css-extract-plugin@0.9.0 ts-loader@6.2.1 typescript@3.7.5 webpack@4.41.5 webpack-cli@3.3.10
```

Some command details to note:

- A version number follows the `@` sign for each package name. npm installs those specific package versions.
- The `-E` option disables npm's default behavior of writing semantic versioning range operators to *package.json*. For example, `"webpack": "4.41.5"` is used instead of `"webpack": "^4.41.5"`. This option prevents unintended upgrades to newer package versions.

See the npm-install docs for more detail.

4. Replace the `scripts` property of the *package.json* file with the following code:

```
"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

Some explanation of the scripts:

- `build`: Bundles the client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The `mode` option disables production optimizations, such as tree shaking and minification. Only use `build` in development.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET Core CLI's publish command to publish the app.

5. Create a file named *webpack.config.js*, in the project root, with the following code:

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/"
    },
    resolve: {
        extensions: [".js", ".ts"]
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader"
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"]
            }
        ]
    },
    plugins: [
        new CleanWebpackPlugin(),
        new HtmlWebpackPlugin({
            template: "./src/index.html"
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css"
        })
    ]
};
```

The preceding file configures the Webpack compilation. Some configuration details to note:

- The `output` property overrides the default value of *dist*. The bundle is instead emitted in the *wwwroot* directory.
- The `resolve.extensions` array includes *.js* to import the SignalR client JavaScript.

6. Create a new *src* directory in the project root to store the project's client-side assets.

7. Create *src/index.html* with the following markup.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR</title>
</head>
<body>
    <div id="divMessages" class="messages">
    </div>
    <div class="input-zone">
        <label id="lblMessage" for="tbMessage">Message:</label>
        <input id="tbMessage" class="input-zone-input" type="text" />
        <button id="btnSend">Send</button>
    </div>
</body>
</html>
```

The preceding HTML defines the homepage's boilerplate markup.

8. Create a new *src/css* directory. Its purpose is to store the project's *.css* files.

9. Create *src/css/main.css* with the following CSS:

```
*, *::before, *::after {
    box-sizing: border-box;
}

html, body {
    margin: 0;
    padding: 0;
}

.input-zone {
    align-items: center;
    display: flex;
    flex-direction: row;
    margin: 10px;
}

.input-zone-input {
    flex: 1;
    margin-right: 10px;
}

.message-author {
    font-weight: bold;
}

.messages {
    border: 1px solid #000;
    margin: 10px;
    max-height: 300px;
    min-height: 300px;
    overflow-y: auto;
    padding: 5px;
}
```

The preceding *main.css* file styles the app.

10. Create *src/tsconfig.json* with the following JSON:

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

The preceding code configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript.

11. Create *src/index.ts* with the following code:

```
import "./css/main.css";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.key === "Enter") {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding TypeScript retrieves references to DOM elements and attaches two event handlers:

- `keyup`: This event fires when the user types in the `tbMessage` textbox. The `send` function is called when the user presses the **Enter** key.
- `click`: This event fires when the user clicks the **Send** button. The `send` function is called.

Configure the app

1. In `Startup.Configure`, add calls to UseDefaultFiles and UseStaticFiles.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();
    app.UseDefaultFiles();
    app.UseStaticFiles();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/hub");
    });

}
```

The preceding code allows the server to locate and serve the *index.html* file. The file is served whether the user enters its full URL or the root URL of the web app.

2. At the end of `Startup.Configure`, map a */hub* route to the `ChatHub` hub. Replace the code that displays *Hello World!* with the following line:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>("/hub");
});
```

3. In `Startup.ConfigureServices`, call AddSignalR.

```
services.AddSignalR();
```

4. Create a new directory named *Hubs* in the project root *SignalRWebPack/* to store the SignalR hub.

5. Create hub *Hubs/ChatHub.cs* with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
    }
}
```

6. Add the following `using` statement at the top of the *Startup.cs* file to resolve the `ChatHub` reference:

```
using SignalRWebPack.Hubs;
```

Enable client and server communication

The app currently displays a basic form to send messages, but is not yet functional. The server is listening to a specific route but does nothing with sent messages.

1. Run the following command at the project root:

```
npm i @microsoft/signalr @types/node
```

   The preceding command installs:

   - The SignalR TypeScript client, which allows the client to send messages to the server.
   - The TypeScript type definitions for Node.js, which enables compile-time checking of Node.js types.

2. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@microsoft/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let m = document.createElement("div");

    m.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(m);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.key === "Enter") {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding code supports receiving messages from the server. The `HubConnectionBuilder` class creates a new builder for configuring the server connection. The `withUrl` function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. Any number of message names can be listened to. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new `div` element is created with the author's name and the message content in its `innerHTML` attribute. It's added to the main `div` element displaying the messages.

3. Now that the client can receive a message, configure it to send messages. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@microsoft/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let messages = document.createElement("div");

    messages.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(messages);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.key === "Enter") {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
    connection.send("newMessage", username, tbMessage.value)
        .then(() => tbMessage.value = "");
}
```

Sending a message through the WebSockets connection requires calling the `send` method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as `newMessage` is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

4. Add the `NewMessage` method to the `ChatHub` class:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
        public async Task NewMessage(long username, string message)
        {
            await Clients.All.SendAsync("messageReceived", username, message);
        }
    }
}
```

The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic `on` method to receive all the messages. A method named after the message name suffices.

In this example, the TypeScript client sends a message identified as `newMessage`. The C# `NewMessage` method

expects the data sent by the client. A call is made to SendAsync on Clients.All. The received messages are sent to all clients connected to the hub.

Test the app

Confirm that the app works with the following steps.

- Visual Studio
- Visual Studio Code

1. Run Webpack in *release* mode. Using the **Package Manager Console** window, run the following command in the project root. If you are not in the project root, enter `cd SignalRWebPack` before entering the command.

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the *wwwroot* folder.

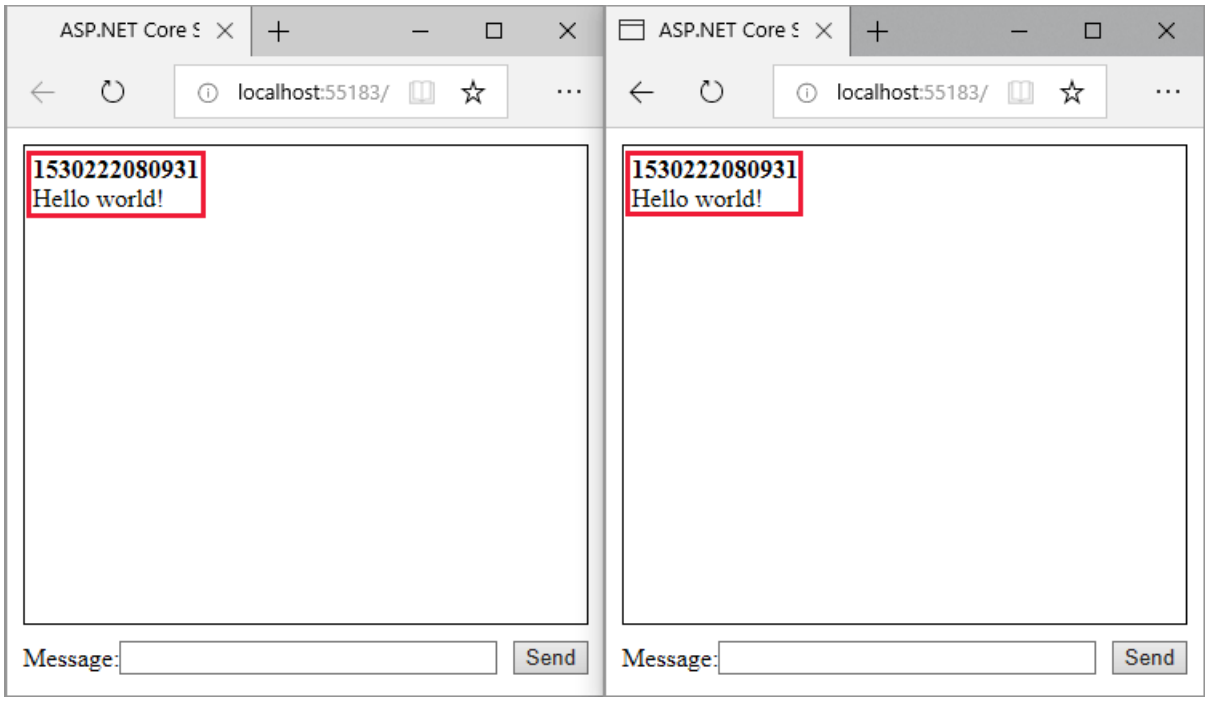
   Webpack completed the following tasks:

   - Purged the contents of the *wwwroot* directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from *src* to the *wwwroot* directory.
   - Injected the following elements into the *wwwroot/index.html* file:
     - A `<link>` tag, referencing the *wwwroot/main.<hash>.css* file. This tag is placed immediately before the closing `</head>` tag.
     - A `<script>` tag, referencing the minified *wwwroot/main.<hash>.js* file. This tag is placed immediately before the closing `</body>` tag.
2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The *wwwroot/index.html* file is served at `http://localhost:<port_number>`.

   If you get compile errors, try closing and reopening the solution.

3. Open another browser instance (any browser). Paste the URL in the address bar.

4. Choose either browser, type something in the **Message** text box, and click the **Send** button. The unique user name and message are displayed on both pages instantly.



Prerequisites

- Visual Studio
- Visual Studio Code

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later
- Node.js with npm

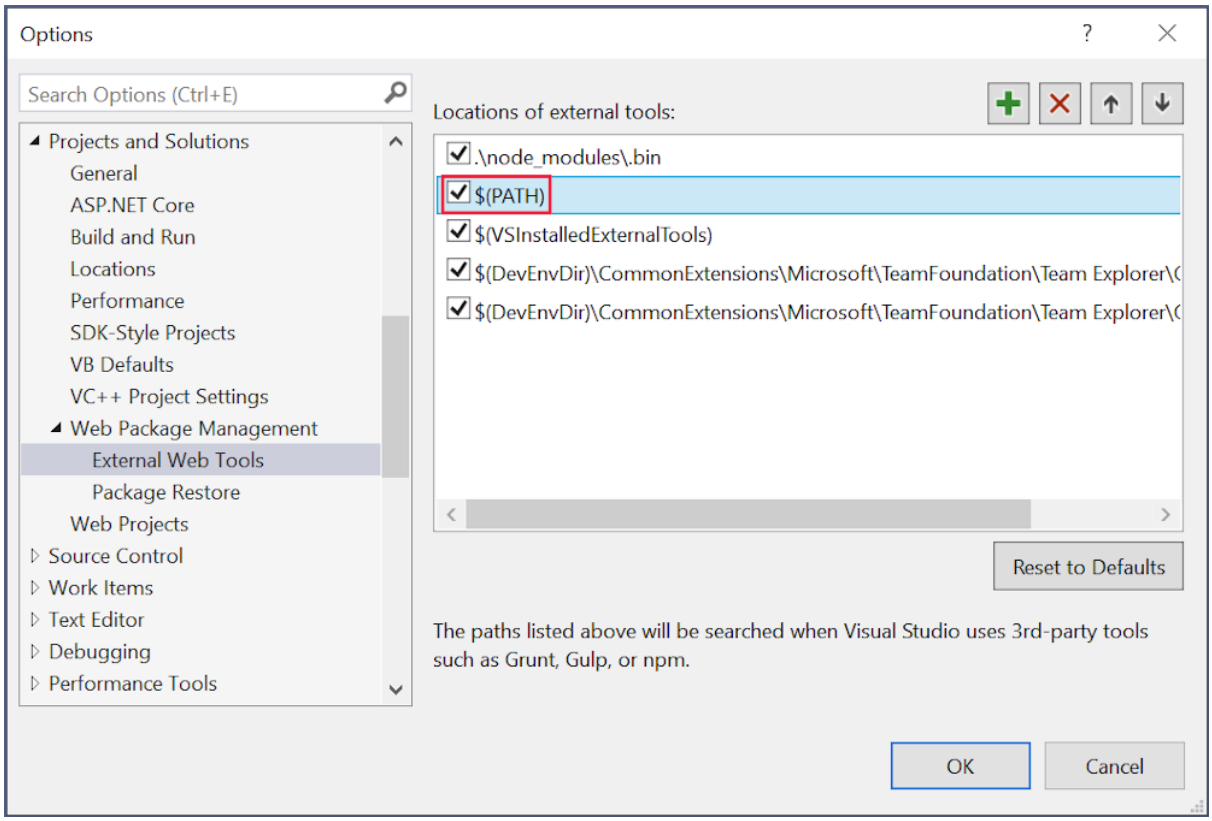
Create the ASP.NET Core web app

- Visual Studio
- Visual Studio Code

Configure Visual Studio to look for npm in the *PATH* environment variable. By default, Visual Studio uses the version of npm found in its installation directory. Follow these instructions in Visual Studio:

1. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.

2. Select the *$(PATH)* entry from the list. Click the up arrow to move the entry to the second position in the list.



Visual Studio configuration is completed. It's time to create the project.

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Web Application** template.
2. Name the project *SignalRWebPack*, and select **Create**.
3. Select *.NET Core* from the target framework drop-down, and select *ASP.NET Core 2.2* from the framework selector drop-down. Select the **Empty** template, and select **Create**.

Configure Webpack and TypeScript

The following steps configure the conversion of TypeScript to JavaScript and the bundling of client-side resources.

1. Run the following command in the project root to create a *package.json* file:

```
npm init -y
```

2. Add the highlighted property to the *package.json* file:

```
{
  "name": "SignalRWebPack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

   Setting the `private` property to `true` prevents package installation warnings in the next step.

3. Install the required npm packages. Run the following command from the project root:

```
npm install -D -E clean-webpack-plugin@1.0.1 css-loader@2.1.0 html-webpack-plugin@4.0.0-beta.5 mini-css-extract-plugin@0.5.0 ts-loader@5.3.3 typescript@3.3.3 webpack@4.29.3 webpack-cli@3.2.3
```

Some command details to note:

- A version number follows the `@` sign for each package name. npm installs those specific package versions.
- The `-E` option disables npm's default behavior of writing semantic versioning range operators to *package.json*. For example, `"webpack": "4.29.3"` is used instead of `"webpack": "^4.29.3"`. This option prevents unintended upgrades to newer package versions.

See the npm-install docs for more detail.

4. Replace the `scripts` property of the *package.json* file with the following code:

```
"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

Some explanation of the scripts:

- `build`: Bundles the client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The `mode` option disables production optimizations, such as tree shaking and minification. Only use `build` in development.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET Core CLI's publish command to publish the app.

5. Create a file named *webpack.config.js* in the project root, with the following code:

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const CleanWebpackPlugin = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/"
    },
    resolve: {
        extensions: [".js", ".ts"]
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader"
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"]
            }
        ]
    },
    plugins: [
        new CleanWebpackPlugin(["wwwroot/*"]),
        new HtmlWebpackPlugin({
            template: "./src/index.html"
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css"
        })
    ]
};
```

The preceding file configures the Webpack compilation. Some configuration details to note:

- The `output` property overrides the default value of *dist*. The bundle is instead emitted in the *wwwroot* directory.
- The `resolve.extensions` array includes *.js* to import the SignalR client JavaScript.

6. Create a new *src* directory in the project root to store the project's client-side assets.

7. Create *src/index.html* with the following markup.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR</title>
</head>
<body>
    <div id="divMessages" class="messages">
    </div>
    <div class="input-zone">
        <label id="lblMessage" for="tbMessage">Message:</label>
        <input id="tbMessage" class="input-zone-input" type="text" />
        <button id="btnSend">Send</button>
    </div>
</body>
</html>
```

The preceding HTML defines the homepage's boilerplate markup.

8. Create a new *src/css* directory. Its purpose is to store the project's *.css* files.

9. Create *src/css/main.css* with the following markup:

```
*, *::before, *::after {
    box-sizing: border-box;
}

html, body {
    margin: 0;
    padding: 0;
}

.input-zone {
    align-items: center;
    display: flex;
    flex-direction: row;
    margin: 10px;
}

.input-zone-input {
    flex: 1;
    margin-right: 10px;
}

.message-author {
    font-weight: bold;
}

.messages {
    border: 1px solid #000;
    margin: 10px;
    max-height: 300px;
    min-height: 300px;
    overflow-y: auto;
    padding: 5px;
}
```

The preceding *main.css* file styles the app.

10. Create *src/tsconfig.json* with the following JSON:

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

The preceding code configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript.

11. Create *src/index.ts* with the following code:

```
import "./css/main.css";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding TypeScript retrieves references to DOM elements and attaches two event handlers:

- `keyup`: This event fires when the user types in the `tbMessage` textbox. The `send` function is called when the user presses the **Enter** key.
- `click`: This event fires when the user clicks the **Send** button. The `send` function is called.

Configure the ASP.NET Core app

1. The code provided in the `Startup.Configure` method displays *Hello World!*. Replace the `app.Run` method call with calls to UseDefaultFiles and UseStaticFiles.

   ```
   app.UseDefaultFiles();
   app.UseStaticFiles();
   ```

   The preceding code allows the server to locate and serve the *index.html* file, whether the user enters its full URL or the root URL of the web app.

2. Call AddSignalR in `Startup.ConfigureServices`. It adds the SignalR services to the project.

   ```
   services.AddSignalR();
   ```

3. Map a */hub* route to the `ChatHub` hub. Add the following lines at the end of `Startup.Configure`:

   ```
   app.UseSignalR(options =>
   {
       options.MapHub<ChatHub>("/hub");
   });
   ```

4. Create a new directory, called *Hubs*, in the project root. Its purpose is to store the SignalR hub, which is created in the next step.

5. Create hub *Hubs/ChatHub.cs* with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
    }
}
```

6. Add the following code at the top of the *Startup.cs* file to resolve the `ChatHub` reference:

```
using SignalRWebPack.Hubs;
```

Enable client and server communication

The app currently displays a simple form to send messages. Nothing happens when you try to do so. The server is listening to a specific route but does nothing with sent messages.

1. Run the following command at the project root:

```
npm install @aspnet/signalr
```

   The preceding command installs the SignalR TypeScript client, which allows the client to send messages to the server.

2. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let m = document.createElement("div");

    m.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(m);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding code supports receiving messages from the server. The `HubConnectionBuilder` class creates a new builder for configuring the server connection. The `withUrl` function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. You can listen to any number of message names. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new `div` element is created with the author's name and the message content in its `innerHTML` attribute. The new message is added to the main `div` element displaying the messages.

3. Now that the client can receive a message, configure it to send messages. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let messageContainer = document.createElement("div");

    messageContainer.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(messageContainer);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
    connection.send("newMessage", username, tbMessage.value)
              .then(() => tbMessage.value = "");
}
```

Sending a message through the WebSockets connection requires calling the `send` method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as `newMessage` is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

4. Add the `NewMessage` method to the `ChatHub` class:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
        public async Task NewMessage(long username, string message)
        {
            await Clients.All.SendAsync("messageReceived", username, message);
        }
    }
}
```

The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic `on` method to receive all the messages. A method named after the message name suffices.

In this example, the TypeScript client sends a message identified as `newMessage`. The C# `NewMessage` method

expects the data sent by the client. A call is made to SendAsync on Clients.All. The received messages are sent to all clients connected to the hub.

Test the app

Confirm that the app works with the following steps.

- Visual Studio
- Visual Studio Code

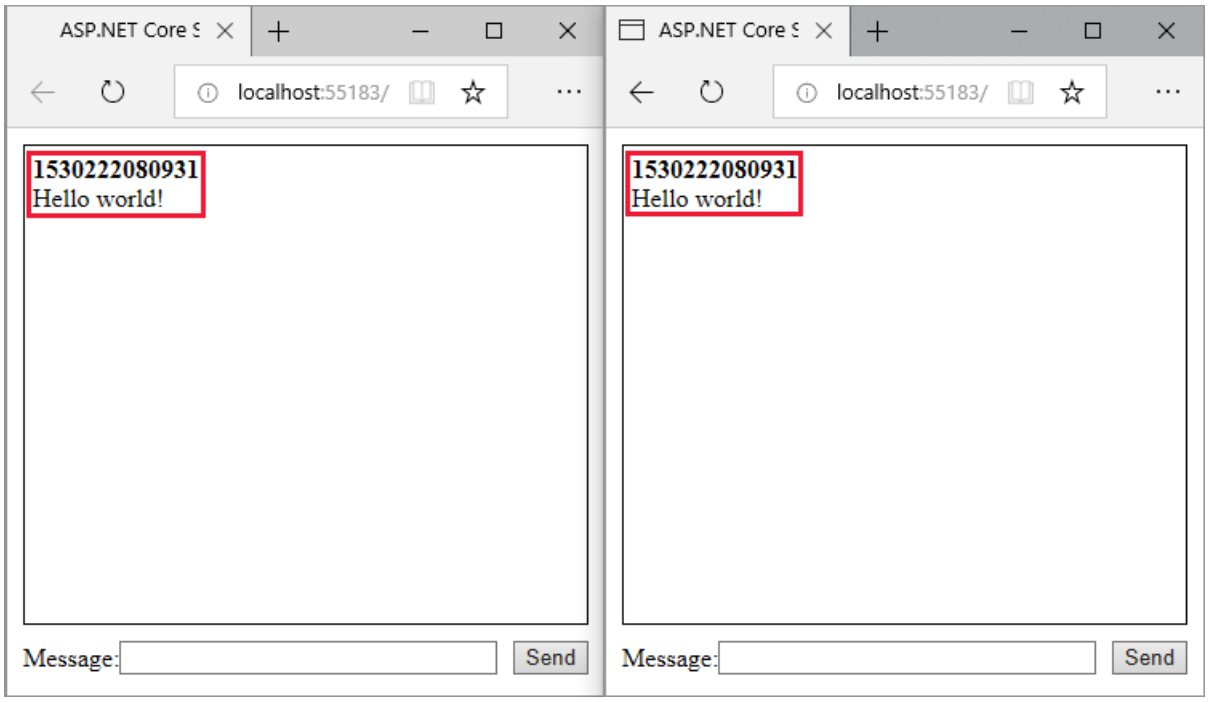
1. Run Webpack in *release* mode. Using the **Package Manager Console** window, run the following command in the project root. If you are not in the project root, enter `cd SignalRWebPack` before entering the command.

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the *wwwroot* folder.

   Webpack completed the following tasks:

   - Purged the contents of the *wwwroot* directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from *src* to the *wwwroot* directory.
   - Injected the following elements into the *wwwroot/index.html* file:
     - A `<link>` tag, referencing the *wwwroot/main.<hash>.css* file. This tag is placed immediately before the closing `</head>` tag.
     - A `<script>` tag, referencing the minified *wwwroot/main.<hash>.js* file. This tag is placed immediately before the closing `</body>` tag.
2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The *wwwroot/index.html* file is served at `http://localhost:<port_number>`.
3. Open another browser instance (any browser). Paste the URL in the address bar.
4. Choose either browser, type something in the **Message** text box, and click the **Send** button. The unique user name and message are displayed on both pages instantly.



Additional resources

- ASP.NET Core SignalR JavaScript client
- Use hubs in ASP.NET Core SignalR

Use ASP.NET Core SignalR with Blazor WebAssembly

9/22/2020 • 7 minutes to read • Edit Online

By Daniel Roth and Luke Latham

This tutorial teaches the basics of building a real-time app using SignalR with Blazor WebAssembly. You learn how to:

- Create a Blazor WebAssembly Hosted app project
- Add the SignalR client library
- Add a SignalR hub
- Add SignalR services and an endpoint for the SignalR hub
- Add Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

- Visual Studio 2019 16.6 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a hosted Blazor WebAssembly app project

Follow the guidance for your choice of tooling:

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

> **NOTE**
> Visual Studio 16.6 or later and .NET Core SDK 3.1.300 or later are required.

1. Create a new project.
2. Select **Blazor App** and select **Next**.
3. Type `BlazorSignalRApp` in the **Project name** field. Confirm the **Location** entry is correct or provide a location for the project. Select **Create**.
4. Choose the **Blazor WebAssembly App** template.
5. Under **Advanced**, select the **ASP.NET Core hosted** check box.

6. Select **Create**.

Add the SignalR client library

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

1. In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.
2. In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.
3. With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.
4. In the search results, select the `Microsoft.AspNetCore.SignalR.Client` package and select **Install**.
5. If the **Preview Changes** dialog appears, select **OK**.
6. If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the `BlazorSignalRApp.Server` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Server.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

Add services and an endpoint for the SignalR hub

1. In the `BlazorSignalRApp.Server` project, open the `Startup.cs` file.
2. Add the namespace for the `ChatHub` class to the top of the file:

   ```
   using BlazorSignalRApp.Server.Hubs;
   ```

3. Add SignalR and Response Compression Middleware services to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR();
    services.AddControllersWithViews();
    services.AddResponseCompression(opts =>
    {
        opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "application/octet-stream" });
    });
}
```

4. In `Startup.Configure`:

   - Use Response Compression Middleware at the top of the processing pipeline's configuration.
   - Between the endpoints for controllers and the client-side fallback, add an endpoint for the hub.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseResponseCompression();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseWebAssemblyDebugging();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseBlazorFrameworkFiles();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapHub<ChatHub>("/chathub");
        endpoints.MapFallbackToFile("index.html");
    });
}
```

Add Razor component code for chat

1. In the `BlazorSignalRApp.Client` project, open the `Pages/Index.razor` file.

2. Replace the markup with the following code:

```
@page "/"
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager NavigationManager
@implements IDisposable

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection hubConnection;
    private List<string> messages = new List<string>();
    private string userInput;
    private string messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(NavigationManager.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message) =>
        {
            var encodedMsg = $"{user}: {message}";
            messages.Add(encodedMsg);
            StateHasChanged();
        });

        await hubConnection.StartAsync();
    }

    Task Send() =>
        hubConnection.SendAsync("SendMessage", userInput, messageInput);

    public bool IsConnected =>
        hubConnection.State == HubConnectionState.Connected;

    public void Dispose()
    {
        _ = hubConnection.DisposeAsync();
    }
}
```

Run the app

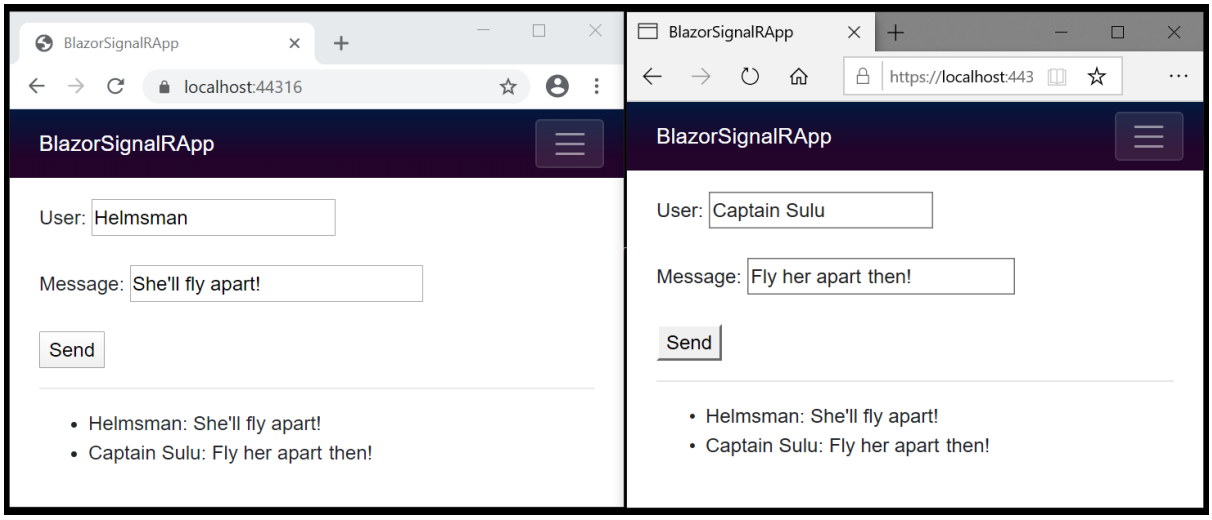
1. Follow the guidance for your tooling:

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

1. In **Solution Explorer**, select the `BlazorSignalRApp.Server` project. Press `F5` to run the app with debugging or `Ctrl+F5` to run the app without debugging.

2. Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

3. Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* © 1991 Paramount

Next steps

In this tutorial, you learned how to:

- Create a Blazor WebAssembly Hosted app project
- Add the SignalR client library
- Add a SignalR hub
- Add SignalR services and an endpoint for the SignalR hub
- Add Razor component code for chat

To learn more about building Blazor apps, see the Blazor documentation:

Introduction to ASP.NET Core Blazor

Additional resources

- Introduction to ASP.NET Core SignalR
- SignalR cross-origin negotiation for authentication

Tutorial: Create a gRPC client and server in ASP.NET Core

9/22/2020 • 9 minutes to read • Edit Online

By John Luo

This tutorial shows how to create a .NET Core gRPC client and an ASP.NET Core gRPC Server.

At the end, you'll have a gRPC client that communicates with the gRPC Greeter service.

View or download sample code (how to download).

In this tutorial, you:

- Create a gRPC Server.
- Create a gRPC client.
- Test the gRPC client service with the gRPC Greeter service.

Prerequisites

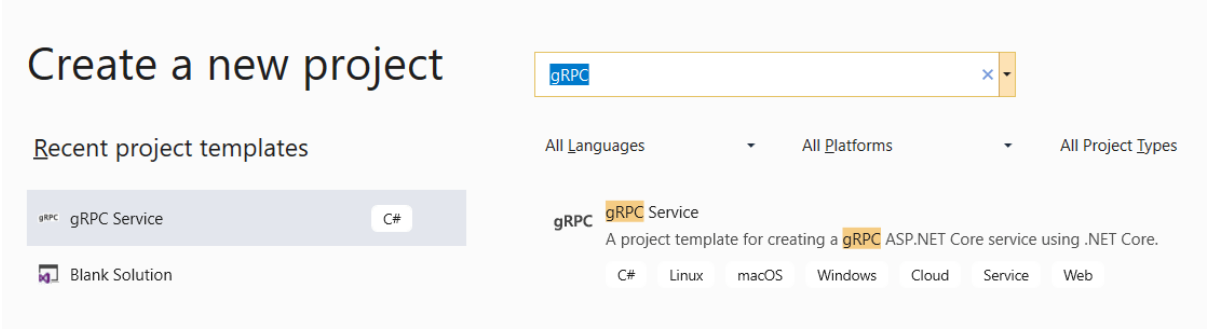
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a gRPC service

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Start Visual Studio and select **Create a new project**. Alternatively, from the Visual Studio **File** menu, select **New** > **Project**.
- In the **Create a new project** dialog, select **gRPC Service** and select **Next**:



- Name the project **GrpcGreeter**. It's important to name the project *GrpcGreeter* so the namespaces will match when you copy and paste code.
- Select **Create**.

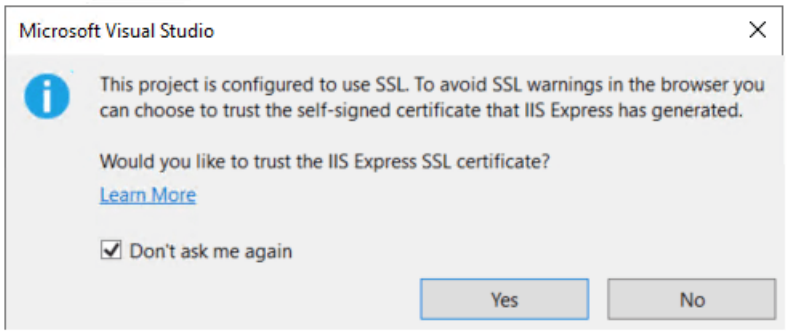
- In the **Create a new gRPC service** dialog:
  - The **gRPC Service** template is selected.
  - Select **Create**.

Run the service

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

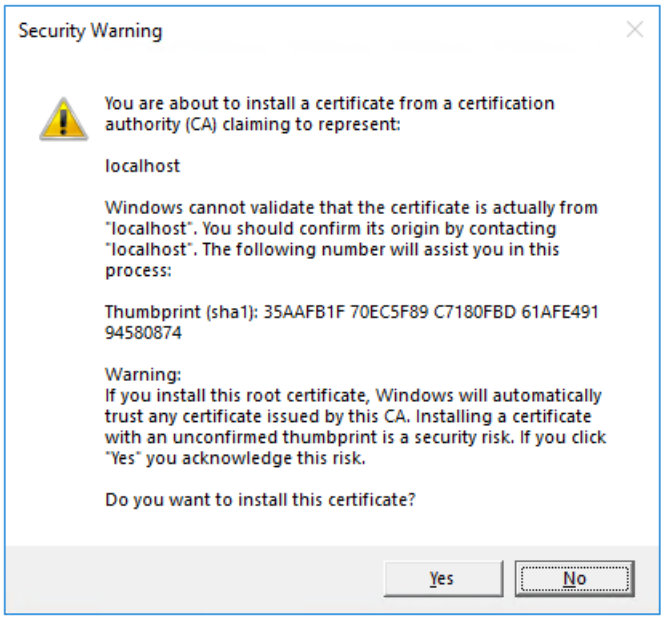
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog:



  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:



  Select **Yes** if you agree to trust the development certificate.

  Visual Studio starts IIS Express and runs the app. The address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for the local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

The logs show the service listening on `https://localhost:5001`.

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

> **NOTE**
>
> The gRPC template is configured to use Transport Layer Security (TLS). gRPC clients need to use HTTPS to call the server.
>
> macOS doesn't support ASP.NET Core gRPC with TLS. Additional configuration is required to successfully run gRPC services on macOS. For more information, see Unable to start ASP.NET Core gRPC app on macOS.

Examine the project files

GrpcGreeter project files:

- *greet.proto*: The *Protos/greet.proto* file defines the `Greeter` gRPC and is used to generate the gRPC server assets. For more information, see Introduction to gRPC.
- *Services* folder: Contains the implementation of the `Greeter` service.
- *appSettings.json*: Contains configuration data, such as protocol used by Kestrel. For more information, see Configuration in ASP.NET Core.
- *Program.cs*: Contains the entry point for the gRPC service. For more information, see .NET Generic Host.
- *Startup.cs*: Contains code that configures app behavior. For more information, see App startup.

Create the gRPC client in a .NET console app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Open a second instance of Visual Studio and select **Create a new project**.
- In the **Create a new project** dialog, select **Console App (.NET Core)** and select **Next**.
- In the **Project name** text box, enter **GrpcGreeterClient** and select **Create**.

Add required packages

The gRPC client project requires the following packages:

- Grpc.Net.Client, which contains the .NET Core client.
- Google.Protobuf, which contains protobuf message APIs for C#.
- Grpc.Tools, which contains C# tooling support for protobuf files. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Install the packages using either the Package Manager Console (PMC) or Manage NuGet Packages.

PMC option to install packages

- From Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**
- From the **Package Manager Console** window, run `cd GrpcGreeterClient` to change directories to the folder containing the *GrpcGreeterClient.csproj* files.

- Run the following commands:

```
Install-Package Grpc.Net.Client
Install-Package Google.Protobuf
Install-Package Grpc.Tools
```

Manage NuGet Packages option to install packages

- Right-click the project in **Solution Explorer** > **Manage NuGet Packages**
- Select the **Browse** tab.
- Enter **Grpc.Net.Client** in the search box.
- Select the **Grpc.Net.Client** package from the **Browse** tab and select **Install**.
- Repeat for `Google.Protobuf` and `Grpc.Tools`.

Add greet.proto

- Create a *Protos* folder in the gRPC client project.
- Copy the *Protos\greet.proto* file from the gRPC Greeter service to the gRPC client project.
- Edit the *GrpcGreeterClient.csproj* project file:
  - Visual Studio
  - Visual Studio Code
  - Visual Studio for Mac

  Right-click the project and select **Edit Project File**.

- Add an item group with a `<Protobuf>` element that refers to the *greet.proto* file:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Create the Greeter client

Build the project to create the types in the `GrpcGreeter` namespace. The `GrpcGreeter` types are generated automatically by the build process.

Update the gRPC client *Program.cs* file with the following code:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using GrpcGreeter;
using Grpc.Net.Client;

namespace GrpcGreeterClient
{
    class Program
    {
        static async Task Main(string[] args)
        {
            // The port number(5001) must match the port of the gRPC server.
            using var channel = GrpcChannel.ForAddress("https://localhost:5001");
            var client =  new Greeter.GreeterClient(channel);
            var reply = await client.SayHelloAsync(
                              new HelloRequest { Name = "GreeterClient" });
            Console.WriteLine("Greeting: " + reply.Message);
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

Program.cs contains the entry point and logic for the gRPC client.

The Greeter client is created by:

- Instantiating a `GrpcChannel` containing the information for creating the connection to the gRPC service.
- Using the `GrpcChannel` to construct the Greeter client:

```
static async Task Main(string[] args)
{
    // The port number(5001) must match the port of the gRPC server.
    using var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client =  new Greeter.GreeterClient(channel);
    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
}
```

The Greeter client calls the asynchronous `SayHello` method. The result of the `SayHello` call is displayed:

```
static async Task Main(string[] args)
{
    // The port number(5001) must match the port of the gRPC server.
    using var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client =  new Greeter.GreeterClient(channel);
    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
}
```

Test the gRPC client with the gRPC Greeter service

- Visual Studio

- Visual Studio Code
- Visual Studio for Mac

- In the Greeter service, press `Ctrl+F5` to start the server without the debugger.
- In the `GrpcGreeterClient` project, press `Ctrl+F5` to start the client without the debugger.

The client sends a greeting to the service with a message containing its name, *GreeterClient*. The service sends the message "Hello GreeterClient" as a response. The "Hello GreeterClient" response is displayed in the command prompt:

```
Greeting: Hello GreeterClient
Press any key to exit...
```

The gRPC service records the details of the successful call in the logs written to the command prompt:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\GH\aspnet\docs\4\Docs\aspnetcore\tutorials\grpc\grpc-start\sample\GrpcGreeter
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5001/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 78.32260000000001ms 200 application/grpc
```

> **NOTE**
>
> The code in this article requires the ASP.NET Core HTTPS development certificate to secure the gRPC service. If the .NET gRPC client fails with the message `The remote certificate is invalid according to the validation procedure.` or `The SSL connection could not be established.`, the development certificate isn't trusted. To fix this issue, see Call a gRPC service with an untrusted/invalid certificate.

> **WARNING**
>
> ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this GitHub issue.

Next steps

- Introduction to gRPC on .NET Core
- gRPC services with C#
- Migrating gRPC services from C-core to ASP.NET Core

Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8

9/22/2020 • 50 minutes to read • Edit Online

By Tom Dykstra and Rick Anderson

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.
- Visual Studio
- Visual Studio Code
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK or later

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

The Visual Studio Code instructions use SQLite, a cross-platform database engine.

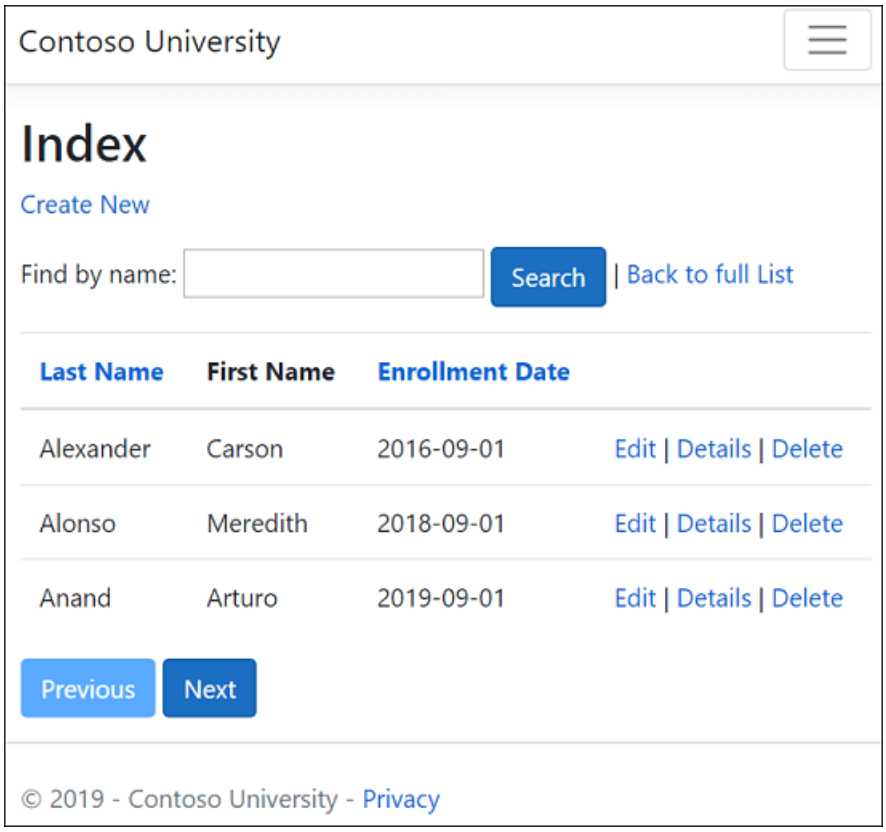
If you choose to use SQLite, download and install a third-party tool for managing and viewing a SQLite database, such as DB Browser for SQLite.

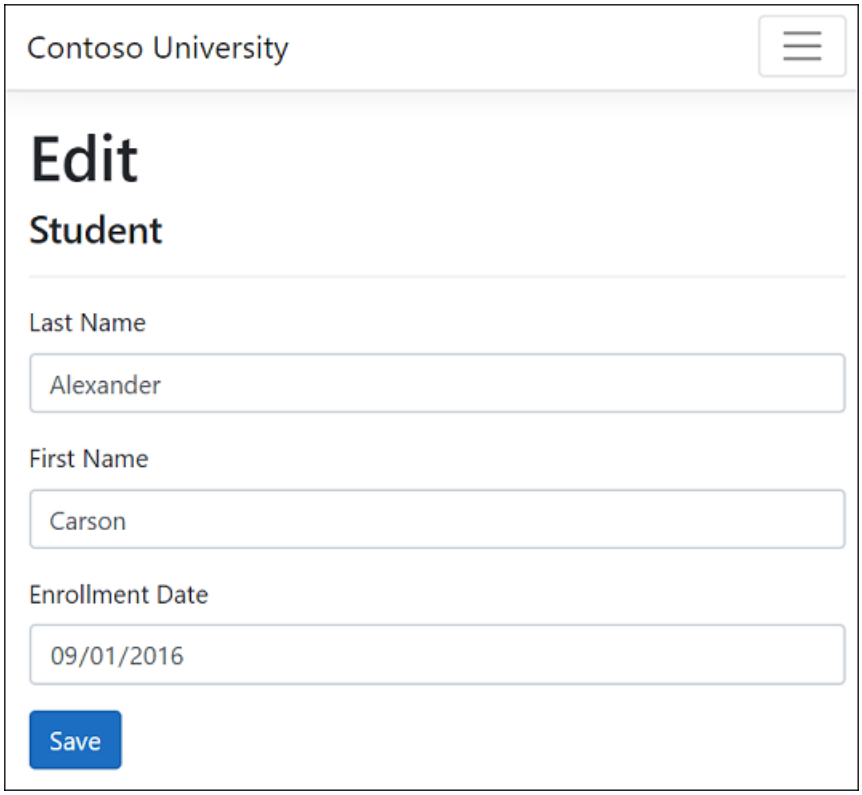
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core, not how to customize the UI.

Follow the link at the top of the page to get the source code for the completed project. The *cu30* folder has the code for the ASP.NET Core 3.0 version of the tutorial. Files that reflect the state of the code for tutorials 1-7 can be found in the *cu30snapshots* folder.

- Visual Studio
- Visual Studio Code

To run the app after downloading the completed project:

- Build the project.
- In Package Manager Console (PMC) run the following command:

```
Update-Database
```

- Run the project to seed the database.

Create the web app project

- Visual Studio
- Visual Studio Code

- From the Visual Studio **File** menu, select **New** > **Project**.
- Select **ASP.NET Core Web Application**.
- Name the project *ContosoUniversity*. It's important to use this exact name including capitalization, so the namespaces match when code is copied and pasted.
- Select **.NET Core** and **ASP.NET Core 3.0** in the dropdowns, and then select **Web Application**.

Set up the site style

Set up the site header, footer, and menu by updating *Pages/Shared/_Layout.cshtml*:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Delete the **Home** and **Privacy** menu entries, and add entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

The changes are highlighted.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow
mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET Core with text about this app:

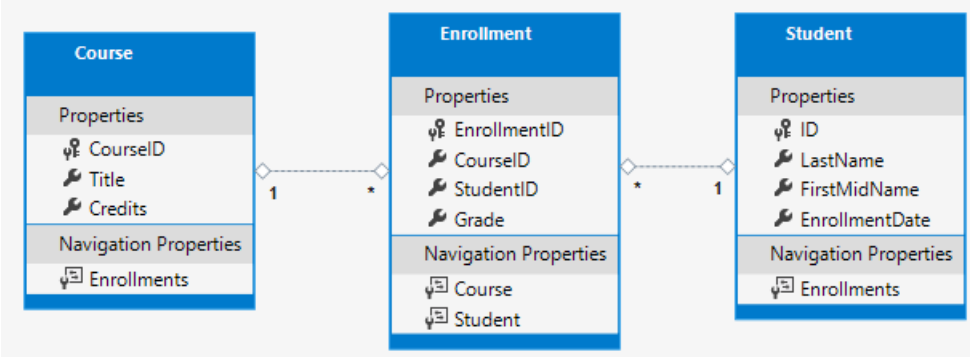
```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in a series of tutorials.
                </p>
                <p>
                    <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
                    <a href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
                </p>
            </div>
        </div>
    </div>
</div>
```

Run the app to verify that the home page appears.

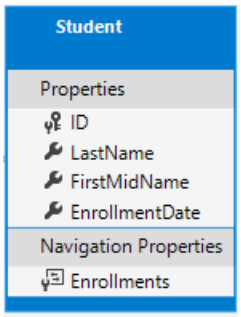
The data model

The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity



- Create a *Models* folder in the project folder.
- Create *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

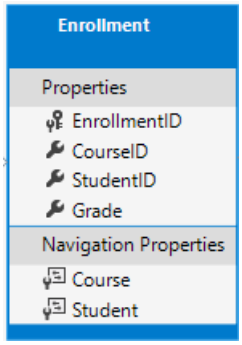
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its StudentID column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have StudentID = 1. StudentID is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. You can use other collection types, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



Create *Models/Enrollment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

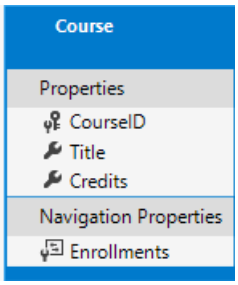
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the project to validate that there are no compiler errors.

Scaffold Student pages

In this section, you use the ASP.NET Core scaffolding tool to generate:

- An EF Core *context* class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the `Microsoft.EntityFrameworkCore.DbContext` class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

- Visual Studio
- Visual Studio Code

- Create a *Students* folder in the *Pages* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the **+** (plus) sign.
  - Change the data context name from *ContosoUniversity.Models.ContosoUniversityContext* to *ContosoUniversity.Data.SchoolContext*.
  - Select **Add**.

The following packages are automatically installed:

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.Extensions.Logging.Debug`
- `Microsoft.EntityFrameworkCore.Tools`

If you have a problem with the preceding step, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - *Create.cshtml* and *Create.cshtml.cs*
  - *Delete.cshtml* and *Delete.cshtml.cs*
  - *Details.cshtml* and *Details.cshtml.cs*
  - *Edit.cshtml* and *Edit.cshtml.cs*
  - *Index.cshtml* and *Index.cshtml.cs*
- Creates *Data/SchoolContext.cs*.
- Adds the context to dependency injection in *Startup.cs*.
- Adds a database connection string to *appsettings.json*.

Database connection string

- Visual Studio
- Visual Studio Code

The connection string specifies SQL Server LocalDB.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext6;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update *SchoolContext.cs* with the following code:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext (DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

The highlighted code creates a DbSet<TEntity> property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

Since an entity set contains multiple entities, the DBSet properties should be plural names. Since the scaffolding tool created a `Student` DBSet, this step changes it to plural `Students`.

To make the Razor Pages code match the new DBSet name, make a global change across the whole project of `_context.Student` to `_context.Students`. There are 8 occurrences.

Build the project to verify there are no compiler errors.

Startup.cs

ASP.NET Core is built with dependency injection. Services (such as the EF Core database context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- Visual Studio
- Visual Studio Code

- In `ConfigureServices`, the highlighted lines were added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Create the database

Update *Program.cs* to create the database if it doesn't exist:

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works well early in development when the schema is rapidly evolving, as long as you don't need to preserve data. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, you delete the database that was created by `EnsureCreated` and use migrations instead. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create *Data/DbInitializer.cs* with the following code:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2019-09-
01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-09-
01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-09-
01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-
01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-
01")}
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
```

```
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
            };

            context.Courses.AddRange(courses);
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In *Program.cs*, replace the `EnsureCreated` call with a `DbInitializer.Initialize` call:

```
// context.Database.EnsureCreated();
DbInitializer.Initialize(context);
```

- Visual Studio
- Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

- Restart the app.
- Select the Students page to see the seeded data.

View the database

- Visual Studio
- Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name you provided earlier plus a dash and a GUID.

- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task<T>` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

Next steps

NEXT TUTORIAL

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.
- Visual Studio
- Visual Studio Code
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK or later

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

The Visual Studio Code instructions use SQLite, a cross-platform database engine.

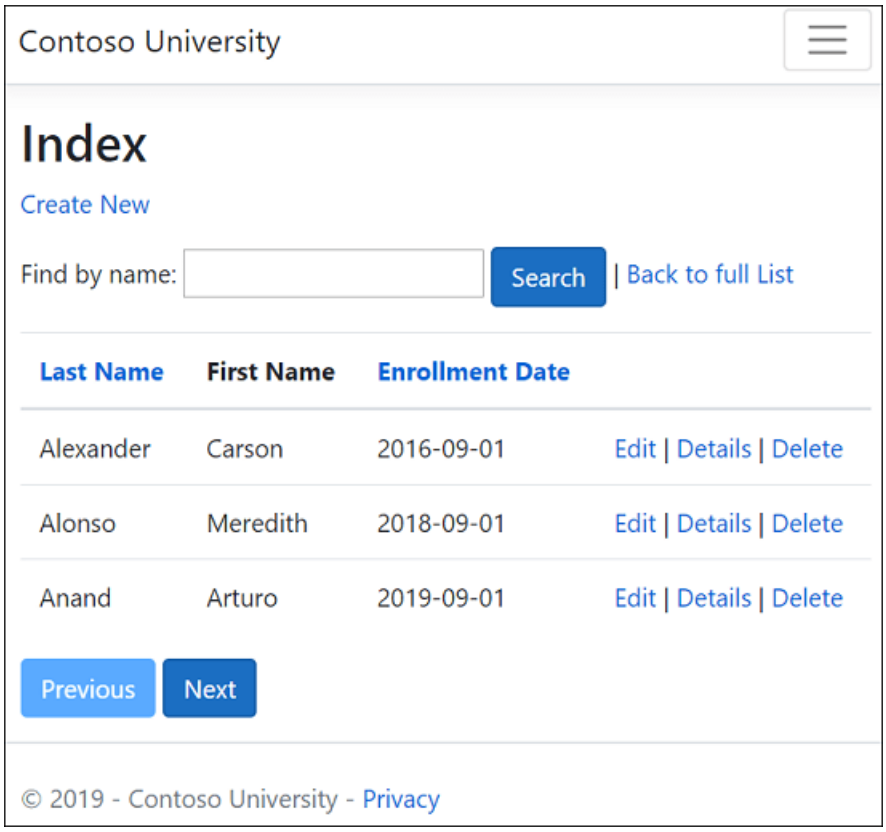
If you choose to use SQLite, download and install a third-party tool for managing and viewing a SQLite database, such as DB Browser for SQLite.

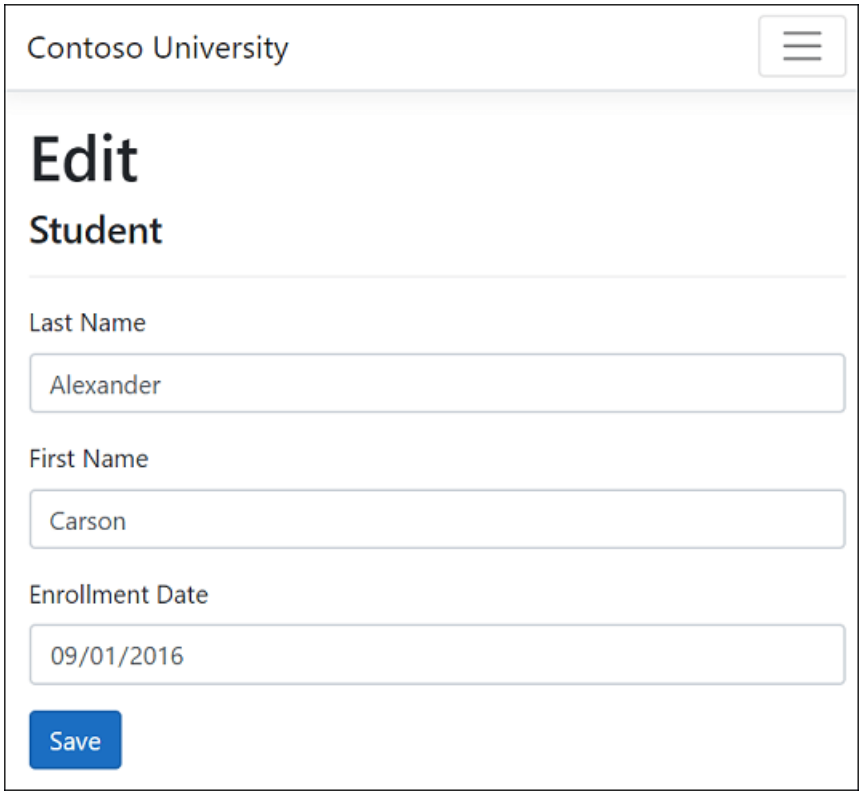
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core, not how to customize the UI.

Follow the link at the top of the page to get the source code for the completed project. The *cu30* folder has the code for the ASP.NET Core 3.0 version of the tutorial. Files that reflect the state of the code for tutorials 1-7 can be found in the *cu30snapshots* folder.

- Visual Studio
- Visual Studio Code

To run the app after downloading the completed project:

- Build the project.
- In Package Manager Console (PMC) run the following command:

```
Update-Database
```

- Run the project to seed the database.

Create the web app project

- Visual Studio
- Visual Studio Code

- From the Visual Studio **File** menu, select **New** > **Project**.
- Select **ASP.NET Core Web Application**.
- Name the project *ContosoUniversity*. It's important to use this exact name including capitalization, so the namespaces match when code is copied and pasted.
- Select **.NET Core** and **ASP.NET Core 3.0** in the dropdowns, and then select **Web Application**.

Set up the site style

Set up the site header, footer, and menu by updating *Pages/Shared/_Layout.cshtml*:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Delete the **Home** and **Privacy** menu entries, and add entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

The changes are highlighted.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow
mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET Core with text about this app:

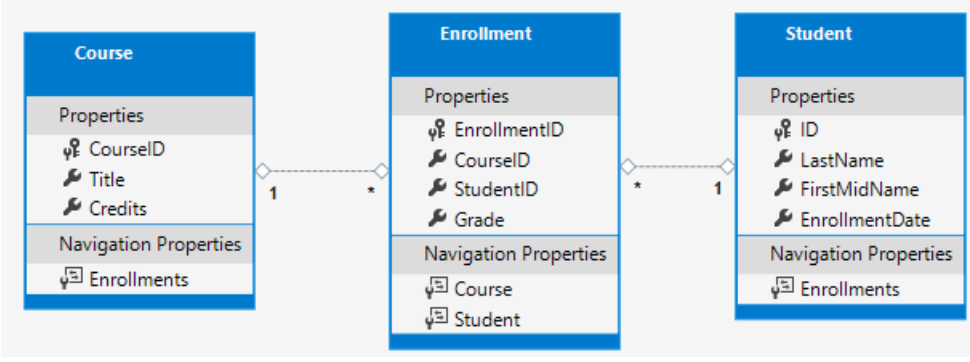
```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in a series of tutorials.
                </p>
                <p>
                    <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
                    <a href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
                </p>
            </div>
        </div>
    </div>
</div>
```

Run the app to verify that the home page appears.

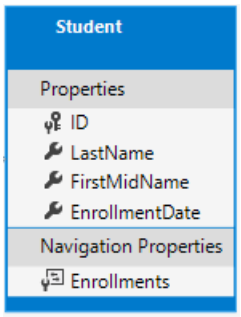
The data model

The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity



- Create a *Models* folder in the project folder.
- Create *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

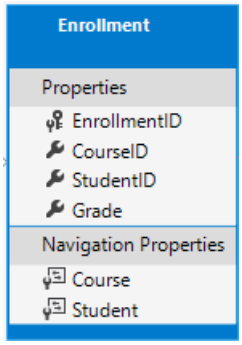
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its StudentID column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have StudentID = 1. StudentID is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. You can use other collection types, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



Create *Models/Enrollment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

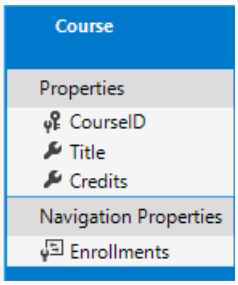
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the project to validate that there are no compiler errors.

Scaffold Student pages

In this section, you use the ASP.NET Core scaffolding tool to generate:

- An EF Core *context* class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the `Microsoft.EntityFrameworkCore.DbContext` class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

- Visual Studio
- Visual Studio Code

- Create a *Students* folder in the *Pages* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the **+** (plus) sign.
  - Change the data context name from *ContosoUniversity.Models.ContosoUniversityContext* to *ContosoUniversity.Data.SchoolContext*.
  - Select **Add**.

The following packages are automatically installed:

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.Extensions.Logging.Debug`
- `Microsoft.EntityFrameworkCore.Tools`

If you have a problem with the preceding step, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - *Create.cshtml* and *Create.cshtml.cs*
  - *Delete.cshtml* and *Delete.cshtml.cs*
  - *Details.cshtml* and *Details.cshtml.cs*
  - *Edit.cshtml* and *Edit.cshtml.cs*
  - *Index.cshtml* and *Index.cshtml.cs*
- Creates *Data/SchoolContext.cs*.
- Adds the context to dependency injection in *Startup.cs*.
- Adds a database connection string to *appsettings.json*.

Database connection string

- Visual Studio
- Visual Studio Code

The connection string specifies SQL Server LocalDB.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext6;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update *SchoolContext.cs* with the following code:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext (DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

The highlighted code creates a DbSet<TEntity> property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

Since an entity set contains multiple entities, the DBSet properties should be plural names. Since the scaffolding tool created a `Student` DBSet, this step changes it to plural `Students`.

To make the Razor Pages code match the new DBSet name, make a global change across the whole project of `_context.Student` to `_context.Students`. There are 8 occurrences.

Build the project to verify there are no compiler errors.

Startup.cs

ASP.NET Core is built with dependency injection. Services (such as the EF Core database context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- Visual Studio
- Visual Studio Code

- In `ConfigureServices`, the highlighted lines were added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Create the database

Update *Program.cs* to create the database if it doesn't exist:

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works well early in development when the schema is rapidly evolving, as long as you don't need to preserve data. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, you delete the database that was created by `EnsureCreated` and use migrations instead. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create *Data/DbInitializer.cs* with the following code:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2019-09-
01")},
                new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-09-
01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-09-
01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-
01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-09-
01")},
                new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-
01")}
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
```

```
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
            };

            context.Courses.AddRange(courses);
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In *Program.cs*, replace the `EnsureCreated` call with a `DbInitializer.Initialize` call:

```
// context.Database.EnsureCreated();
DbInitializer.Initialize(context);
```

- Visual Studio
- Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

- Restart the app.
- Select the Students page to see the seeded data.

View the database

- Visual Studio
- Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name you provided earlier plus a dash and a GUID.

- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task<T>` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

Next steps

NEXT TUTORIAL

The Contoso University sample web app demonstrates how to create an ASP.NET Core Razor Pages app using Entity Framework (EF) Core.

The sample app is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This page is the first in a series of tutorials that explain how to build the Contoso University sample app.

Download or view the completed app. Download instructions.

Prerequisites

- Visual Studio
- Visual Studio Code

Visual Studio 2019 with the following workloads:

- **ASP.NET and web development**
- **.NET Core cross-platform development**

.NET Core 2.1 SDK or later

Familiarity with Razor Pages. New programmers should complete Get started with Razor Pages before starting this series.

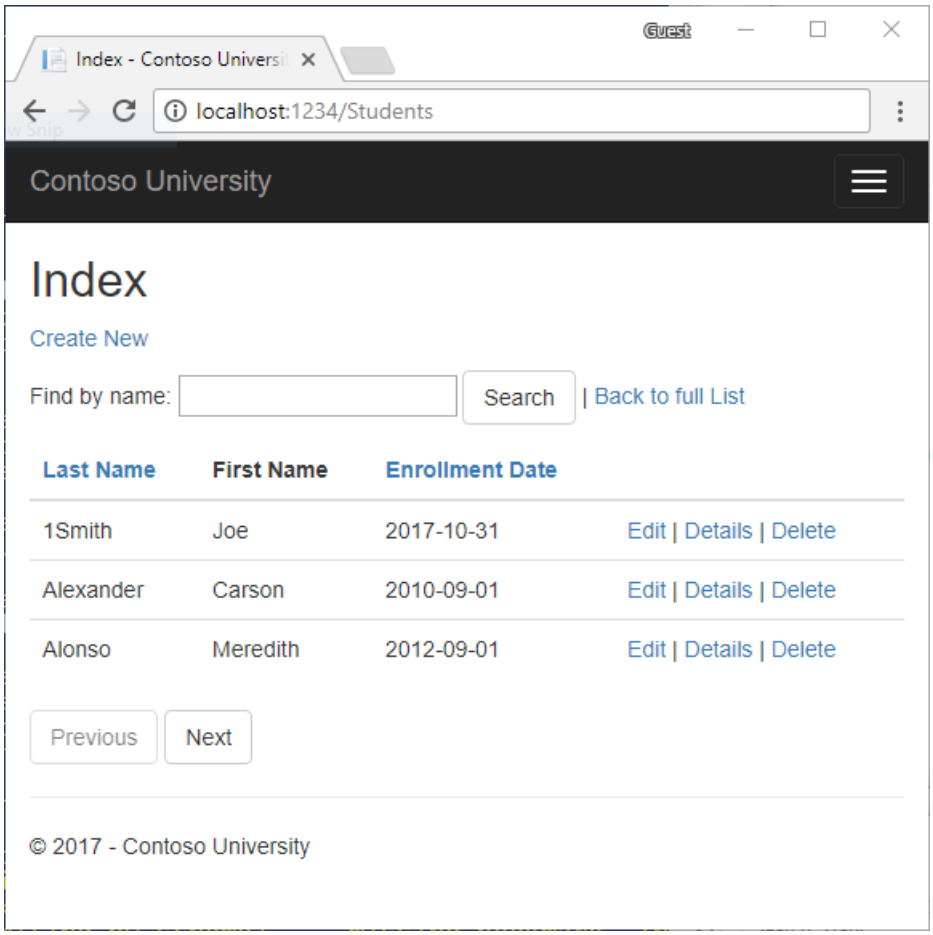
Troubleshooting

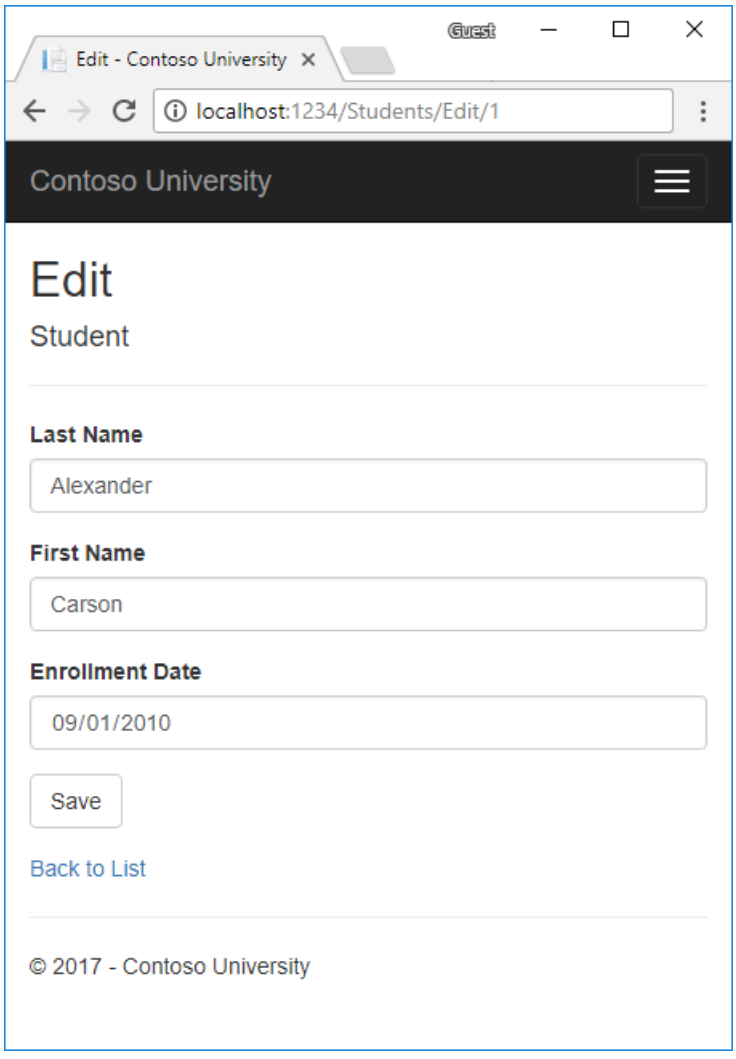
If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the completed project. A good way to get help is by posting a question to StackOverflow.com for ASP.NET Core or EF Core.

The Contoso University web app

The app built in these tutorials is a basic university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is close to what's generated by the built-in templates. The tutorial focus is on EF Core with

Razor Pages, not the UI.

Create the ContosoUniversity Razor Pages web app

- Visual Studio
- Visual Studio Code

- From the Visual Studio **File** menu, select **New** > **Project**.
- Create a new ASP.NET Core Web Application. Name the project **ContosoUniversity**. It's important to name the project *ContosoUniversity* so the namespaces match when code is copy/pasted.
- Select **ASP.NET Core 2.1** in the dropdown, and then select **Web Application**.

For images of the preceding steps, see Create a Razor web app. Run the app.

Set up the site style

A few changes set up the site menu, layout, and home page. Update *Pages/Shared/_Layout.cshtml* with the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Add menu entries for **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Contact** menu entry.

The changes are highlighted. (All the markup is *not* displayed.)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] : Contoso University</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">Contoso University</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Students/Index">Students</a></li>
                    <li><a asp-page="/Courses/Index">Courses</a></li>
                    <li><a asp-page="/Instructors/Index">Instructors</a></li>
                    <li><a asp-page="/Departments/Index">Departments</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 : Contoso University</p>
        </footer>
    </div>

    @*Remaining markup not shown for brevity.*@
```

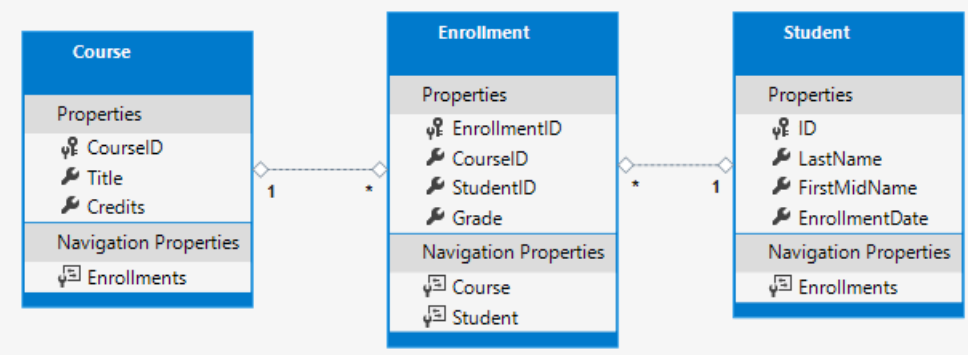
In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this app:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core Razor Pages web app.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p>
            <a class="btn btn-default"
               href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro">
                See the tutorial &raquo;
            </a>
        </p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p>
            <a class="btn btn-default"
               href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples/">
                See project source code &raquo;
            </a>
        </p>
    </div>
</div>
```

Create the data model

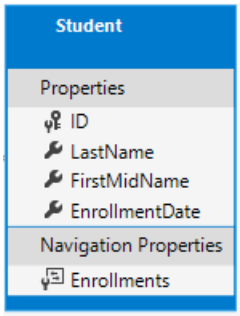
Create entity classes for the Contoso University app. Start with the following three entities:



There's a one-to-many relationship between `Student` and `Enrollment` entities. There's a one-to-many relationship between `Course` and `Enrollment` entities. A student can enroll in any number of courses. A course can have any number of students enrolled in it.

In the following sections, a class for each one of these entities is created.

The Student entity



Create a *Models* folder. In the *Models* folder, create a class file named *Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

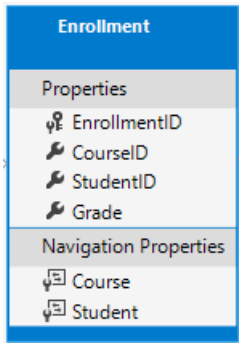
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property becomes the primary key column of the database (DB) table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. In `classnameID`, `classname` is the name of the class. The alternative automatically recognized primary key is `StudentID` in the preceding example.

The `Enrollments` property is a navigation property. Navigation properties link to other entities that are related to this entity. In this case, the `Enrollments` property of a `Student entity` holds all of the `Enrollment` entities that are related to that `Student`. For example, if a Student row in the DB has two related Enrollment rows, the `Enrollments` navigation property contains those two `Enrollment` entities. A related `Enrollment` row is a row that contains that student's primary key value in the `StudentID` column. For example, suppose the student with ID=1 has two rows in the `Enrollment` table. The `Enrollment` table has two rows with `StudentID` = 1. `StudentID` is a foreign key in the `Enrollment` table that specifies the student in the `Student` table.

If a navigation property can hold multiple entities, the navigation property must be a list type, such as `ICollection<T>`. `ICollection<T>` can be specified, or a type such as `List<T>` or `HashSet<T>`. When `ICollection<T>` is used, EF Core creates a `HashSet<T>` collection by default. Navigation properties that hold multiple entities come from many-to-many and one-to-many relationships.

The Enrollment entity



In the *Models* folder, create *Enrollment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key. This entity uses the `classnameID` pattern instead of `ID` like the `Student` entity. Typically developers choose one pattern and use it throughout the data model. In a later tutorial, using ID without classname is shown to make it easier to implement inheritance in the data model.

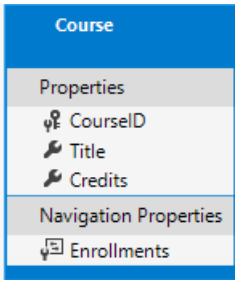
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity. The `Student` entity differs from the `Student.Enrollments` navigation property, which contains multiple `Enrollment` entities.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



In the *Models* folder, create *Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the DB generate it.

Scaffold the student model

In this section, the student model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the student model.

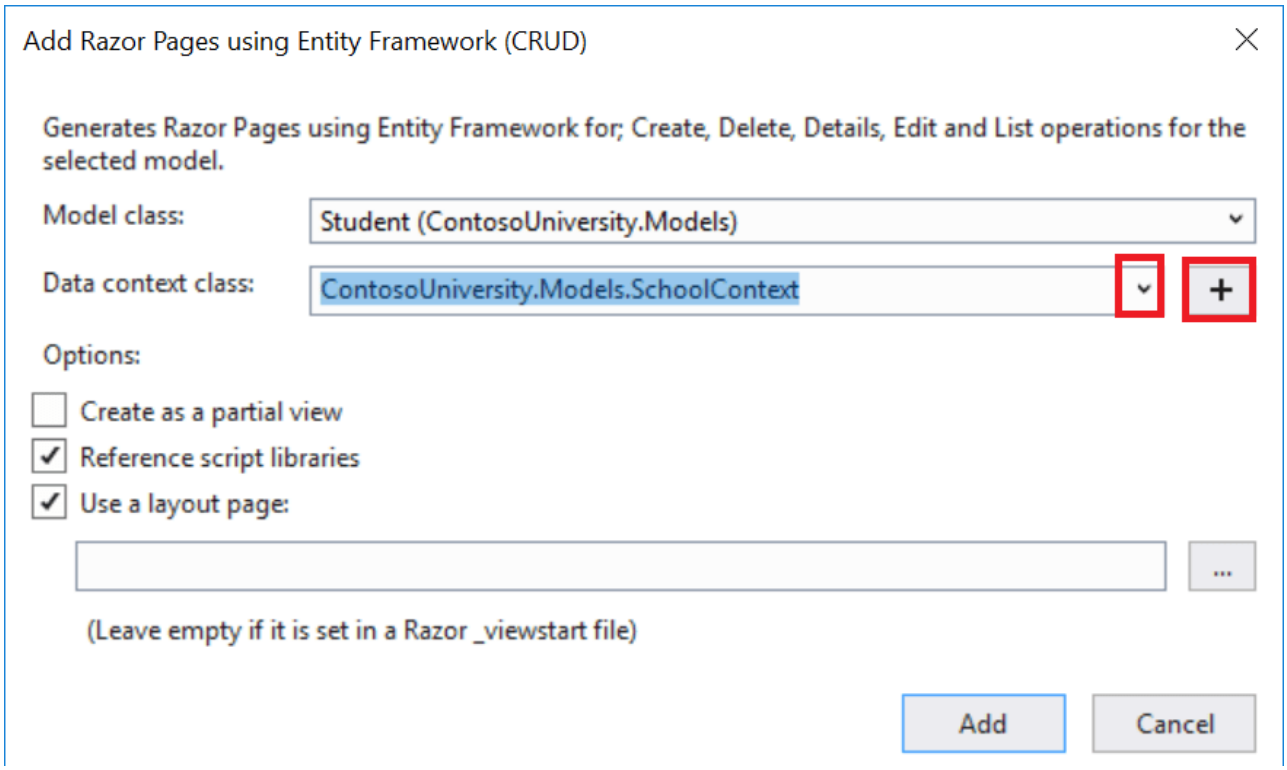
- Build the project.
- Create the *Pages/Students* folder.

- Visual Studio
- Visual Studio Code

- In **Solution Explorer**, right click on the *Pages/Students* folder > **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **ADD**.

Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:

- In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
- In the **Data context class** row, select the **+** (plus) sign and change the generated name to **ContosoUniversity.Models.SchoolContext**.
- In the **Data context class** drop-down, select **ContosoUniversity.Models.SchoolContext**
- Select **Add**.



See Scaffold the movie model if you have a problem with the preceding step.

The scaffold process created and changed the following files:

Files created

- *Pages/Students* Create, Delete, Details, Edit, Index.
- *Data/SchoolContext.cs*

File updates

- *Startup.cs* : Changes to this file are detailed in the next section.
- *appsettings.json* : The connection string used to connect to a local database is added.

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a db context instance is shown later in the tutorial.

The scaffolding tool automatically created a DB Context and registered it with the dependency injection container.

Examine the `ConfigureServices` method in *Startup.cs*. The highlighted line was added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for
        //non -essential cookies is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddDbContext<SchoolContext>(options =>
       options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Update main

In *Program.cs*, modify the `Main` method to do the following:

- Get a DB context instance from the dependency injection container.
- Call the EnsureCreated.
- Dispose the context when the `EnsureCreated` method completes.

The following code shows the updated *Program.cs* file.

```
using ContosoUniversity.Models;                  // SchoolContext
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;   // CreateScope
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }

            host.Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

`EnsureCreated` ensures that the database for the context exists. If it exists, no action is taken. If it does not exist, then the database and all its schema are created. `EnsureCreated` does not use migrations to create the database. A database that is created with `EnsureCreated` cannot be later updated using migrations.

`EnsureCreated` is called on app start, which allows the following work flow:

- Delete the DB.
- Change the DB schema (for example, add an `EmailAddress` field).
- Run the app.
- `EnsureCreated` creates a DB with the `EmailAddress` column.

`EnsureCreated` is convenient early in development when the schema is rapidly evolving. Later in the tutorial the DB is deleted and migrations are used.

Test the app

Run the app and accept the cookie policy. This app doesn't keep personal information. You can read about the cookie policy at EU General Data Protection Regulation (GDPR) support.

- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Examine the SchoolContext DB context

The main class that coordinates EF Core functionality for a given data model is the DB context class. The data context is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update *SchoolContext.cs* with the following code:

```
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Models
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Student { get; set; }
        public DbSet<Enrollment> Enrollment { get; set; }
        public DbSet<Course> Course { get; set; }
    }
}
```

The highlighted code creates a DbSet<TEntity> property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a DB table.
- An entity corresponds to a row in the table.

`DbSet<Enrollment>` and `DbSet<Course>` could be omitted. EF Core includes them implicitly because the `Student` entity references the `Enrollment` entity, and the `Enrollment` entity references the `Course` entity. For this tutorial, keep `DbSet<Enrollment>` and `DbSet<Course>` in the `SchoolContext`.

SQL Server Express LocalDB

The connection string specifies SQL Server LocalDB. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB creates *.mdf* DB files in the `C:/Users/<user>` directory.

Add code to initialize the DB with test data

EF Core creates an empty DB. In this section, an `Initialize` method is written to populate it with test data.

In the *Data* folder, create a new class file named *DbInitializer.cs* and add the following code:

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Models
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Student.Any())
            {
```

```
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-
01")},
            new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
            new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
            new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
            new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Student.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Course.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollment.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

Note: The preceding code uses `Models` for the namespace (`namespace ContosoUniversity.Models`) rather than `Data` . `Models` is consistent with the scaffolder-generated code. For more information, see this GitHub scaffolding issue.

The code checks if there are any students in the DB. If there are no students in the DB, the DB is initialized with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

The `EnsureCreated` method automatically creates the DB for the DB context. If the DB exists, `EnsureCreated` returns without modifying the DB.

In *Program.cs*, modify the `Main` method to call `Initialize`:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<SchoolContext>();
                // using ContosoUniversity.Data;
                DbInitializer.Initialize(context);
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred creating the DB.");
            }
        }

        host.Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

- Visual Studio
- Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

View the DB

The database name is generated from the context name you provided earlier plus a dash and a GUID. Thus, the database name will be "SchoolContext-{GUID}". The GUID will be different for each user. Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio. In SSOX, click **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**.

Expand the **Tables** node.

Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With

synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Student = await _context.Student.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Automatically create the Task object that's returned. For more information, see Task Return Type.
- The implicit return type `Task` represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the DB are executed asynchronously. That includes, `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the DB.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

In the next tutorial, basic CRUD (create, read, update, delete) operations are examined.

Additional resources

- YouTube version of this tutorial

NEXT

Part 2, Razor Pages with EF Core in ASP.NET Core - CRUD

9/22/2020 • 22 minutes to read • Edit Online

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, you add enrollments to the Details page.

Read enrollments

To display a student's enrollment data on the page, you need to read it. The scaffolded code in *Pages/Students/Details.cshtml.cs* reads only the Student data, without the Enrollment data:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the Reading related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in *Pages/Students/Details.cshtml* with the following code to display a list of enrollments. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for the

selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is unnecessary if the query can only return one entity, as when it searches on a unique key.
- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in *Pages/Students/Create.cshtml.cs* with the following code.

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` has to be converted to DateTime.

Run the app, and create a student entity to test the Create page.

Overposting

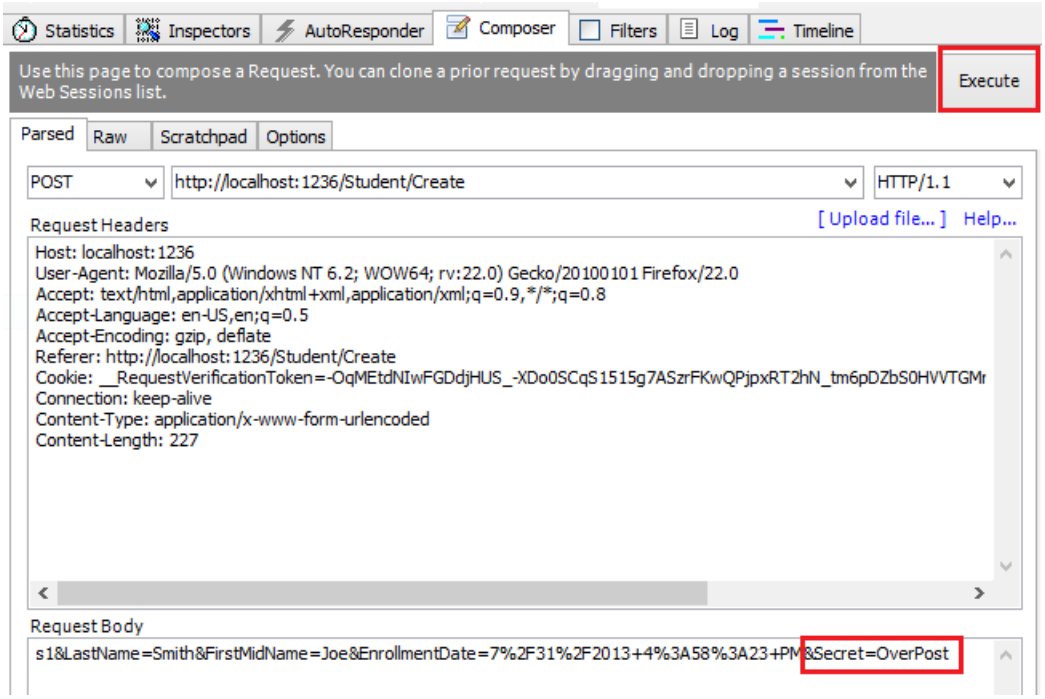
Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents

overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI that it is used for (for example, the Create page).

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `Student` view model:

```
using System;

namespace ContosoUniversity.Models
{
    public class StudentVM
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }
    }
}
```

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type doesn't need to be related to the model type, it just needs to have properties that match.

Using `StudentVM` requires Create.cshtml be updated to use `StudentVM` rather than `Student`.

Update the Edit page

In *Pages/Students/Edit.cshtml.cs*, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

    if (studentToUpdate == null)
    {
        return NotFound();
    }

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When you don't have to include related data, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.

- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`: The entity isn't being tracked by the database context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL UPDATE statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, you implement a custom error message when the call to `SaveChanges` fails.

Replace the code in *Pages/Students/Delete.cshtml.cs* with the following code. The changes are highlighted (other than cleanup of `using` statements).

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Student == null)
            {
                return NotFound();
            }

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = "Delete failed. Try again";
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
```

```
        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var student = await _context.Students.FindAsync(id);

            if (student == null)
            {
                return NotFound();
            }

            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException /* ex */)
            {
                //Log the error (uncomment ex variable name and write a log.)
                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object. The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is false when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` (because the delete operation failed), the `saveChangesError` parameter is true.

The `OnPostAsync` method retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated. If `Remove` fails:

- The database exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to the Delete Razor Page (*Pages/Students/Delete.cshtml*):

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

PREVIOUS TUTORIAL NEXT TUTORIAL

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

To minimize complexity and keep these tutorials focused on EF Core, EF Core code is used in the page models. Some developers use a service layer or repository pattern in to create an abstraction layer between the UI (Razor Pages) and the data access layer.

In this tutorial, the Create, Edit, Delete, and Details Razor Pages in the *Students* folder are examined.

The scaffolded code uses the following pattern for Create, Edit, and Delete pages:

- Get and display the requested data with the HTTP GET method `OnGetAsync`.

- Save changes to the data with the HTTP POST method `OnPostAsync`.

The Index and Details pages get and display the requested data with the HTTP GET method `OnGetAsync`

SingleOrDefaultAsync vs. FirstOrDefaultAsync

The generated code uses FirstOrDefaultAsync, which is generally preferred over SingleOrDefaultAsync.

`FirstOrDefaultAsync` is more efficient than `SingleOrDefaultAsync` at fetching one entity:

- Unless the code needs to verify that there's not more than one entity returned from the query.
- `SingleOrDefaultAsync` fetches more data and does unnecessary work.
- `SingleOrDefaultAsync` throws an exception if there's more than one entity that fits the filter part.
- `FirstOrDefaultAsync` doesn't throw if there's more than one entity that fits the filter part.

FindAsync

In much of the scaffolded code, FindAsync can be used in place of `FirstOrDefaultAsync`.

`FindAsync`:

- Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the DB.
- Is simple and concise.
- Is optimized to look up a single entity.
- Can have perf benefits in some situations, but that rarely happens for typical web apps.
- Implicitly uses FirstAsync instead of SingleAsync.

But if you want to `Include` other entities, then `FindAsync` is no longer appropriate. This means that you may need to abandon `FindAsync` and move to a query as your app progresses.

Customize the Details page

Browse to `Pages/Students` page. The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the *Pages/Students/Index.cshtml* file.

```
<td>
    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
</td>
```

Run the app and select a **Details** link. The URL is of the form `http://localhost:5000/Students/Details?id=2`. The Student ID is passed using a query string (`?id=2`).

Update the Edit, Details, and Delete Razor Pages to use the `"{id:int}"` route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`.

A request to the page with the "{id:int}" route template that does **not** include a integer route value returns an HTTP 404 (not found) error. For example, `http://localhost:5000/Students/Details` returns a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

Run the app, click on a Details link, and verify the URL is passing the ID as route data (

`http://localhost:5000/Students/Details/2`).

Don't globally change `@page` to `@page "{id:int}"`, doing so breaks the links to the Home and Create pages.

Add related data

The scaffolded code for the Students Index page doesn't include the `Enrollments` property. In this section, the contents of the `Enrollments` collection is displayed in the Details page.

The `OnGetAsync` method of *Pages/Students/Details.cshtml.cs* uses the `FirstOrDefaultAsync` method to retrieve a single `Student` entity. Add the following highlighted code:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Student
                        .Include(s => s.Enrollments)
                            .ThenInclude(e => e.Course)
                        .AsNoTracking()
                        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the reading-related data tutorial.

The AsNoTracking method improves performance in scenarios when the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display related enrollments on the Details page

Open *Pages/Students/Details.cshtml*. Add the following highlighted code to display a list of enrollments:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd>
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

If code indentation is wrong after the code is pasted, press CTRL-K-D to correct it.

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for the selected student is displayed.

Update the Create page

Update the `OnPostAsync` method in *Pages/Students/Create.cshtml.cs* with the following code:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Student.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return null;
}
```

TryUpdateModelAsync

Examine the TryUpdateModelAsync code:

```
var emptyStudent = new Student();

if (await TryUpdateModelAsync<Student>(
    emptyStudent,
    "student",   // Prefix for form value.
    s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
{
```

In the preceding code, `TryUpdateModelAsync<Student>` tries to update the `emptyStudent` object using the posted form values from the PageContext property in the PageModel. `TryUpdateModelAsync` only updates the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).

In the preceding sample:

- The second argument (`"student", // Prefix`) is the prefix uses to look up values. It's not case sensitive.
- The posted form values are converted to the types in the `Student` model using model binding.

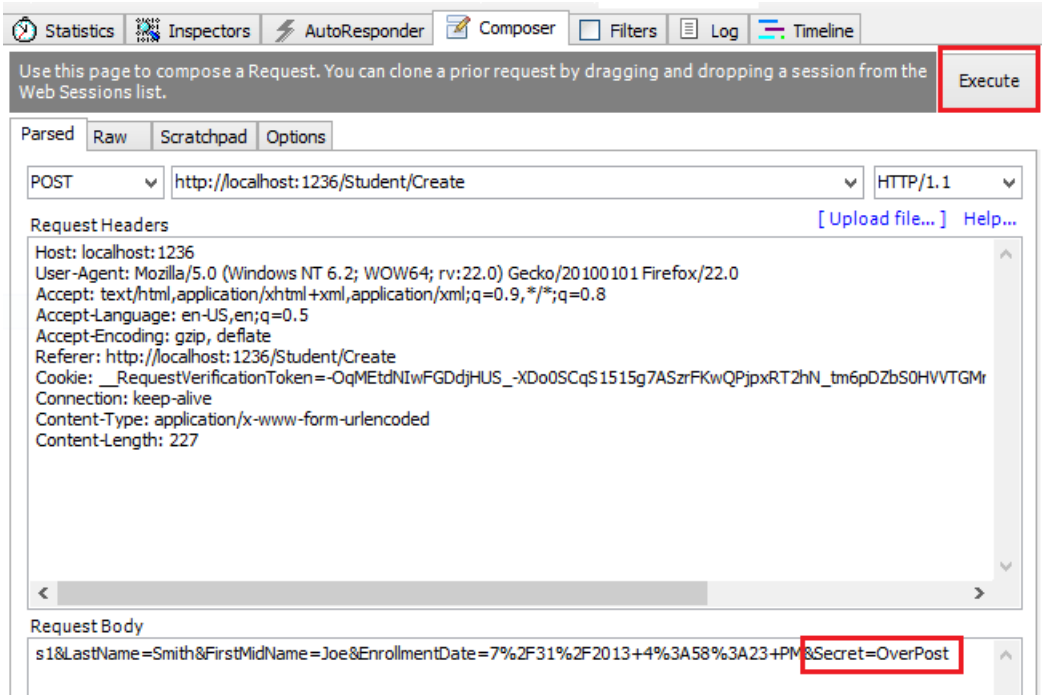
Overposting

Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if the app doesn't have a `Secret` field on the create/update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the DB. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. The app designer never intended the `Secret` property to be set with the Create page.

View model

A view model typically contains a subset of the properties included in the model used by the application. The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the DB. The view model contains only the properties needed for the UI layer (for example, the Create page). In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser. Consider the following `Student` view model:

```
using System;

namespace ContosoUniversity.Models
{
    public class StudentVM
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }
    }
}
```

View models provide an alternative way to prevent overposting. The view model contains only the properties to view (display) or update.

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type doesn't need to be related to the model type, it just needs to have properties that match.

Using `StudentVM` requires CreateVM.cshtml be updated to use `StudentVM` rather than `Student`.

In Razor Pages, the `PageModel` derived class is the view model.

Update the Edit page

Update the page model for the Edit page. The major changes are highlighted:

```
public class EditModel : PageModel
{
    private readonly SchoolContext _context;

    public EditModel(SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Student = await _context.Student.FindAsync(id);

        if (Student == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var studentToUpdate = await _context.Student.FindAsync(id);

        if (await TryUpdateModelAsync<Student>(
            studentToUpdate,
            "student",
            s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
        {
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }

        return Page();
    }
}
```

The code changes are similar to the Create page with a few exceptions:

- `OnPostAsync` has an optional `id` parameter.
- The current student is fetched from the DB, rather than creating an empty student.
- `FirstOrDefaultAsync` has been replaced with FindAsync. `FindAsync` is a good choice when selecting an entity from the primary key. See FindAsync for more information.

Test the Edit and Create pages

Create and edit a few student entities.

Entity States

The DB context keeps track of whether entities in memory are in sync with their corresponding rows in the DB. The

DB context sync information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the DB context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the DB. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the DB.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`: The entity isn't being tracked by the DB context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state to automatically be changed to `Modified`. Calling `SaveChanges` generates a SQL UPDATE statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Re-reading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, code is added to implement a custom error message when the call to `SaveChanges` fails. Add a string to contain possible error messages:

```
public class DeleteModel : PageModel
{
    private readonly SchoolContext _context;

    public DeleteModel(SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }
    public string ErrorMessage { get; set; }
```

Replace the `OnGetAsync` method with the following code:

```
public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Student
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ErrorMessage = "Delete failed. Try again";
    }

    return Page();
}
```

The preceding code contains the optional parameter `saveChangesError`. `saveChangesError` indicates whether the method was called after a failure to delete the student object. The delete operation might fail because of transient network problems. Transient network errors are more likely in the cloud. `saveChangesError` is false when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` (because the delete operation failed), the `saveChangesError` parameter is true.

The Delete pages OnPostAsync method

Replace the `OnPostAsync` with the following code:

```
public async Task<IActionResult> OnPostAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Student
                    .AsNoTracking()
                    .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    try
    {
        _context.Student.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("./Delete",
                             new { id, saveChangesError = true });
    }
}
```

The preceding code retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated. If `Remove` fails:

- The DB exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Update the Delete Razor Page

Add the following highlighted error message to the Delete Razor Page.

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
```

Test Delete.

Common errors

Students/Index or other links don't work:

Verify the Razor Page contains the correct `@page` directive. For example, The Students/Index Razor Page should **not** contain a route template:

```
@page "{id:int}"
```

Each Razor Page must include the `@page` directive.

Additional resources

- YouTube version of this tutorial

PREVIOUS NEXT

Part 3, Razor Pages with EF Core in ASP.NET Core - Sort, Filter, Paging

9/22/2020 • 29 minutes to read • Edit Online

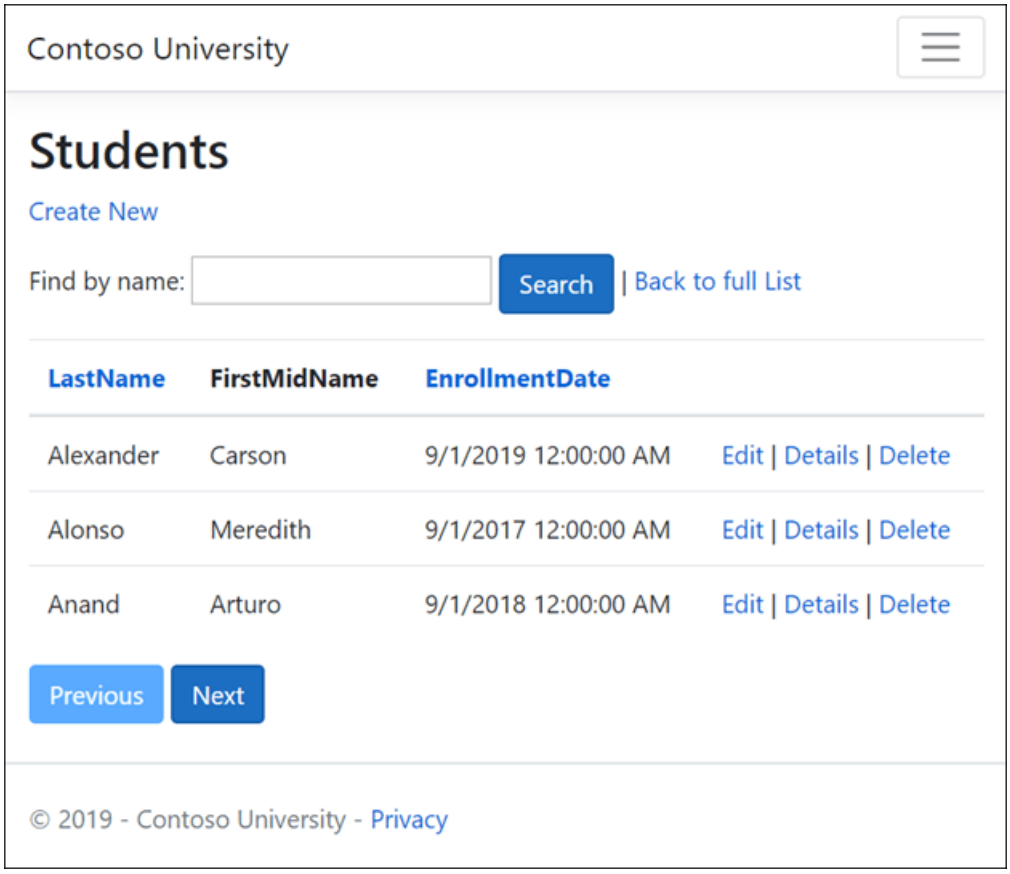
By Tom Dykstra, Rick Anderson, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial adds sorting, filtering, and paging functionality to the Students pages.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Click a column heading repeatedly to switch between ascending and descending sort order.



Add sorting

Replace the code in *Pages/Students/Index.cshtml.cs* with the following code to add sorting.

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;

        public IndexModel(SchoolContext context)
        {
            _context = context;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }

        public IList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder)
        {
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;

            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            Students = await studentsIQ.AsNoTracking().ToListAsync();
        }
    }
}
```

The preceding code:

- Adds properties to contain the sorting parameters.
- Changes the name of the `Student` property to `Students`.
- Replaces the code in the `OnGetAsync` method.

The `OnGetAsync` method receives a `sortOrder` parameter from the query string in the URL. The URL (including the query string) is generated by the Anchor Tag Helper.

The `sortOrder` parameter is either "Name" or "Date." The `sortOrder` parameter is optionally followed by "_desc" to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the default (fall-through case) in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The code uses the C# conditional operator ?:. The `?:` operator is a ternary operator (it takes three operands). The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to "name_desc." If `sortOrder` is **not** null or empty, `NameSort` is set to an empty string.

These two statements enable the page to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

```
IQueryable<Student> studentsIQ = from s in _context.Students
                                 select s;

switch (sortOrder)
{
    case "name_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
        break;
    case "Date":
        studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
        break;
    case "date_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
        break;
    default:
        studentsIQ = studentsIQ.OrderBy(s => s.LastName);
        break;
}

Students = await studentsIQ.AsNoTracking().ToListAsync();
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following

statement:

```
Students = await studentsIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns. For information about an alternative way to code this functionality, see Use dynamic LINQ to simplify code in the MVC version of this tutorial series.

Add column heading hyperlinks to the Student Index page

Replace the code in *Students/Index.cshtml*, with the following code. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.
- Changes the page heading from Index to Students.
- Changes `Model.Student` to `Model.Students`.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click the column headings.

Add filtering

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Update the OnGetAsync method

Replace the code in *Students/Index.cshtml.cs* with the following code to add filtering:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;

        public IndexModel(SchoolContext context)
        {
            _context = context;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }

        public IList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder, string searchString)
        {
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";

            CurrentFilter = searchString;

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s => s.LastName.Contains(searchString)
                                       || s.FirstMidName.Contains(searchString));
            }

            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            Students = await studentsIQ.AsNoTracking().ToListAsync();
        }
    }
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method, and saves the parameter value in the

`CurrentFilter` property. The search string value is received from a text box that's added in the next section.

- Adds to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is executed only if there's a value to search for.

IQueryable vs. IEnumerable

The code calls the `Where` method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. SQLite defaults to case-sensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())`
```

The preceding code would ensure that the filter is case-insensitive even if the `Where` method is called on an `IEnumerable` or runs on SQLite.

When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used.

Calling `Contains` on an `IQueryable` is usually preferable for performance reasons. With `IQueryable`, the filtering is done by the database server. If an `IEnumerable` is created first, all the rows have to be returned from the database server.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

For more information, see How to use case-insensitive query with Sqlite provider.

Update the Razor page

Replace the code in *Pages/Students/Index.cshtml* to create a **Search** button and assorted chrome.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string. If you're using SQLite, the filter is case-insensitive only if you implemented the optional `ToUpper` code shown earlier.
- Select **Search**.

Notice that the URL contains the search string. For example:

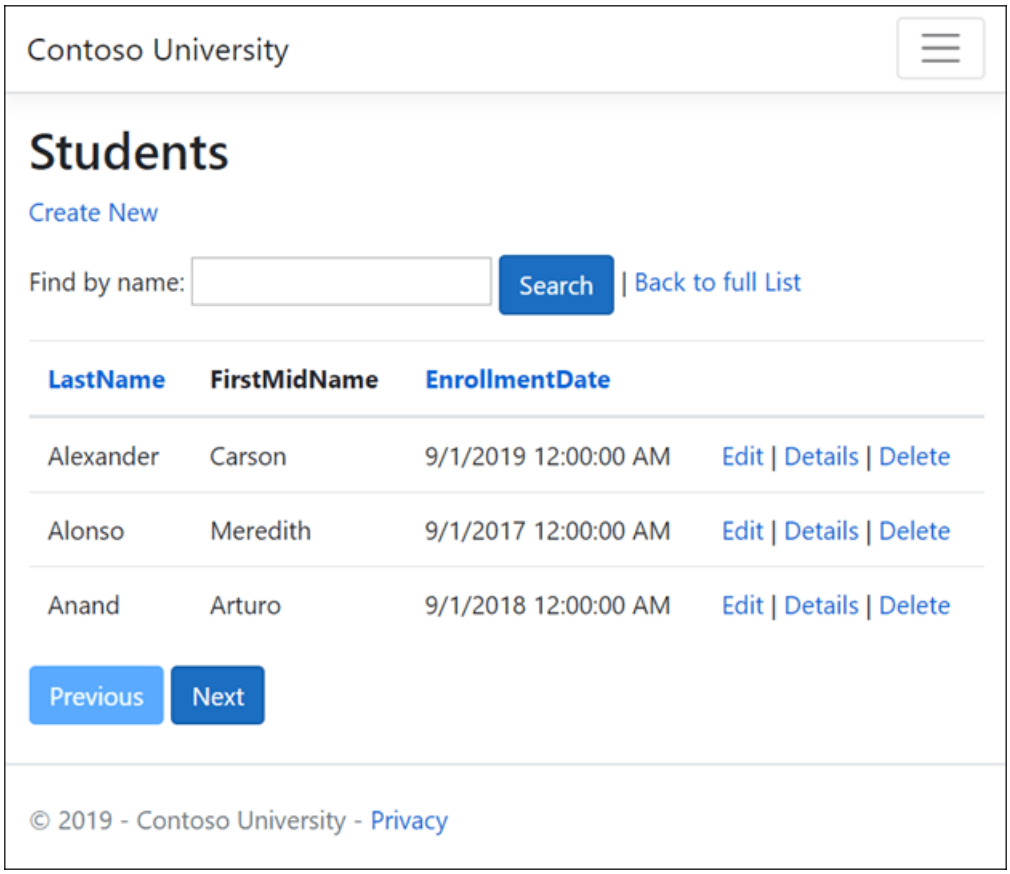
```
https://localhost:<port>/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



Create the PaginatedList class

In the project folder, create `PaginatedList.cs` with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object; constructors can't run asynchronous code.

Add paging to the PageModel class

Replace the code in *Students/Index.cshtml.cs* to add paging.

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;

        public IndexModel(SchoolContext context)
        {
            _context = context;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }

        public PaginatedList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder,
            string currentFilter, string searchString, int? pageIndex)
        {
            CurrentSort = sortOrder;
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";
            if (searchString != null)
            {
                pageIndex = 1;
            }
            else
            {
                searchString = currentFilter;
            }

            CurrentFilter = searchString;

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s => s.LastName.Contains(searchString)
                                       || s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            int pageSize = 3;
            Students = await PaginatedList<Student>.CreateAsync(
                studentsIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
        }
    }
}
```

```
}
```

The preceding code:

- Changes the type of the `Students` property from `IList<Student>` to `PaginatedList<Student>`.
- Adds the page index, the current `sortOrder`, and the `currentFilter` to the `OnGetAsync` method signature.
- Saves the sort order in the CurrentSort property.
- Resets page index to 1 when there's a new search string.
- Uses the `PaginatedList` class to get Student entities.

All the parameters that `OnGetAsync` receives are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

The `CurrentSort` property provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

The `CurrentFilter` property provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

The two question marks after `pageIndex` in the `PaginatedList.CreateAsync` call represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `(pageIndex ?? 1)` means return the value of `pageIndex` if it has a value. If `pageIndex` doesn't have a value, return 1.

Add paging links to the Razor Page

Replace the code in *Students/Index.cshtml* with the following code. The changes are highlighted:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
```

```
            <a asp-page ./Index >Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.Students.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Students.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method:

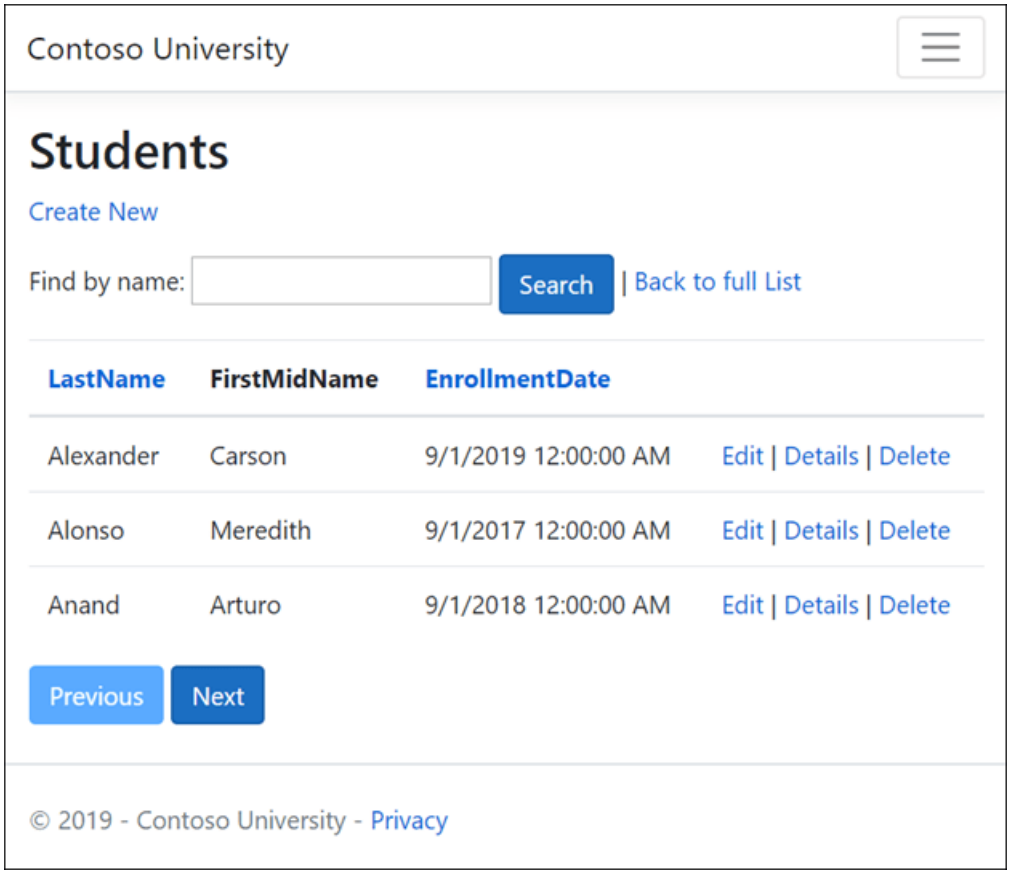
```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Students[0].LastName)
</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



Add grouping

This section creates an About page that displays how many students have enrolled for each enrollment date. The

update uses grouping and includes the following steps:

- Create a view model for the data used by the **About** page.
- Update the About page to use the view model.

Create the view model

Create a *Models/SchoolViewModels* folder.

Create *SchoolViewModels/EnrollmentDateGroup.cs* with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Create the Razor Page

Create a *Pages/About.cshtml* file with the following code:

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Students)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Create the page model

Create a *Pages/About.cshtml.cs* file with the following code:

```
using ContosoUniversity.Models.SchoolViewModels;
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

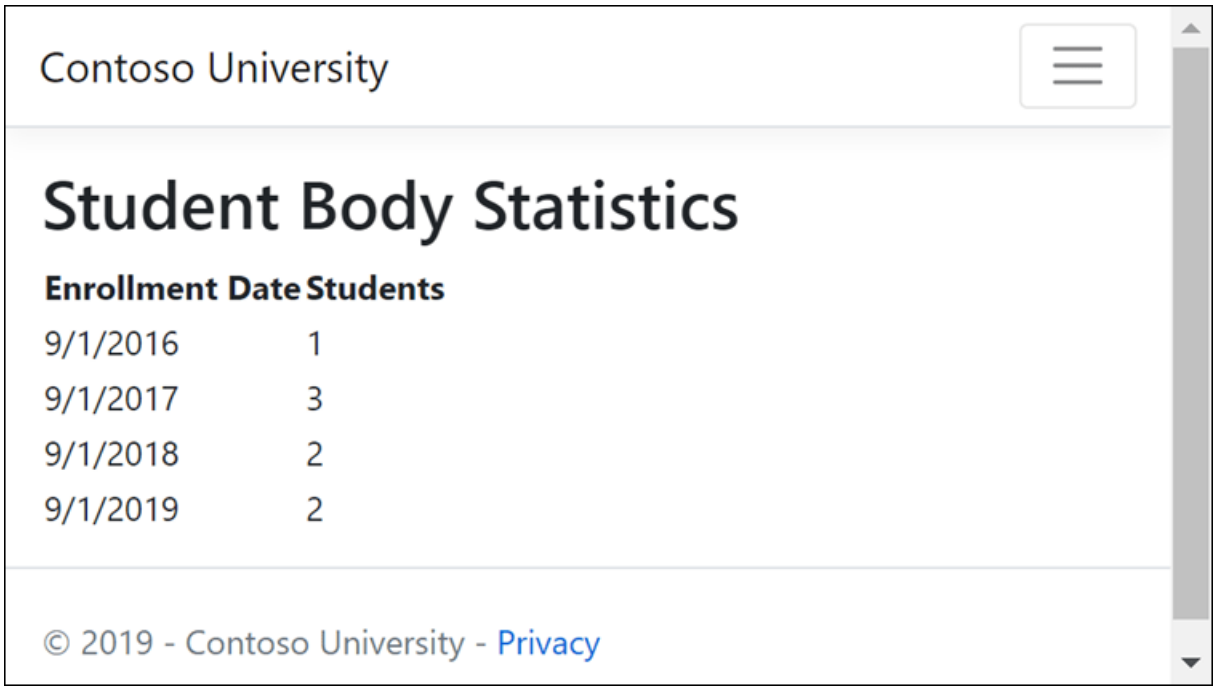
        public IList<EnrollmentDateGroup> Students { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Students = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.



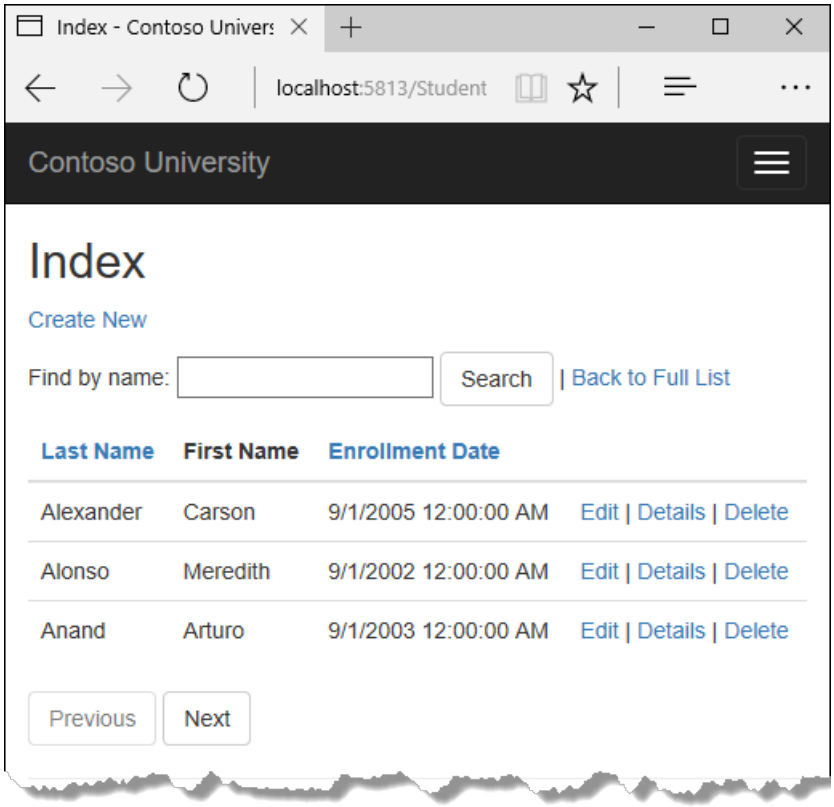
Next steps

In the next tutorial, the app uses migrations to update the data model.

PREVIOUS TUTORIAL | NEXT TUTORIAL

In this tutorial, sorting, filtering, grouping, and paging, functionality is added.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Clicking a column heading repeatedly switches between ascending and descending sort order.



If you run into problems you can't solve, download the completed app.

Add sorting to the Index page

Add strings to the *Students/Index.cshtml.cs* `PageModel` to contain the sorting parameters:

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }
```

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The preceding code receives a `sortOrder` parameter from the query string in the URL. The URL (including the query string) is generated by the Anchor Tag Helper

The `sortOrder` parameter is either "Name" or "Date." The `sortOrder` parameter is optionally followed by "_desc" to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the default (fall-through case) in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The following code contains the C# conditional ?: operator:

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to "name_desc." If `sortOrder` is **not** null or empty, `NameSort` is set to an empty string.

The `?: operator` is also known as the ternary operator.

These two statements enable the page to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following statement:

```
Student = await studentIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns.

Add column heading hyperlinks to the Student Index page

Replace the code in *Students/Index.cshtml*, with the following highlighted code:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click **Last Name**.
- Click **Enrollment Date**.

To get a better understanding of the code:

- In *Students/Index.cshtml.cs*, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort` and `DateSort`.
- In *Students/Index.cshtml*, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Add a Search Box to the Students Index page

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Add filtering functionality to the Index method

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder, string searchString)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method. The search string value is received from a text box that's added in the next section.
- Added to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is executed only if there's a value to search for.

Note: The preceding code calls the `Where` method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory

collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
```

The preceding code would ensure that results are case-insensitive if the code changes to use `IEnumerable`. When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used. Returning an `IEnumerable` from a repository can have a significant performance penalty:

1. All the rows are returned from the DB server.
2. The filter is applied to all the returned rows in the application.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

Add a Search Box to the Student Index page

In *Pages/Students/Index.cshtml*, add the following highlighted code to create a **Search** button and assorted chrome.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
```

The preceding code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string.

- Select **Search**.

Notice that the URL contains the search string.

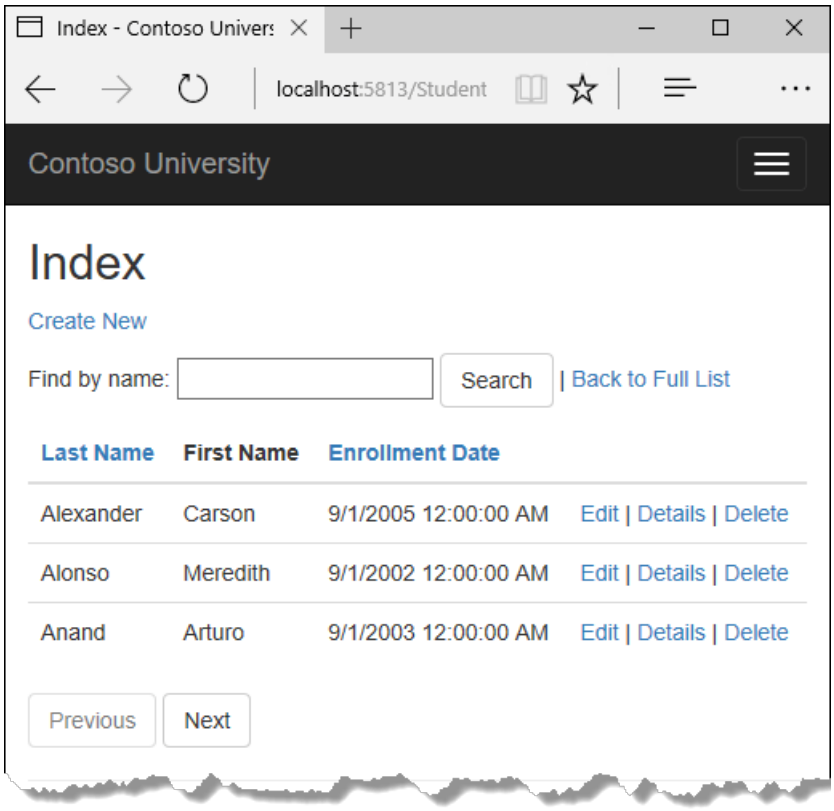
```
http://localhost:5000/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging functionality to the Students Index page

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs` with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object, constructors can't run asynchronous code.

Add paging functionality to the Index method

In *Students/Index.cshtml.cs*, update the type of `Student` from `IList<Student>` to `PaginatedList<Student>`:

```
public PaginatedList<Student> Student { get; set; }
```

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
{
    CurrentSort = sortOrder;
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    if (searchString != null)
    {
        pageIndex = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    Student = await PaginatedList<Student>.CreateAsync(
        studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
}
```

The preceding code adds the page index, the current `sortOrder`, and the `currentFilter` to the method signature.

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
```

All the parameters are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

`CurrentSort` provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

`CurrentFilter` provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

```
if (searchString != null)
{
    pageIndex = 1;
}
else
{
    searchString = currentFilter;
}
```

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

```
Student = await PaginatedList<Student>.CreateAsync(
    studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
```

The two question marks in `PaginatedList.CreateAsync` represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `(pageIndex ?? 1)` means return the value of `pageIndex` if it has a value. If `pageIndex` doesn't have a value, return 1.

Add paging links to the student Razor Page

Update the markup in *Students/Index.cshtml*. The changes are highlighted:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
```

```
                    asp-route-currentFilter="@Model.CurrentFilter">
                     @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.Student.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Student.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method so that the user can sort within filter results:

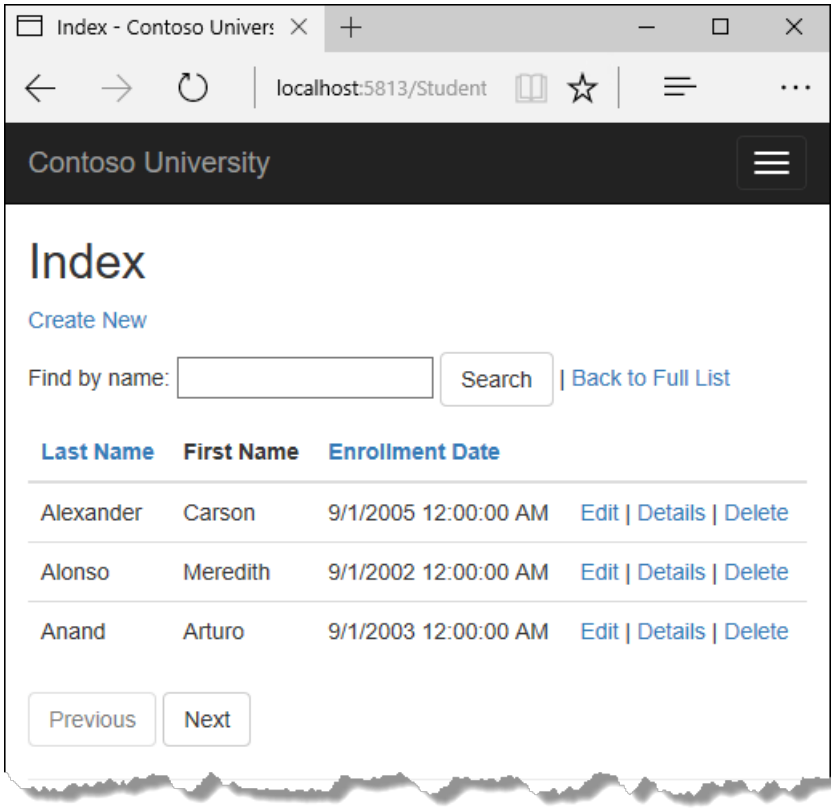
```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Student[0].LastName)
</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



To get a better understanding of the code:

- In *Students/Index.cshtml.cs*, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort`, `DateSort`, `CurrentSort`, and `Model.Student.PageIndex`.
- In *Students/Index.cshtml*, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Update the About page to show student statistics

In this step, *Pages/About.cshtml* is updated to display how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the **About** Page.
- Update the About page to use the view model.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the *SchoolViewModels* folder, add a *EnrollmentDateGroup.cs* with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Update the About page model

The web templates in ASP.NET Core 2.2 do not include the About page. If you are using ASP.NET Core 2.2, create the About Razor Page.

Update the *Pages/About.cshtml.cs* file with the following code:

```
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Student { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Student
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Student = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Modify the About Razor Page

Replace the code in the *Pages/About.cshtml* file with the following code:

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

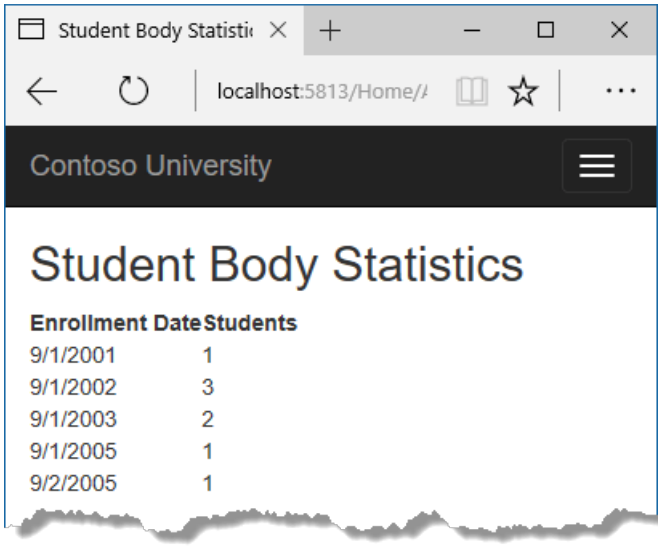
<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Student)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.

If you run into problems you can't solve, download the completed app for this stage.



Additional resources

- Debugging ASP.NET Core 2.x source
- YouTube version of this tutorial

In the next tutorial, the app uses migrations to update the data model.

PREVIOUS NEXT

Part 4, Razor Pages with EF Core migrations in ASP.NET Core

9/22/2020 • 11 minutes to read • Edit Online

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial introduces the EF Core migrations feature for managing data model changes.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial series started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes, you have to drop the database. The next time the app runs, the call to `EnsureCreated` re-creates the database to match the new data model. The `DbInitializer` class then runs to seed the new database.

This approach to keeping the database in sync with the data model works well until you deploy the app to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test database each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the database schema instead of creating a new database.

Rather than dropping and recreating the database when the data model changes, migrations updates the schema and retains existing data.

> **NOTE**
>
> **SQLite limitations**
>
> This tutorial uses the Entity Framework Core *migrations* feature where possible. Migrations updates the database schema to match changes in the data model. However, migrations only does the kinds of changes that the database engine supports, and SQLite's schema change capabilities are limited. For example, adding a column is supported, but removing a column is not supported. If a migration is created to remove a column, the `ef migrations add` command succeeds but the `ef database update` command fails.
>
> The workaround for the SQLite limitations is to manually write migrations code to perform a table rebuild when something in the table changes. The code would go in the `Up` and `Down` methods for a migration and would involve:
>
> - Creating a new table.
> - Copying data from the old table to the new table.
> - Dropping the old table.
> - Renaming the new table.
>
> Writing database-specific code of this type is outside the scope of this tutorial. Instead, this tutorial drops and re-creates the database whenever an attempt to apply a migration would fail. For more information, see the following resources:
>
> - SQLite EF Core Database Provider Limitations
> - Customize migration code
> - Data seeding
> - SQLite ALTER TABLE statement

Drop the database

- Visual Studio
- Visual Studio Code

Use **SQL Server Object Explorer** (SSOX) to delete the database, or run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

Create an initial migration

- Visual Studio
- Visual Studio Code

Run the following commands in the PMC:

```
Add-Migration InitialCreate
Update-Database
```

Up and Down methods

The EF Core `migrations add` command generated code to create the database. This migrations code is in the *Migrations<timestamp>_InitialCreate.cs* file. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

```
using System;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ContosoUniversity.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Course",
                columns: table => new
                {
                    CourseID = table.Column<int>(nullable: false),
                    Title = table.Column<string>(nullable: true),
                    Credits = table.Column<int>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Course", x => x.CourseID);
                });

            migrationBuilder.CreateTable(
                name: "Student",
                columns: table => new
                {
                    ID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    LastName = table.Column<string>(nullable: true),
                    FirstMidName = table.Column<string>(nullable: true),
```

```
                    EnrollmentDate = table.Column<DateTime>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Student", x => x.ID);
                });

            migrationBuilder.CreateTable(
                name: "Enrollment",
                columns: table => new
                {
                    EnrollmentID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    CourseID = table.Column<int>(nullable: false),
                    StudentID = table.Column<int>(nullable: false),
                    Grade = table.Column<int>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Enrollment", x => x.EnrollmentID);
                    table.ForeignKey(
                        name: "FK_Enrollment_Course_CourseID",
                        column: x => x.CourseID,
                        principalTable: "Course",
                        principalColumn: "CourseID",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_Enrollment_Student_StudentID",
                        column: x => x.StudentID,
                        principalTable: "Student",
                        principalColumn: "ID",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_CourseID",
                table: "Enrollment",
                column: "CourseID");

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_StudentID",
                table: "Enrollment",
                column: "StudentID");
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Enrollment");

            migrationBuilder.DropTable(
                name: "Course");

            migrationBuilder.DropTable(
                name: "Student");
        }
    }
}
```

The preceding code is for the initial migration. The code:

- Was generated by the `migrations add InitialCreate` command.
- Is executed by the `database update` command.
- Creates a database for the data model specified by the database context class.

The migration name parameter ("InitialCreate" in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

The migrations history table

- Use SSOX or your SQLite tool to inspect the database.
- Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the database.
- View the data in the `__EFMigrationsHistory` table. It shows one row for the first migration.

The data model snapshot

Migrations creates a *snapshot* of the current data model in *Migrations/SchoolContextModelSnapshot.cs*. When you add a migration, EF determines what changed by comparing the current data model to the snapshot file.

Because the snapshot file tracks the state of the data model, you can't delete a migration by deleting the `<timestamp>_<migrationname>.cs` file. To back out the most recent migration, you have to use the `migrations remove` command. That command deletes the migration and ensures the snapshot is correctly reset. For more information, see dotnet ef migrations remove.

Remove EnsureCreated

This tutorial series started by using `EnsureCreated`. `EnsureCreated` doesn't create a migrations history table and so can't be used with migrations. It's designed for testing or rapid prototyping where the database is dropped and re-created frequently.

From this point forward, the tutorials will use migrations.

In *Data/DBInitializer.cs*, comment out the following line:

```
context.Database.EnsureCreated();
```

Run the app and verify that the database is seeded.

Applying migrations in production

We recommend that production apps **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app that is deployed to a server farm. If the app is scaled out to multiple server instances, it's hard to ensure database schema updates don't happen from multiple servers or conflict with read/write access.

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

Troubleshooting

If the app uses SQL Server LocalDB and displays the following exception:

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.
The login failed.
Login failed for user 'user name'.
```

The solution may be to run `dotnet ef database update` at a command prompt.

Additional resources

- EF Core CLI.
- Package Manager Console (Visual Studio)

Next steps

The next tutorial builds out the data model, adding entity properties and new entities.

PREVIOUS TUTORIAL NEXT TUTORIAL

In this tutorial, the EF Core migrations feature for managing data model changes is used.

If you run into problems you can't solve, download the completed app.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes:

- The DB is dropped.
- EF creates a new one that matches the model.
- The app seeds the DB with test data.

This approach to keeping the DB in sync with the data model works well until you deploy the app to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the DB schema instead of creating a new DB.

Rather than dropping and recreating the DB when the data model changes, migrations updates the schema and retains existing data.

Drop the database

Use **SQL Server Object Explorer** (SSOX) or the `database drop` command:

- Visual Studio
- Visual Studio Code

In the **Package Manager Console** (PMC), run the following command:

```
Drop-Database
```

Run `Get-Help about_EntityFrameworkCore` from the PMC to get help information.

Create an initial migration and update the DB

Build the project and create the first migration.

- Visual Studio
- Visual Studio Code

```
Add-Migration InitialCreate
Update-Database
```

Examine the Up and Down methods

The EF Core `migrations add` command generated code to create the DB. This migrations code is in the *Migrations<timestamp>_InitialCreate.cs* file. The `Up` method of the `InitialCreate` class creates the DB tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true),
                Credits = table.Column<int>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        migrationBuilder.CreateTable(
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");

        migrationBuilder.DropTable(
            name: "Course");

        migrationBuilder.DropTable(
            name: "Student");
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, migrations calls the `Down` method.

The preceding code is for the initial migration. That code was created when the `migrations add InitialCreate` command was run. The migration name parameter ("InitialCreate" in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

If the initial migration is created and the DB exists:

- The DB creation code is generated.
- The DB creation code doesn't need to run because the DB already matches the data model. If the DB creation code is run, it doesn't make any changes because the DB already matches the data model.

When the app is deployed to a new environment, the DB creation code must be run to create the DB.

Previously the DB was dropped and doesn't exist, so migrations creates the new DB.

The data model snapshot

Migrations create a *snapshot* of the current database schema in *Migrations/SchoolContextModelSnapshot.cs*. When

you add a migration, EF determines what changed by comparing the data model to the snapshot file.

To delete a migration, use the following command:

- Visual Studio
- Visual Studio Code

Remove-Migration

The remove migrations command deletes the migration and ensures the snapshot is correctly reset.

Remove EnsureCreated and test the app

For early development, `EnsureCreated` was used. In this tutorial, migrations are used. `EnsureCreated` has the following limitations:

- Bypasses migrations and creates the DB and schema.
- Doesn't create a migrations table.
- Can *not* be used with migrations.
- Is designed for testing or rapid prototyping where the DB is dropped and re-created frequently.

Remove `EnsureCreated`:

```
context.Database.EnsureCreated();
```

Run the app and verify the DB is seeded.

Inspect the database

Use **SQL Server Object Explorer** to inspect the DB. Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the DB. View the data in the `__EFMigrationsHistory` table, it shows one row for the first migration. The last log in the preceding CLI output example shows the INSERT statement that creates this row.

Run the app and verify that everything works.

Applying migrations in production

We recommend production apps should **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app in server farm. For example, if the app has been cloud deployed with scale-out (multiple instances of the app are running).

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

EF Core uses the `__MigrationsHistory` table to see if any migrations need to run. If the DB is up-to-date, no migration is run.

Troubleshooting

Download the completed app.

The app generates the following exception:

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.
The login failed.
Login failed for user 'user name'.
```

Solution: Run `dotnet ef database update`

Additional resources

- YouTube version of this tutorial
- .NET Core CLI.
- Package Manager Console (Visual Studio)

Part 5, Razor Pages with EF Core in ASP.NET Core - Data Model

9/22/2020 • 57 minutes to read • Edit Online

By Tom Dykstra and Rick Anderson

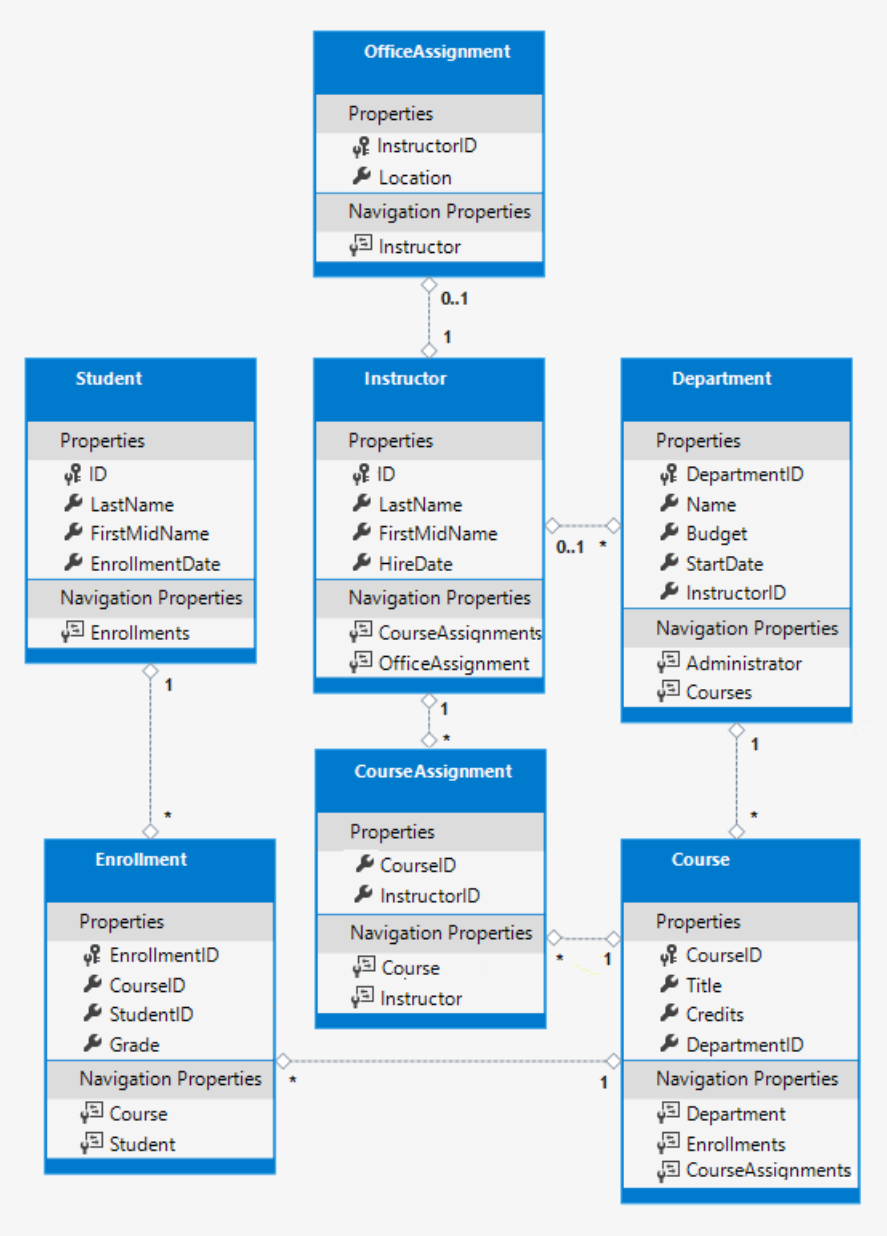
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

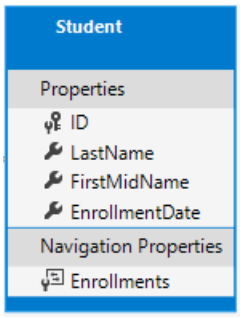
The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The completed data model is shown in the following illustration:



The Student entity



Replace the code in *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code adds a `FullName` property and adds the following attributes to existing properties:

- `[DataType]`
- `[DisplayFormat]`
- `[StringLength]`
- `[Column]`
- `[Required]`
- `[Display]`

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties.

`FullName` can't be set, so it has only a get accessor. No `FullName` column is created in the database.

The DataType attribute

```
[DataType(DataType.Date)]
```

For student enrollment dates, all of the pages currently display the time of day along with the date, although only the date is relevant. By using data annotation attributes, you can make one code change that will fix the display format in every page that shows the data.

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes. The `DataType` attributes don't provide validation.

The DisplayFormat attribute

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format. The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, and client-side input validation.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

The StringLength attribute

```
[StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
```

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The code shown limits names to no more than 50 characters. An example that sets the minimum string length is shown later.

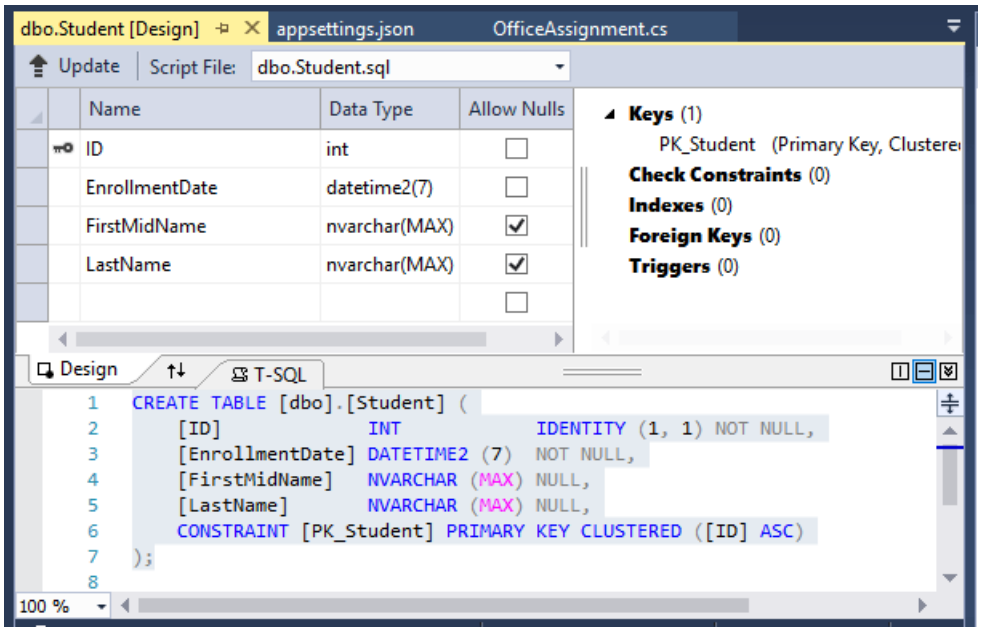
The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute can be used to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

- Visual Studio
- Visual Studio Code

In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)`. When a migration is created and applied later in this tutorial, the name fields become `nvarchar(50)` as a result of the string length attributes.

The Column attribute

```
[Column("FirstName")]
public string FirstMidName { get; set; }
```

Attributes can control how classes and properties are mapped to the database. In the `Student` model, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the database.

When the database is created, property names on the model are used for column names (except when the `Column` attribute is used). The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

With the `[Column]` attribute, `Student.FirstMidName` in the data model maps to the `FirstName` column of the `Student` table. The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. That discrepancy will be resolved by adding a migration later in this tutorial.

The Required attribute

```
[Required]
```

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (for example, `DateTime`, `int`, and `double`). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for full control over the string.

The Display attribute

```
[Display(Name = "Last Name")]
```

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

Create a migration

Run the app and go to the Students page. An exception is thrown. The `[Column]` attribute causes EF to expect to find a column named `FirstName`, but the column name in the database is still `FirstMidName`.

- Visual Studio
- Visual Studio Code

The error message is similar to the following example:

```
SqlException: Invalid column name 'FirstName'.
```

- In the PMC, enter the following commands to create a new migration and update the database:

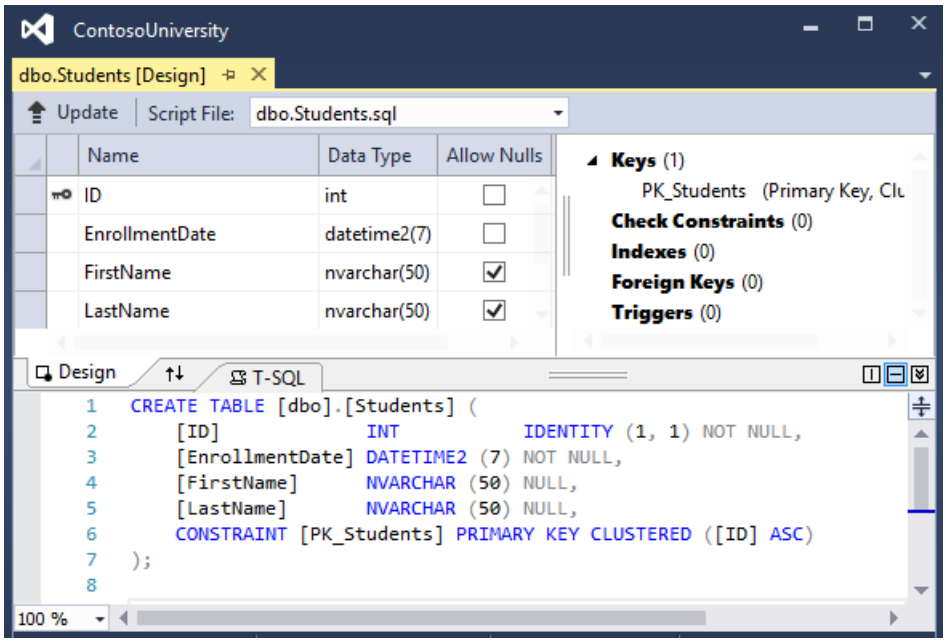
  ```
  Add-Migration ColumnFirstName
  Update-Database
  ```

  The first of these commands generates the following warning message:

  ```
  An operation was scaffolded that may result in the loss of data.
  Please review the migration for accuracy.
  ```

  The warning is generated because the name fields are now limited to 50 characters. If a name in the database had more than 50 characters, the 51 to last character would be lost.

- Open the Student table in SSOX:

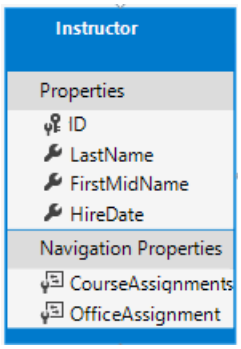


Before the migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

- Run the app and go to the Students page.
- Notice that times are not input or displayed along with dates.
- Select **Create New**, and try to enter a name longer than 50 characters.

> **NOTE**
> In the following sections, building the app at some stages generates compiler errors. The instructions specify when to build the app.

The Instructor Entity



Create *Models/Instructor.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

Navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

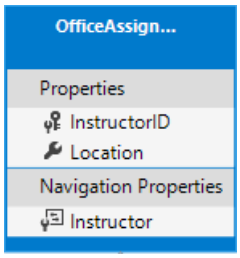
An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

An instructor can have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

```
public OfficeAssignment OfficeAssignment { get; set; }
```

The OfficeAssignment entity



Create *Models/OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than classnameID or ID.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity.

EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because `InstructorID` doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship.

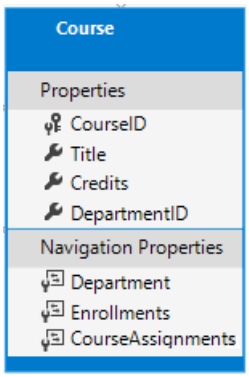
The Instructor navigation property

The `Instructor.OfficeAssignment` navigation property can be null because there might not be an `OfficeAssignment` row for a given instructor. An instructor might not have an office assignment.

The `OfficeAssignment.Instructor` navigation property will always have an instructor entity because the foreign key `InstructorID` type is `int`, a non-nullable value type. An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The Course Entity



Update *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a foreign key property for a data model when the model has a navigation property for a related entity. EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. However, explicitly including the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` property is null if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, EF Core assumes that PK values are generated by the database. Database-generated is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the database can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

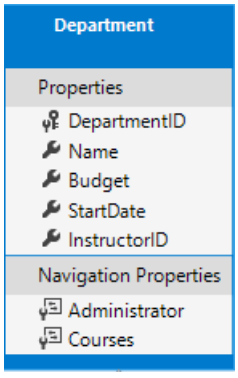
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection:

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

`CourseAssignment` is explained later.

The Department entity



Create *Models/Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required. EF Core chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

The question mark (?) in the preceding code specifies the property is nullable.

A department may have many courses, so there's a Courses navigation property:

```
public ICollection<Course> Courses { get; set; }
```

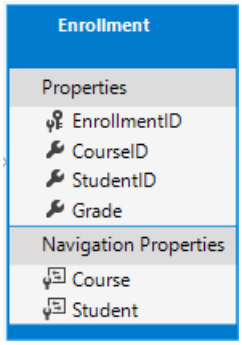
By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. This default behavior can result in circular cascade delete rules. Circular cascade delete rules cause an exception when a migration is added.

For example, if the `Department.InstructorID` property was defined as non-nullable, EF Core would configure a cascade delete rule. In that case, the department would be deleted when the instructor assigned as its administrator is deleted. In this scenario, a restrict rule would make more sense. The following fluent API would set a restrict rule and disable cascade delete.

```
modelBuilder.Entity<Department>()
   .HasOne(d => d.Administrator)
   .WithMany()
   .OnDelete(DeleteBehavior.Restrict)
```

The Enrollment entity

An enrollment record is for one course taken by one student.



Update *Models/Enrollment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

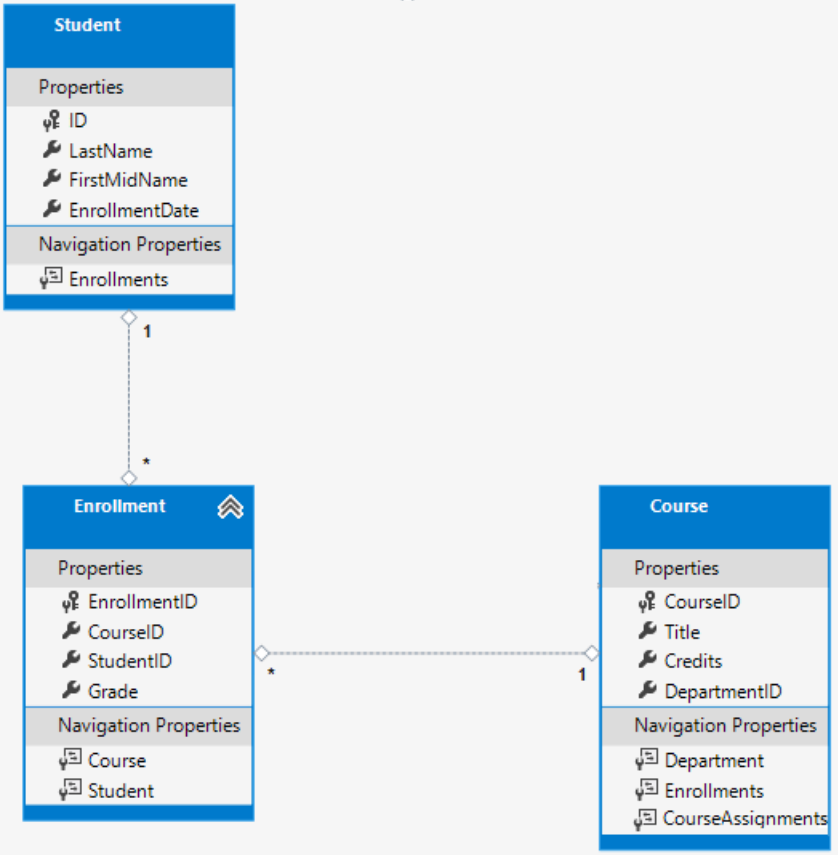
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities. The `Enrollment` entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the `Enrollment` table contains additional data besides FKs for the joined tables (in this case, the PK and `Grade`).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



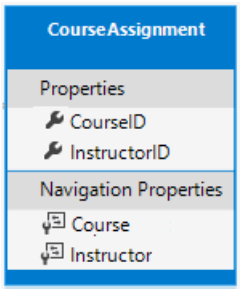
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs (`CourseID` and `StudentID`). A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a pure join table.

Note: EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see Many-to-many relationships in EF Core 2.0.

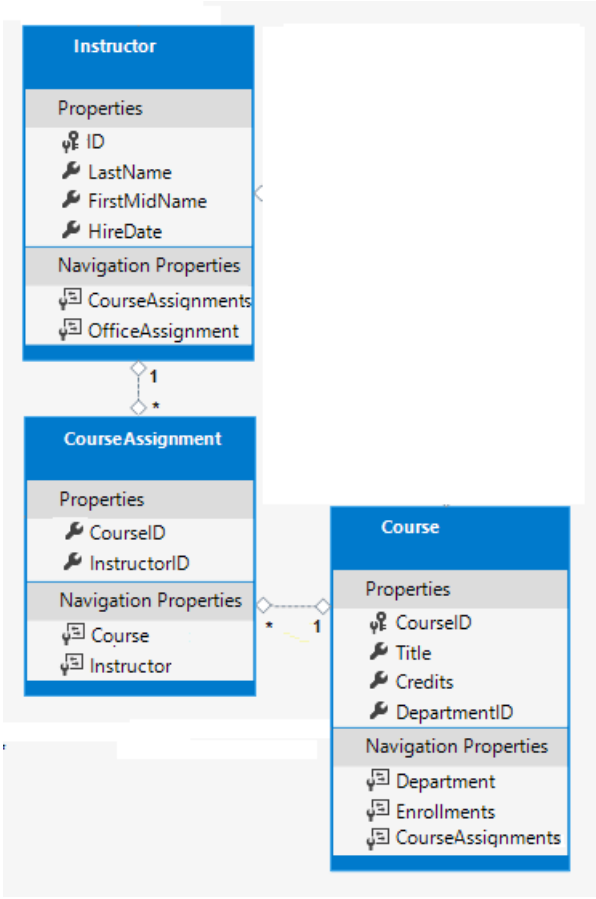
The CourseAssignment entity



Create *Models/CourseAssignment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

The Instructor-to-Courses many-to-many relationship requires a join table, and the entity for that join table is CourseAssignment.



It's common to name a join entity `EntityName1EntityName2`. For example, the Instructor-to-Courses join table using this pattern would be `CourseInstructor`. However, we recommend using a name that describes the relationship.

Data models start out simple and grow. Join tables without payload (PJTs) frequently evolve to include payload. By starting with a descriptive entity name, the name doesn't need to change when the join table changes. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked with a join entity called Ratings. For the Instructor-to-Courses many-to-many

relationship, `CourseAssignment` is preferred over `CourseInstructor`.

Composite key

The two FKs in `CourseAssignment` (`InstructorID` and `CourseID`) together uniquely identify each row of the `CourseAssignment` table. `CourseAssignment` doesn't require a dedicated PK. The `InstructorID` and `CourseID` properties function as a composite PK. The only way to specify composite PKs to EF Core is with the *fluent API*. The next section shows how to configure the composite PK.

The composite key ensures that:

- Multiple rows are allowed for one course.
- Multiple rows are allowed for one instructor.
- Multiple rows aren't allowed for the same instructor and course.

The `Enrollment` join entity defines its own PK, so duplicates of this sort are possible. To prevent such duplicates:

- Add a unique index on the FK fields, or
- Configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Update *Data/SchoolContext.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

The preceding code adds the new entities and configures the `CourseAssignment` entity's composite PK.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

In this tutorial, the fluent API is used only for database mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API (specifying a composite PK). There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

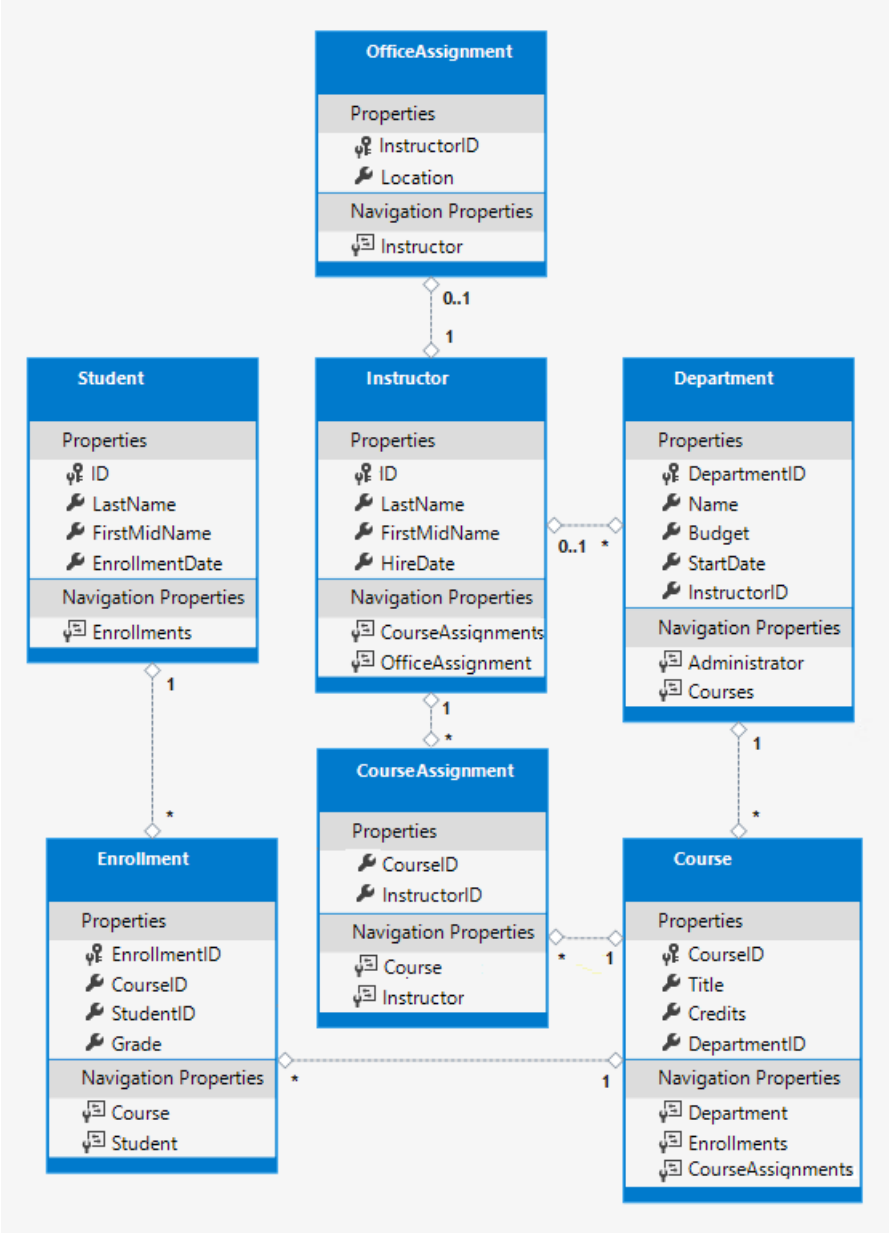
Some of the attributes used in this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Entity diagram

The following illustration shows the diagram that EF Power Tools create for the completed School model.



The preceding diagram shows:

- Several one-to-many relationship lines (1 to *).
- The one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment` entities.
- The zero-or-one-to-many relationship line (0..1 to *) between the `Instructor` and `Department` entities.

Seed the database

Update the code in *Data/DbInitializer.cs*:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();
```

```
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2016-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2019-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2017-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2019-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2011-09-01") }
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            context.Instructors.AddRange(instructors);
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
            };

            context.Departments.AddRange(departments);
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
                    DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
```

```
        },
        new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
        new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
    };

    context.Courses.AddRange(courses);
    context.SaveChanges();

    var officeAssignments = new OfficeAssignment[]
    {
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
            Location = "Smith 17" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
            Location = "Gowan 27" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
            Location = "Thompson 304" },
    };

    context.OfficeAssignments.AddRange(officeAssignments);
    context.SaveChanges();

    var courseInstructors = new CourseAssignment[]
    {
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
```

```
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
    };

    context.CourseAssignments.AddRange(courseInstructors);
    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            Grade = Grade.A
        },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            Grade = Grade.C
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
            Grade = Grade.B
            },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Li").ID,
            CourseID = courses.Single(c => c.Title == "Composition").CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Justice").ID,
            CourseID = courses.Single(c => c.Title == "Literature").CourseID,
            Grade = Grade.B
            }
    };

    foreach (Enrollment e in enrollments)
```

```
            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing. See `Enrollments` and `CourseAssignments` for examples of how many-to-many join tables can be seeded.

Add a migration

Build the project.

- Visual Studio
- Visual Studio Code

In PMC, run the following command.

```
Add-Migration ComplexDataModel
```

The preceding command displays a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
To undo this action, use 'ef migrations remove'
```

If the `database update` command is run, the following error is produced:

```
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint
"FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in
database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.
```

In the next section, you see what to do about this error.

Apply the migration or drop and re-create

Now that you have an existing database, you need to think about how to apply changes to it. This tutorial shows two alternatives:

- Drop and re-create the database. Choose this section if you're using SQLite.
- Apply the migration to the existing database. The instructions in this section work for SQL Server only, **not for SQLite**.

Either choice works for SQL Server. While the apply-migration method is more complex and time-consuming, it's the preferred approach for real-world, production environments.

Drop and re-create the database

Skip this section if you're using SQL Server and want to do the apply-migration approach in the following section.

To force EF Core to create a new database, drop and update the database:

- Visual Studio
- Visual Studio Code

- In the **Package Manager Console** (PMC), run the following command:

```
Drop-Database
```

- Delete the *Migrations* folder, then run the following command:

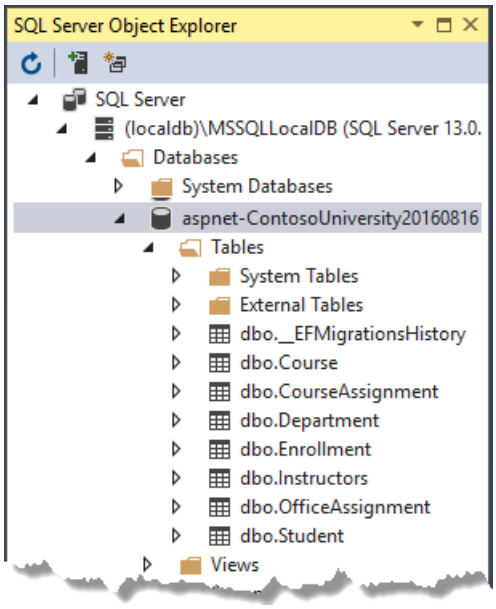
```
Add-Migration InitialCreate
Update-Database
```

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

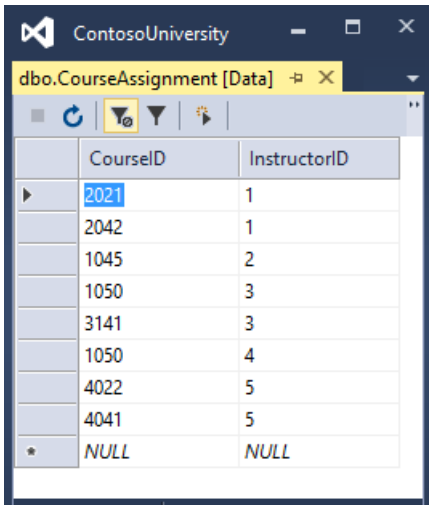
- Visual Studio
- Visual Studio Code

Open the database in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.



- Examine the **CourseAssignment** table:
  - Right-click the **CourseAssignment** table and select **View Data**.
  - Verify the **CourseAssignment** table contains data.



Apply the migration

This section is optional. These steps work only for SQL Server LocalDB and only if you skipped the preceding Drop and re-create the database section.

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. With production data, steps must be taken to migrate the existing data. This section provides an example of fixing FK constraint violations. Don't make these code changes without a backup. Don't make these code changes if you completed the preceding Drop and re-create the database section.

The *{timestamp}_ComplexDataModel.cs* file contains the following code:

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

The preceding code adds a non-nullable `DepartmentID` FK to the `Course` table. The database from the previous tutorial contains rows in `Course`, so that table cannot be updated by migrations.

To make the `ComplexDataModel` migration work with existing data:

- Change the code to give the new column (`DepartmentID`) a default value.
- Create a fake department named "Temp" to act as the default department.

Fix the foreign key constraints

In the `ComplexDataModel` migration class, update the `Up` method:

- Open the *{timestamp}_ComplexDataModel.cs* file.
- Comment out the line of code that adds the `DepartmentID` column to the `Course` table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

Add the following highlighted code. The new code goes after the `.CreateTable( name: "Department"` block:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(type: "int", nullable: true),
        Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(type: "datetime2", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

 migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00, GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

With the preceding changes, existing `Course` rows will be related to the "Temp" department after the `ComplexDataModel.Up` method runs.

The way of handling the situation shown here is simplified for this tutorial. A production app would:

- Include code or scripts to add `Department` rows and related `Course` rows to the new `Department` rows.
- Not use the "Temp" department or the default value for `Course.DepartmentID`.

- Visual Studio
- Visual Studio Code

- In the **Package Manager Console** (PMC), run the following command:

```
Update-Database
```

Because the `DbInitializer.Initialize` method is designed to work only with an empty database, use SSOX to delete all the rows in the Student and Course tables. (Cascade delete will take care of the Enrollment table.)

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

Next steps

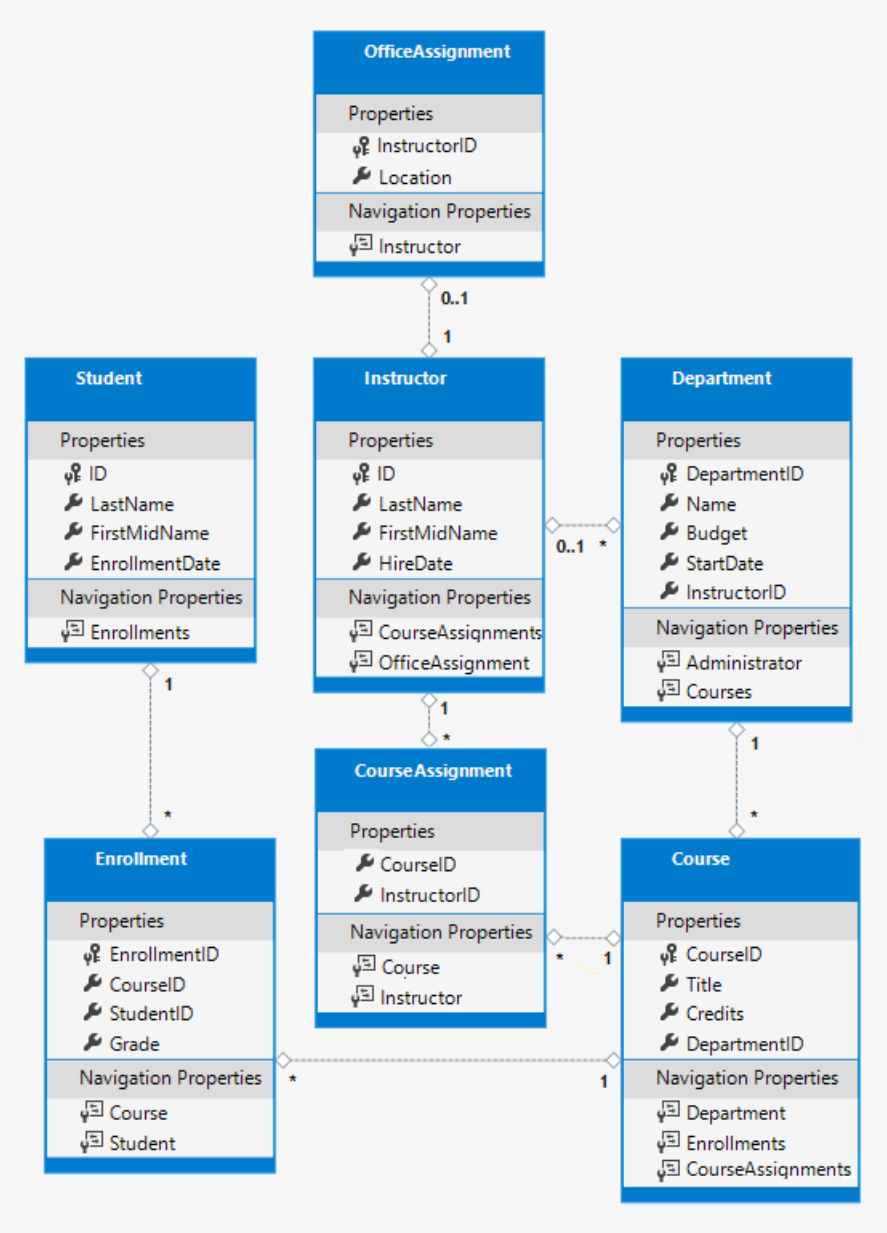
The next two tutorials show how to read and update related data.

PREVIOUS TUTORIAL NEXT TUTORIAL

The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The entity classes for the completed data model are shown in the following illustration:



If you run into problems you can't solve, download the completed app.

Customize the data model with attributes

In this section, the data model is customized using attributes.

The DataType attribute

The student pages currently displays the time of the enrollment date. Typically, date fields show only the date and not the time.

Update *Models/Student.cs* with the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers consume. The `DataType` attributes don't provide validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

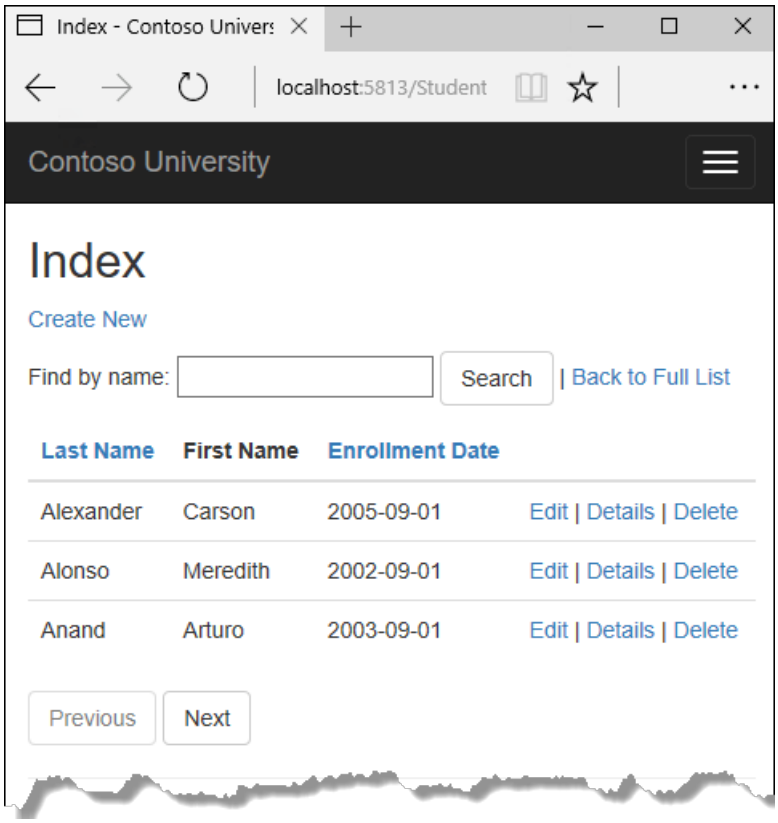
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, client-side input validation, etc.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

Run the app. Navigate to the Students Index page. Times are no longer displayed. Every view that uses the `Student` model displays the date without time.



The StringLength attribute

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

Update the `Student` model with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

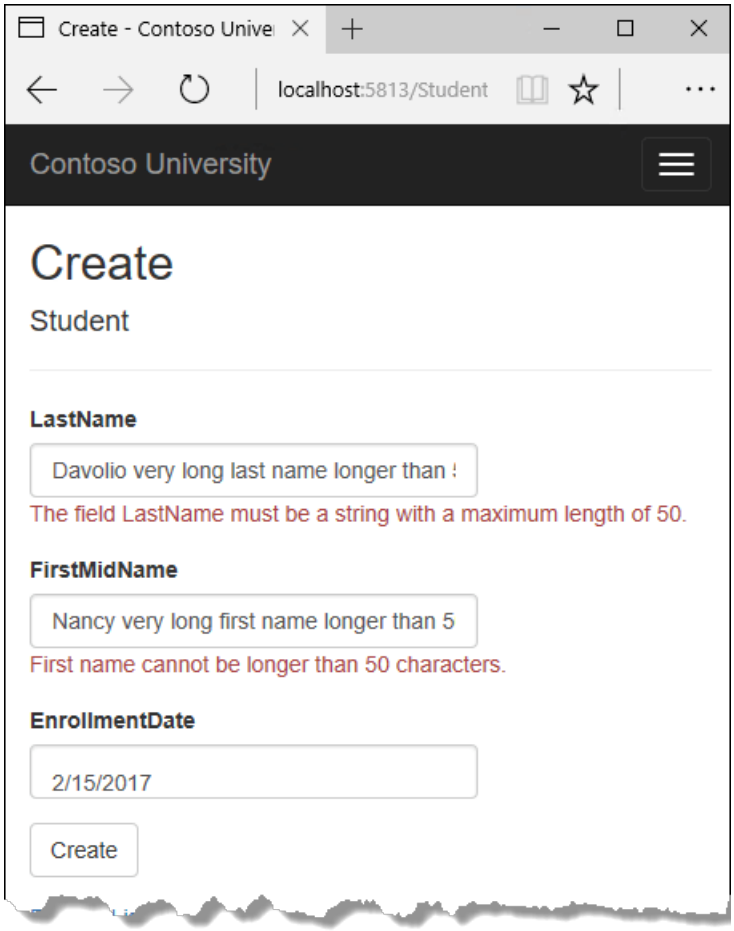
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code limits names to no more than 50 characters. The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute is used to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

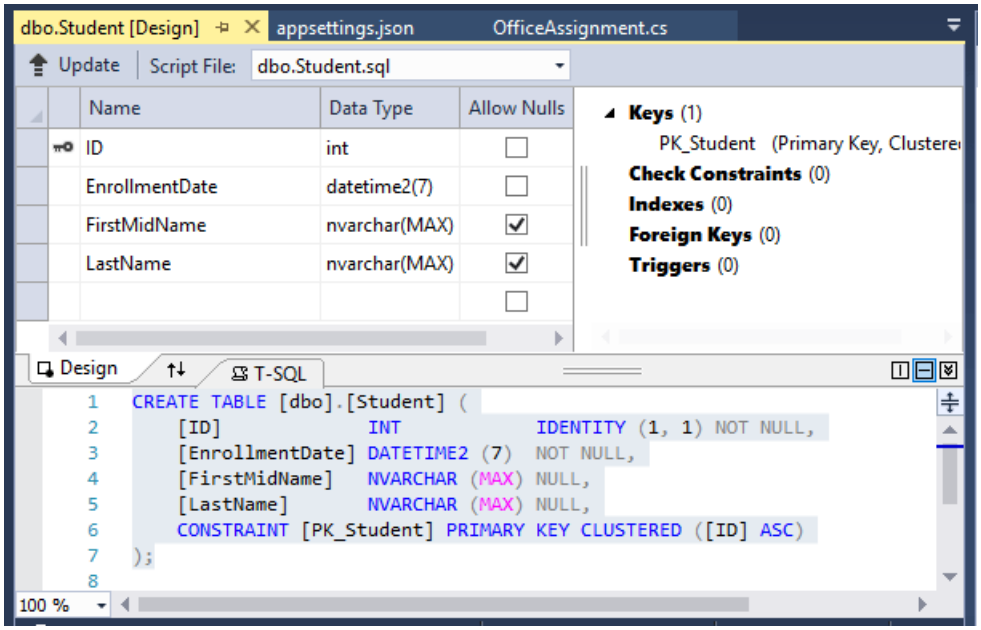
```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

Run the app:

- Navigate to the Students page.
- Select **Create New**, and enter a name longer than 50 characters.
- Select **Create**, client-side validation shows an error message.



In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)` because migrations has not been run on the DB. When migrations are run later in this tutorial, the name fields become

`nvarchar(50)`.

The Column attribute

Attributes can control how classes and properties are mapped to the database. In this section, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the DB.

When the DB is created, property names on the model are used for column names (except when the `Column` attribute is used).

The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

Update the *Student.cs* file with the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

With the preceding change, `Student.FirstMidName` in the app maps to the `FirstName` column of the `Student` table.

The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. If the app is run before applying migrations, the following exception is generated:

```
SqlException: Invalid column name 'FirstName'.
```

To update the DB:

- Build the project.
- Open a command window in the project folder. Enter the following commands to create a new migration and update the DB:

- Visual Studio
- Visual Studio Code

```
Add-Migration ColumnFirstName
Update-Database
```

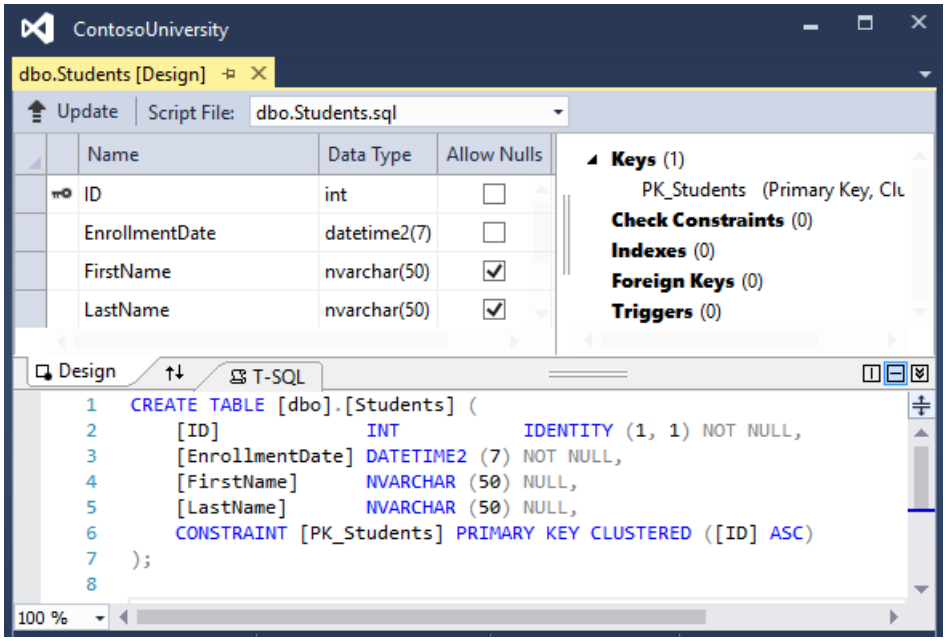
The `migrations add ColumnFirstName` command generates the following warning message:

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
```

The warning is generated because the name fields are now limited to 50 characters. If a name in the DB had more than 50 characters, the 51 to last character would be lost.

- Test the app.

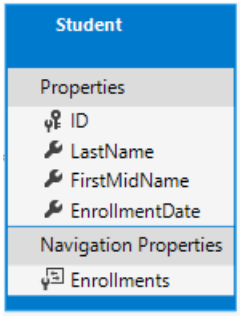
Open the Student table in SSOX:



Before migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

> **NOTE**
> In the following section, building the app at some stages generates compiler errors. The instructions specify when to build the app.

Student entity update



Update *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (`DateTime` , `int` , `double` , etc.). Types that can't be null are automatically treated as required fields.

The `Required` attribute could be replaced with a minimum length parameter in the `StringLength` attribute:

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

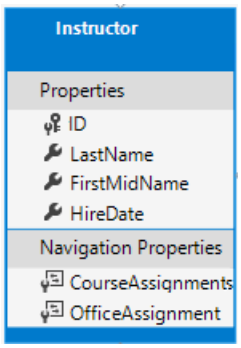
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` cannot be set, it has only a get accessor. No `FullName` column is created in the database.

Create the Instructor Entity



Create *Models/Instructor.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

If a navigation property holds multiple entities:

- It must be a list type where the entries can be added, deleted, and updated.

Navigation property types include:

- `ICollection<T>`
- `List<T>`
- `HashSet<T>`

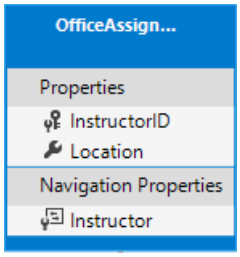
If `ICollection<T>` is specified, EF Core creates a `HashSet<T>` collection by default.

The `CourseAssignment` entity is explained in the section on many-to-many relationships.

Contoso University business rules state that an instructor can have at most one office. The `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment entity



Create *Models/OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than classnameID or ID.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key

(FK) to the `Instructor` entity. EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because:

- `InstructorID` doesn't follow the ID or classnameID naming convention.

Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The `OfficeAssignment` navigation property for the `Instructor` entity is nullable because:

- Reference types (such as classes are nullable).
- An instructor might not have an office assignment.

The `OfficeAssignment` entity has a non-nullable `Instructor` navigation property because:

- `InstructorID` is non-nullable.
- An office assignment can't exist without an instructor.

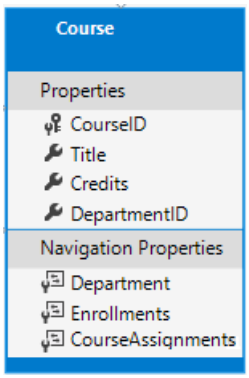
When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The `[Required]` attribute could be applied to the `Instructor` navigation property:

```
[Required]
public Instructor Instructor { get; set; }
```

The preceding code specifies that there must be a related instructor. The preceding code is unnecessary because the `InstructorID` foreign key (which is also the PK) is non-nullable.

Modify the Course Entity



Update *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a FK property for a data model when the model has a navigation property for a related entity.

EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. Having the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` entity is null if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, EF Core assumes that PK values are generated by the DB. DB generated PK values is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the DB can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

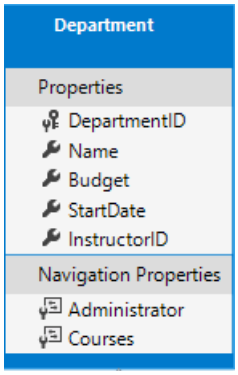
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection:

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

`CourseAssignment` is explained later.

Create the Department entity



Create *Models/Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the DB:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required. EF Core generally chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

The question mark (?) in the preceding code specifies the property is nullable.

A department may have many courses, so there's a Courses navigation property:

```
public ICollection<Course> Courses { get; set; }
```

Note: By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. Cascading delete can result in circular cascade delete rules. Circular cascade delete rules causes an exception when a migration is added.

For example, if the `Department.InstructorID` property was defined as non-nullable:

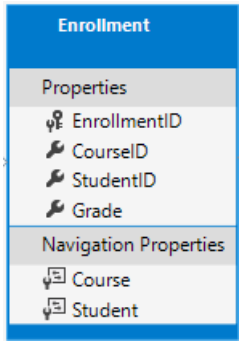
- EF Core configures a cascade delete rule to delete the department when the instructor is deleted.
- Deleting the department when the instructor is deleted isn't the intended behavior.
- The following fluent API would set a restrict rule instead of cascade.

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

The preceding code disables cascade delete on the department-instructor relationship.

Update the Enrollment entity

An enrollment record is for one course taken by one student.



Update *Models/Enrollment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

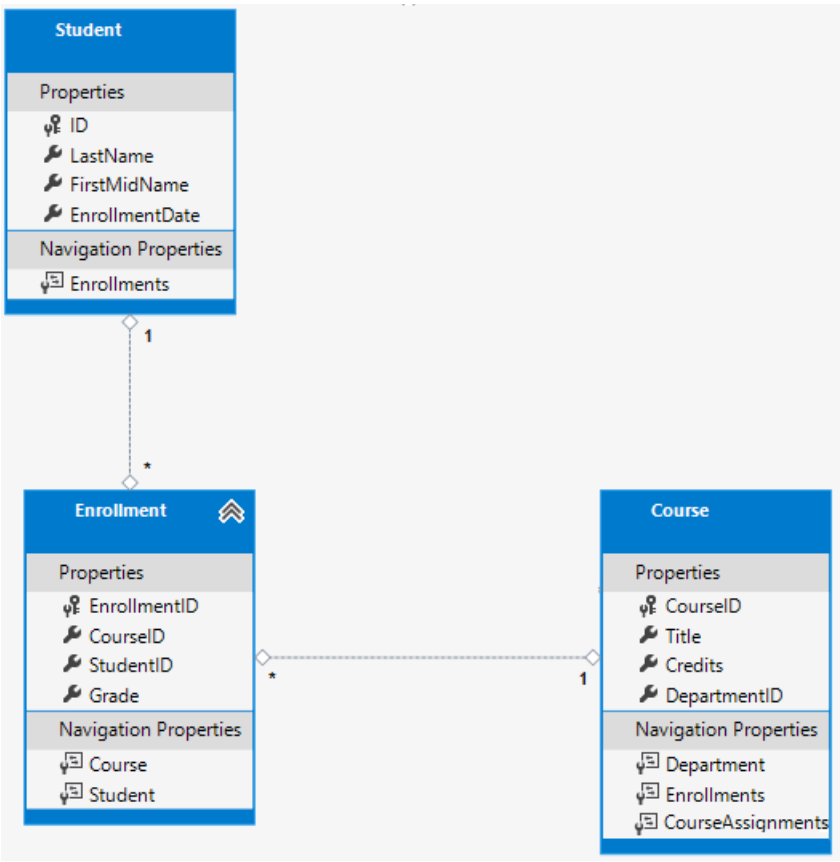
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities. The `Enrollment` entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the `Enrollment` table contains additional data besides FKs for the joined tables (in this case, the PK and `Grade`).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



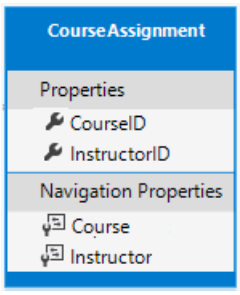
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs (`CourseID` and `StudentID`). A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a pure join table.

Note: EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see Many-to-many relationships in EF Core 2.0.

The CourseAssignment entity

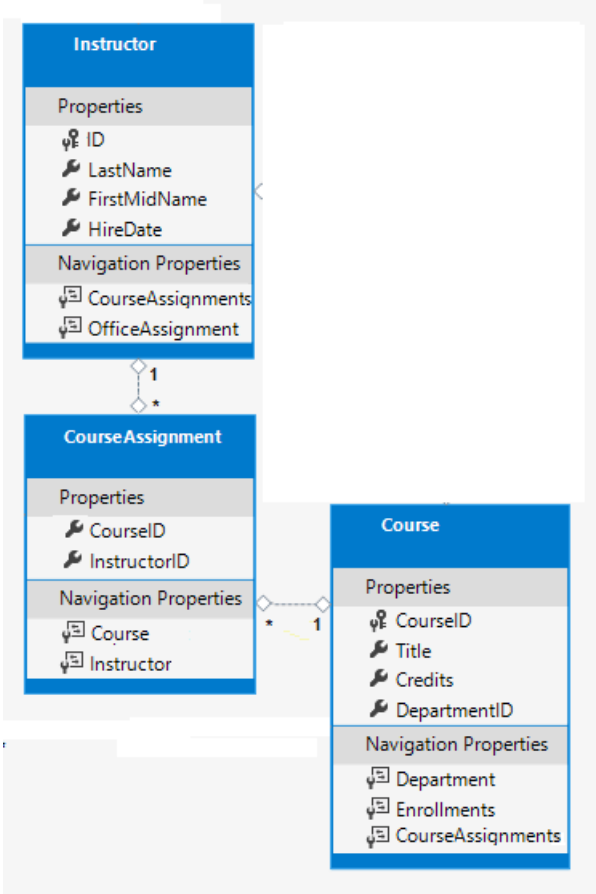


Create *Models/CourseAssignment.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Instructor-to-Courses



The Instructor-to-Courses many-to-many relationship:

- Requires a join table that must be represented by an entity set.
- Is a pure join table (table without payload).

It's common to name a join entity `EntityName1EntityName2`. For example, the Instructor-to-Courses join table using this pattern is `CourseInstructor`. However, we recommend using a name that describes the relationship.

Data models start out simple and grow. No-payload joins (PJTs) frequently evolve to include payload. By starting with a descriptive entity name, the name doesn't need to change when the join table changes. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked with a join entity called Ratings. For the Instructor-to-Courses many-to-many relationship, `CourseAssignment` is preferred over `CourseInstructor`.

Composite key

FKs are not nullable. The two FKs in `CourseAssignment` (`InstructorID` and `CourseID`) together uniquely identify each row of the `CourseAssignment` table. `CourseAssignment` doesn't require a dedicated PK. The `InstructorID` and `CourseID` properties function as a composite PK. The only way to specify composite PKs to EF Core is with the *fluent API*. The next section shows how to configure the composite PK.

The composite key ensures:

- Multiple rows are allowed for one course.
- Multiple rows are allowed for one instructor.
- Multiple rows for the same instructor and course isn't allowed.

The `Enrollment` join entity defines its own PK, so duplicates of this sort are possible. To prevent such duplicates:

- Add a unique index on the FK fields, or
- Configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the DB context

Add the following highlighted code to *Data/SchoolContext.cs*:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Models
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollment { get; set; }
        public DbSet<Student> Student { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

The preceding code adds the new entities and configures the `CourseAssignment` entity's composite PK.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is

called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

In this tutorial, the fluent API is used only for DB mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API (specifying a composite PK). There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

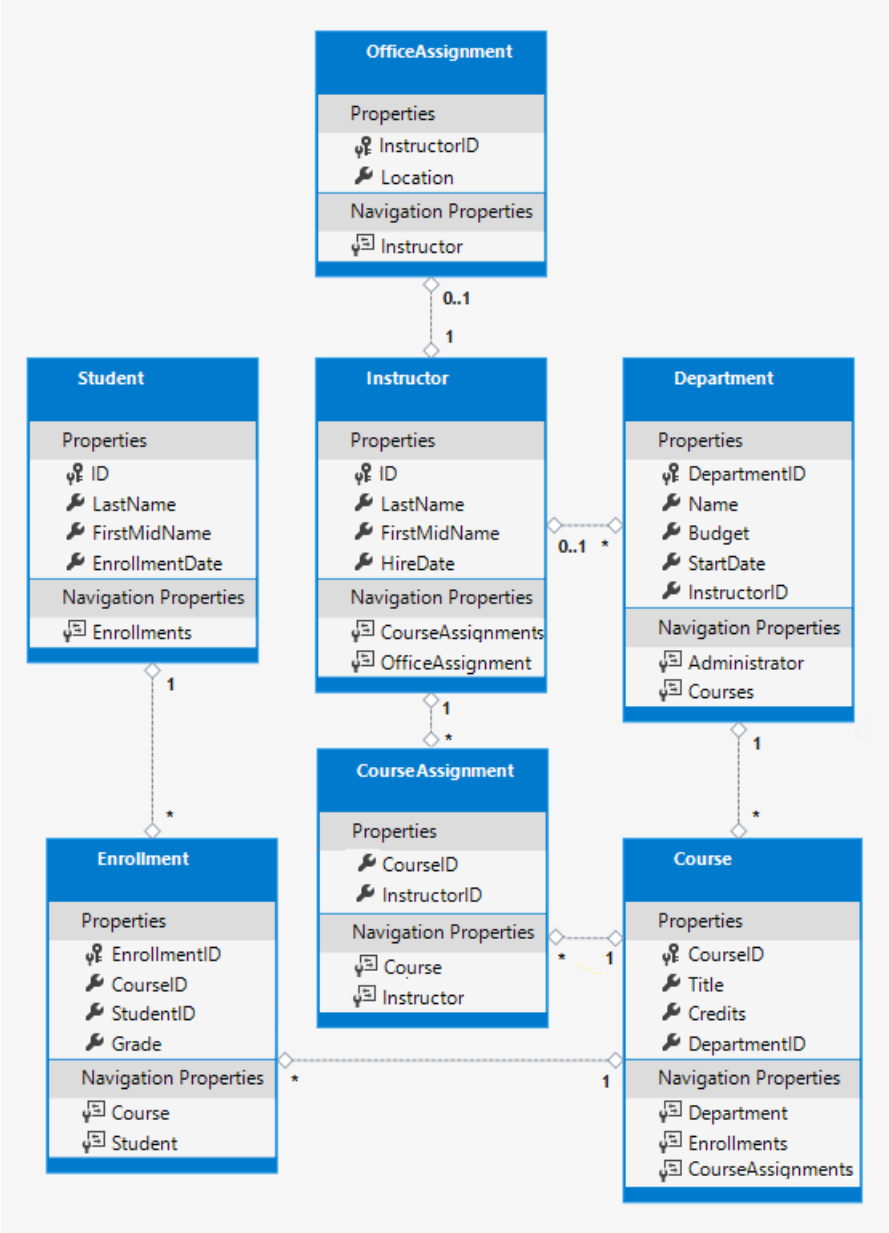
Some of the attributes used in the this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

The following illustration shows the diagram that EF Power Tools create for the completed School model.



The preceding diagram shows:

- Several one-to-many relationship lines (1 to *).
- The one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment` entities.
- The zero-or-one-to-many relationship line (0..1 to *) between the `Instructor` and `Department` entities.

Seed the DB with Test Data

Update the code in *Data/DbInitializer.cs*:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();
```

```
            // Look for any students.
            if (context.Student.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

            foreach (Student s in students)
            {
                context.Student.Add(s);
            }
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            foreach (Instructor i in instructors)
            {
                context.Instructors.Add(i);
            }
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
            };

            foreach (Department d in departments)
```

```
{
    context.Departments.Add(d);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
    },
    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
    new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
};

foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var officeAssignments = new OfficeAssignment[]
{
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
        Location = "Smith 17" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
        Location = "Gowan 27" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
        Location = "Thompson 304" },
};

foreach (OfficeAssignment o in officeAssignments)
{
    context.OfficeAssignments.Add(o);
}
context.SaveChanges();

var courseInstructors = new CourseAssignment[]
{
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
        },
```

```
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
        },
};

foreach (CourseAssignment ci in courseInstructors)
{
    context.CourseAssignments.Add(ci);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        Grade = Grade.A
    },
        new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        Grade = Grade.C
        },
        new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
        Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        Grade = Grade.B
        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
        Grade = Grade.B
        },
        new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
        Grade = Grade.B
        },
        new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
        },
        new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
        Grade = Grade.B
        },
```

```
                    },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").ID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollment.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollment.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing. See `Enrollments` and `CourseAssignments` for examples of how many-to-many join tables can be seeded.

Add a migration

Build the project.

- Visual Studio
- Visual Studio Code

```
Add-Migration ComplexDataModel
```

The preceding command displays a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If the `database update` command is run, the following error is produced:

```
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint
"FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in
database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.
```

Apply the migration

Now that you have an existing database, you need to think about how to apply future changes to it. This tutorial shows two approaches:

- Drop and re-create the database
- Apply the migration to the existing database. While this method is more complex and time-consuming, it's the preferred approach for real-world, production environments. **Note**: This is an optional section of the tutorial. You can do the drop and re-create steps and skip this section. If you do want to follow the steps in this section, don't do the drop and re-create steps.

Drop and re-create the database

The code in the updated `DbInitializer` adds seed data for the new entities. To force EF Core to create a new DB, drop and update the DB:

- Visual Studio
- Visual Studio Code

In the **Package Manager Console** (PMC), run the following command:

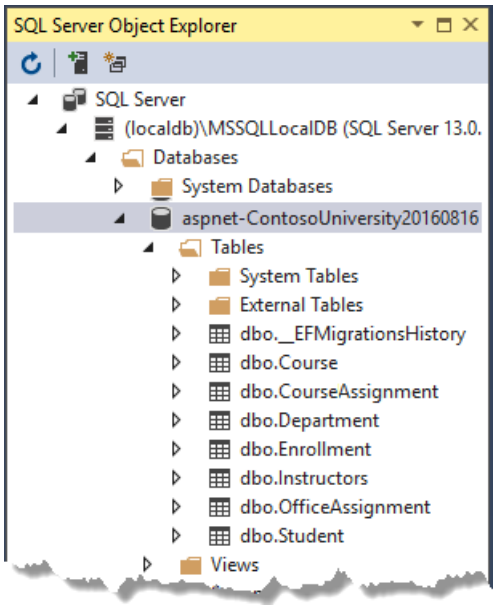
```
Drop-Database
Update-Database
```

Run `Get-Help about_EntityFrameworkCore` from the PMC to get help information.

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new DB.

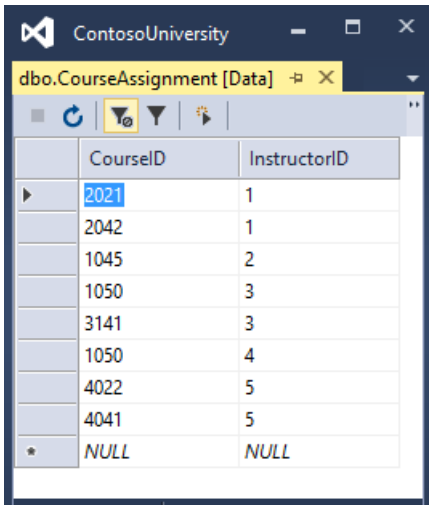
Open the DB in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.



Examine the **CourseAssignment** table:

- Right-click the **CourseAssignment** table and select **View Data**.
- Verify the **CourseAssignment** table contains data.



Apply the migration to the existing database

This section is optional. These steps work only if you skipped the preceding Drop and re-create the database section.

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. With production data, steps must be taken to migrate the existing data. This section provides an example of fixing FK constraint violations. Don't make these code changes without a backup. Don't make these code changes if you completed the previous section and updated the database.

The *{timestamp}_ComplexDataModel.cs* file contains the following code:

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

The preceding code adds a non-nullable `DepartmentID` FK to the `Course` table. The DB from the previous tutorial contains rows in `Course`, so that table cannot be updated by migrations.

To make the `ComplexDataModel` migration work with existing data:

- Change the code to give the new column (`DepartmentID`) a default value.
- Create a fake department named "Temp" to act as the default department.

Fix the foreign key constraints

Update the `ComplexDataModel` classes `Up` method:

- Open the *{timestamp}_ComplexDataModel.cs* file.
- Comment out the line of code that adds the `DepartmentID` column to the `Course` table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

Add the following highlighted code. The new code goes after the `.CreateTable( name: "Department"` block:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy", SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(type: "int", nullable: true),
        Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(type: "datetime2", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

 migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00, GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

With the preceding changes, existing `Course` rows will be related to the "Temp" department after the `ComplexDataModel` `Up` method runs.

A production app would:

- Include code or scripts to add `Department` rows and related `Course` rows to the new `Department` rows.
- Not use the "Temp" department or the default value for `Course.DepartmentID`.

The next tutorial covers related data.

Additional resources

- YouTube version of this tutorial(Part 1)
- YouTube version of this tutorial(Part 2)

PREVIOUS NEXT

Part 6, Razor Pages with EF Core in ASP.NET Core - Read Related Data

9/22/2020 • 28 minutes to read • Edit Online

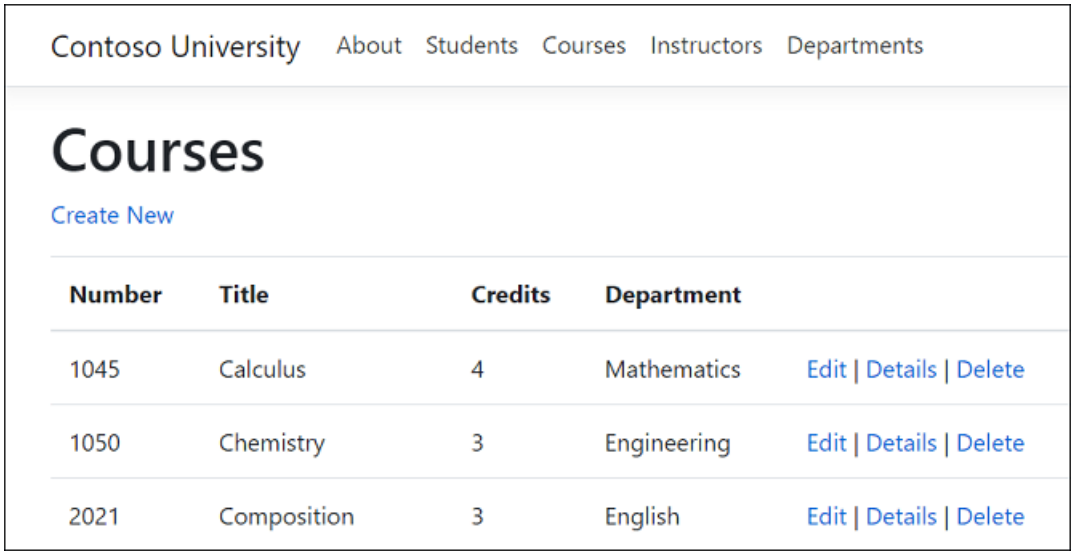
By Tom Dykstra, Jon P Smith, and Rick Anderson

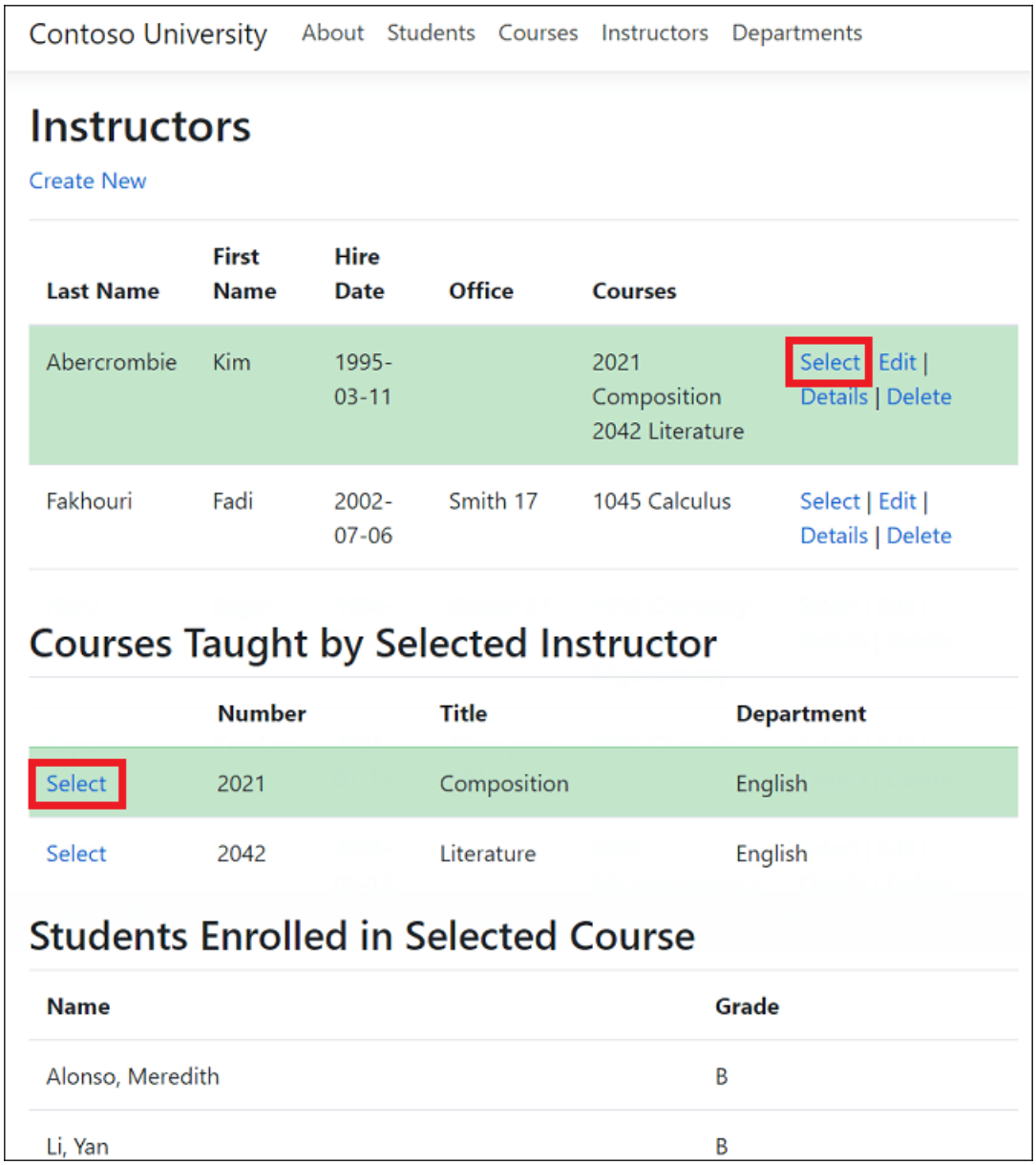
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to read and display related data. Related data is data that EF Core loads into navigation properties.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy loading

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When an entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than a giant single query. Eager loading is specified with the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

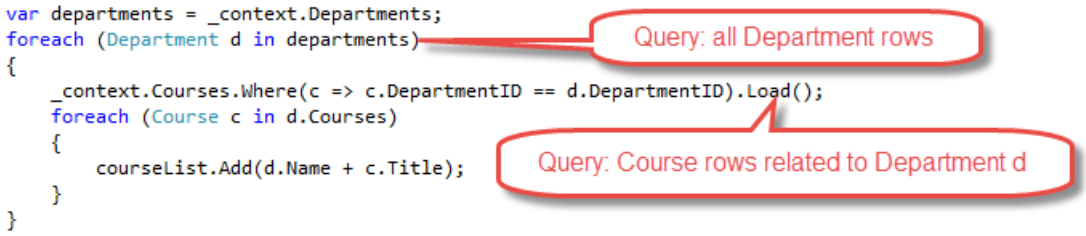
Query: all Department entities and related Course entities

  Eager loading sends multiple queries when a collection navigation is included:

  - One query for the main query
  - One query for each collection "edge" in the load tree.

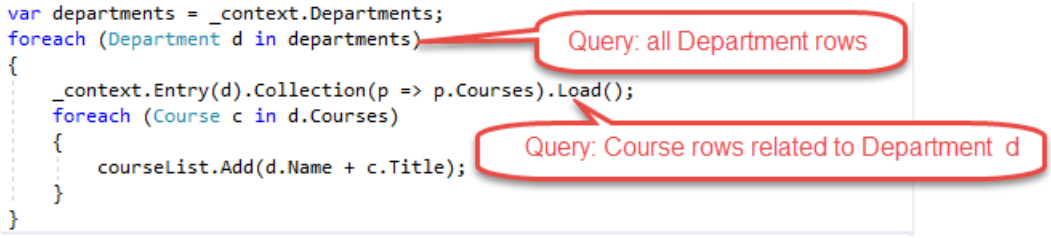
- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "Fixes up" means that EF Core automatically populates the navigation properties.

Separate queries with `Load` is more like explicit loading than eager loading.



Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

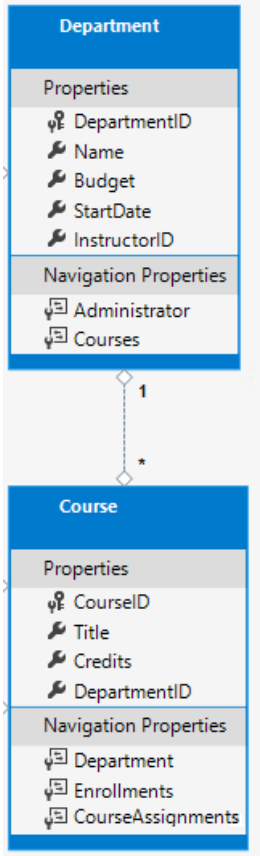
- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the database. With explicit loading, the code specifies the navigation properties to be loaded. Use the `Load` method to do explicit loading. For example:



- Lazy loading. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the database each time a navigation property is accessed for the first time. Lazy loading can hurt performance, for example when developers use N+1 patterns, loading a parent and enumerating through children.

Create Course pages

The `Course` entity includes a navigation property that contains the related `Department` entity.



To display the name of the assigned department for a course:

- Load the related `Department` entity into the `Course.Department` navigation property.
- Get the name from the `Department` entity's `Name` property.

Scaffold Course pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold Student pages with the following exceptions:
  - Create a *Pages/Courses* folder.
  - Use `Course` for the model class.
  - Use the existing context class instead of creating a new one.

- Open *Pages/Courses/Index.cshtml.cs* and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

- Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Display the department name

Update Pages/Courses/Index.cshtml.cs with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IList<Course> Courses { get; set; }

        public async Task OnGetAsync()
        {
            Courses = await _context.Courses
                .Include(c => c.Department)
                .AsNoTracking()
                .ToListAsync();
        }
    }
}
```

The preceding code changes the `Course` property to `Courses` and adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities don't need to be tracked because they're not updated in the current context.

Update *Pages/Courses/Index.cshtml* with the following code.

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel

@{
    ViewData["Title"] = "Courses";
}

<h1>Courses</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Courses)
{
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.CourseID)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Credits)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Department.Name)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

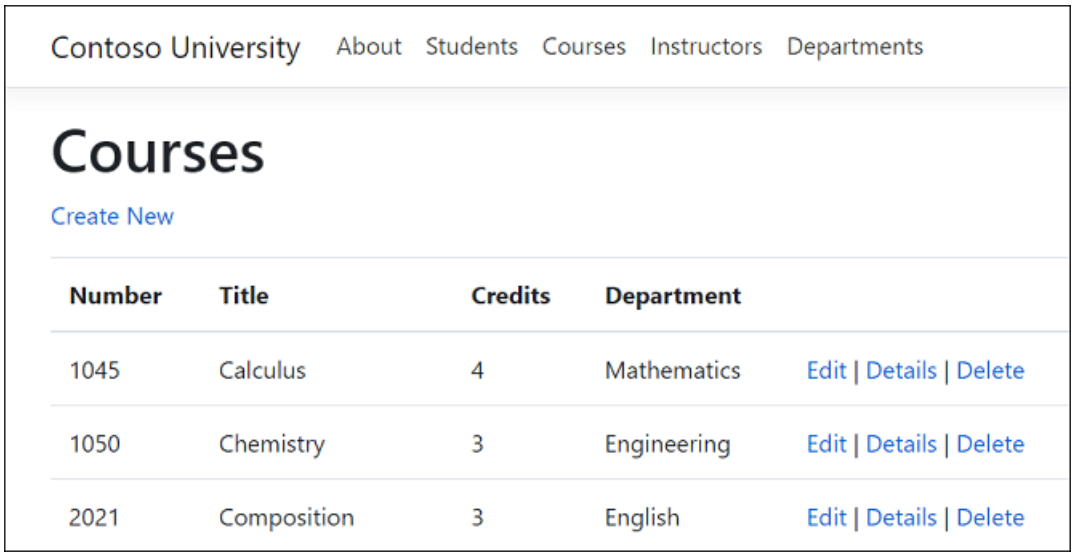
The following changes have been made to the scaffolded code:

- Changed the `Course` property name to `Courses`.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- Changed the **Department** column to display the department name. The code displays the `Name` property of

the `Department` entity that's loaded into the `Department` navigation property:

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method. The `Select` method is an alternative that loads only the related data needed. For single items, like the `Department.Name` it uses a SQL INNER JOIN. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
            .Select(p => new CourseViewModel
            {
                CourseID = p.CourseID,
                Title = p.Title,
                Credits = p.Credits,
                DepartmentName = p.Department.Name
            }).ToListAsync();
}
```

The preceding code doesn't return any entity types, therefore no tracking is done. For more information about the EF tracking, see Tracking vs. No-Tracking Queries.

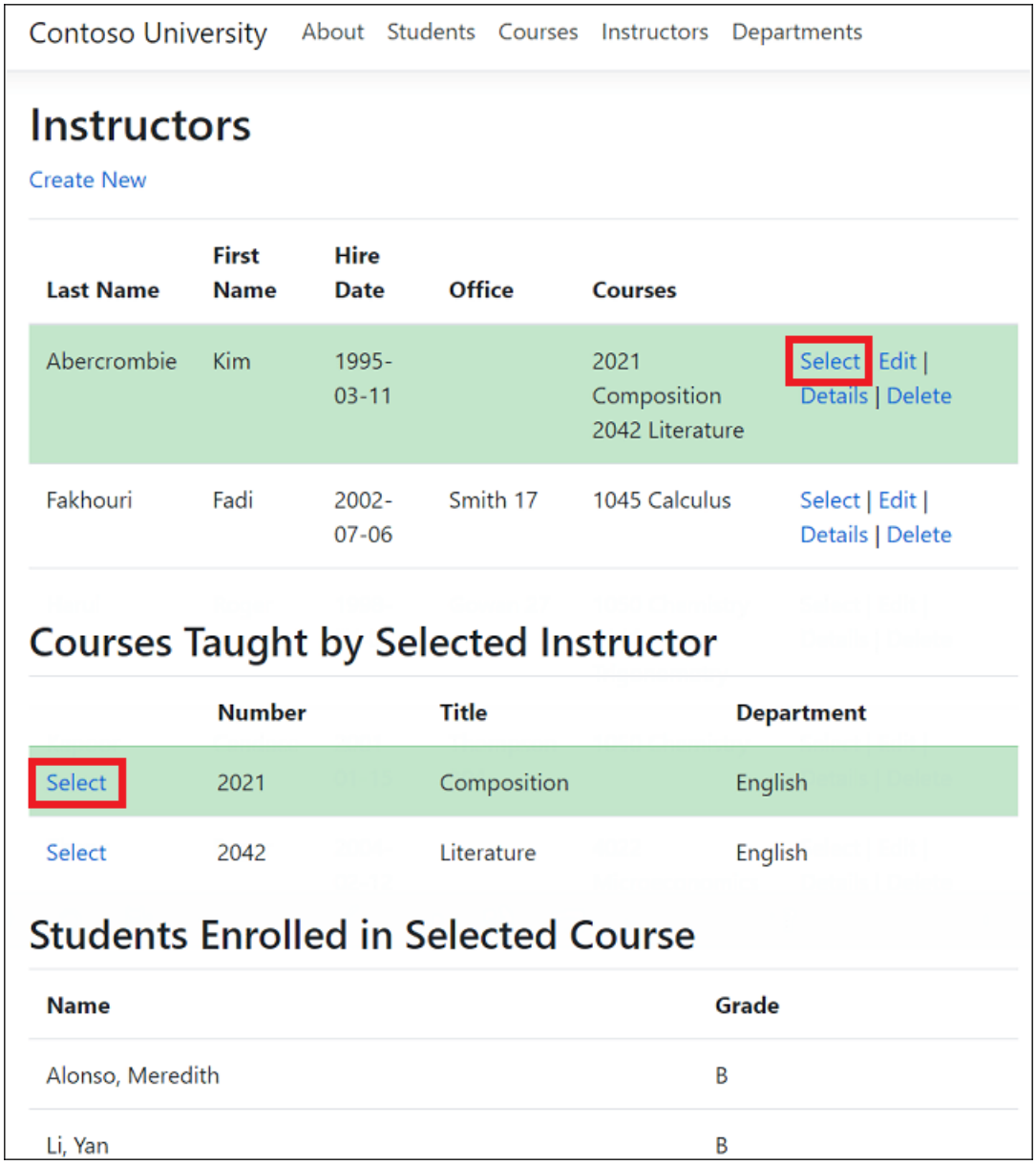
The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelect.cshtml and IndexSelect.cshtml.cs for a complete example.

Create Instructor pages

This section scaffolds Instructor pages and adds related Courses and Enrollments to the Instructors Index page.



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model

The instructors page shows data from three different tables. A view model is needed that includes three properties representing the three tables.

Create *SchoolViewModels/InstructorIndexData.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Scaffold Instructor pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold the student pages with the following exceptions:
  - Create a *Pages/Instructors* folder.
  - Use `Instructor` for the model class.
  - Use the existing context class instead of creating a new one.

To see what the scaffolded page looks like before you update it, run the app and navigate to the Instructors page.

Update *Pages/Instructors/Index.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Department)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Enrollments)
                            .ThenInclude(i => i.Student)
                .AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
                Instructor instructor = InstructorData.Instructors
                    .Where(i => i.ID == id.Value).Single();
                InstructorData.Courses = instructor.CourseAssignments.Select(s => s.Course);
            }

            if (courseID != null)
            {
                CourseID = courseID.Value;
                var selectedCourse = InstructorData.Courses
                    .Where(x => x.CourseID == courseID).Single();
                InstructorData.Enrollments = selectedCourse.Enrollments;
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the *Pages/Instructors/Index.cshtml.cs* file:

```
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The code specifies eager loading for the following navigation properties:

- `Instructor.OfficeAssignment`
- `Instructor.CourseAssignments`
  - `CourseAssignments.Course`
    - `Course.Department`
    - `Course.Enrollments`
      - `Enrollment.Student`

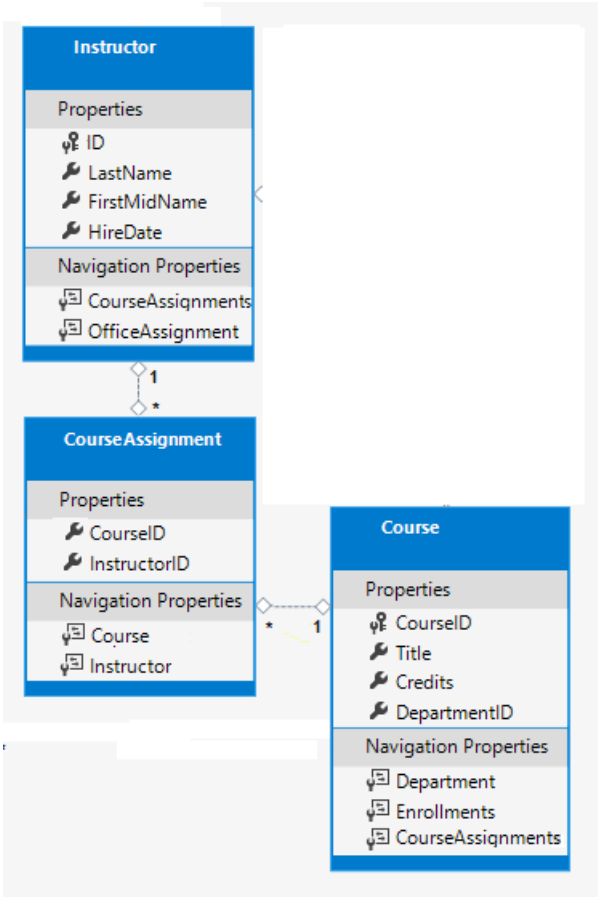
Notice the repetition of `Include` and `ThenInclude` methods for `CourseAssignments` and `Course`. This repetition is necessary to specify eager loading for two navigation properties of the `Course` entity.

The following code executes when an instructor is selected (`id != null`).

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = InstructorData.Instructors
        .Where(i => i.ID == id.Value).Single();
    InstructorData.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

The `Where` method returns a collection. But in this case, the filter will select a single entity, so the `Single` method is called to convert the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `CourseAssignments` property. `CourseAssignments` provides access to the related `Course` entities.



The `Single` method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    var selectedCourse = InstructorData.Courses
        .Where(x => x.CourseID == courseID).Single();
    InstructorData.Enrollments = selectedCourse.Enrollments;
}
```

Update the instructors Index page

Update *Pages/Instructors/Index.cshtml* with the following code.

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
```

```
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.InstructorData.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.InstructorData.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.InstructorData.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
```

```
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}

@if (Model.InstructorData.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.InstructorData.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `https://localhost:5001/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the URL is:

  `https://localhost:5001/Instructors/2`

- Adds an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

  ```
  @if (item.OfficeAssignment != null)
  {
      @item.OfficeAssignment.Location
  }
  ```

- Adds a **Courses** column that displays courses taught by each instructor. See Explicit line transition for more about this razor syntax.

- Adds code that dynamically adds `class="success"` to the `tr` element of the selected instructor and course. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Adds a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

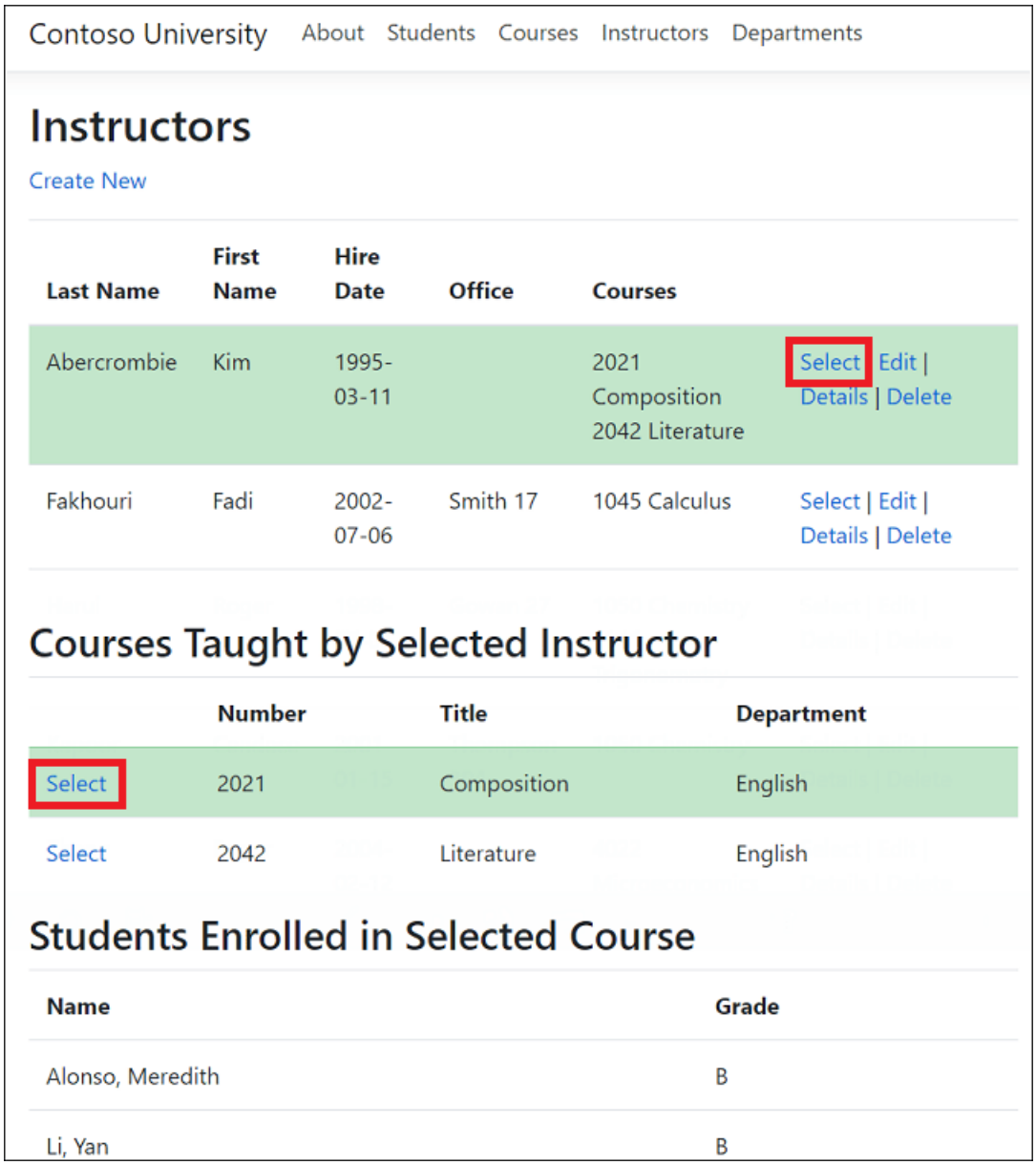
```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

- Adds a table of courses for the selected Instructor.
- Adds a table of student enrollments for the selected course.

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If `OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link for an instructor. The row style changes and courses assigned to that instructor are displayed.

Select a course to see the list of enrolled students and their grades.



Using Single

The `Single` method can pass in the `Where` condition instead of calling the `Where` method separately:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    InstructorData = new InstructorIndexData();

    InstructorData.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Enrollments)
                        .ThenInclude(i => i.Student)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = InstructorData.Instructors.Single(
            i => i.ID == id.Value);
        InstructorData.Courses = instructor.CourseAssignments.Select(
            s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        InstructorData.Enrollments = InstructorData.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}
```

Use of `Single` with a Where condition is a matter of personal preference. It provides no benefits over using the `Where` method.

Explicit loading

The current code specifies eager loading for `Enrollments` and `Students`:

```
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

Suppose users rarely want to see enrollments in a course. In that case, an optimization would be to only load the enrollment data if it's requested. In this section, the `OnGetAsync` is updated to use explicit loading of `Enrollments` and `Students`.

Update *Pages/Instructors/Index.cshtml.cs* with the following code.

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Department)
                //.Include(i => i.CourseAssignments)
                //    .ThenInclude(i => i.Course)
                //        .ThenInclude(i => i.Enrollments)
                //            .ThenInclude(i => i.Student)
                //.AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
                Instructor instructor = InstructorData.Instructors
                    .Where(i => i.ID == id.Value).Single();
                InstructorData.Courses = instructor.CourseAssignments.Select(s => s.Course);
            }

            if (courseID != null)
            {
                CourseID = courseID.Value;
                var selectedCourse = InstructorData.Courses
                    .Where(x => x.CourseID == courseID).Single();
                await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
                foreach (Enrollment enrollment in selectedCourse.Enrollments)
                {
                    await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
                }
                InstructorData.Enrollments = selectedCourse.Enrollments;
            }
        }
    }
}
```

The preceding code drops the *ThenInclude* method calls for enrollment and student data. If a course is selected, the explicit loading code retrieves:

- The `Enrollment` entities for the selected course.

- The `Student` entities for each `Enrollment`.

Notice that the preceding code comments out `.AsNoTracking()`. Navigation properties can only be explicitly loaded for tracked entities.

Test the app. From a user's perspective, the app behaves identically to the previous version.

Next steps

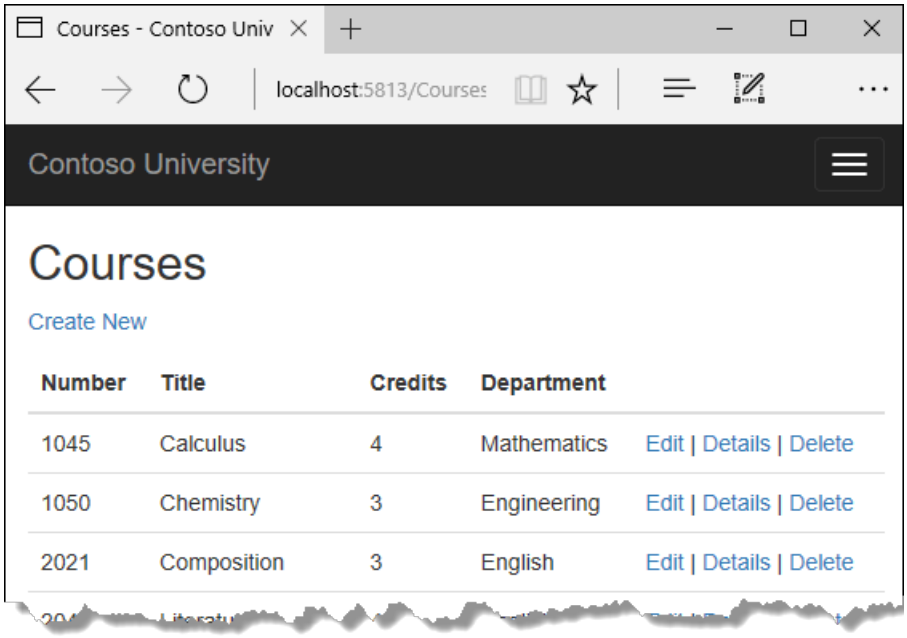
The next tutorial shows how to update related data.

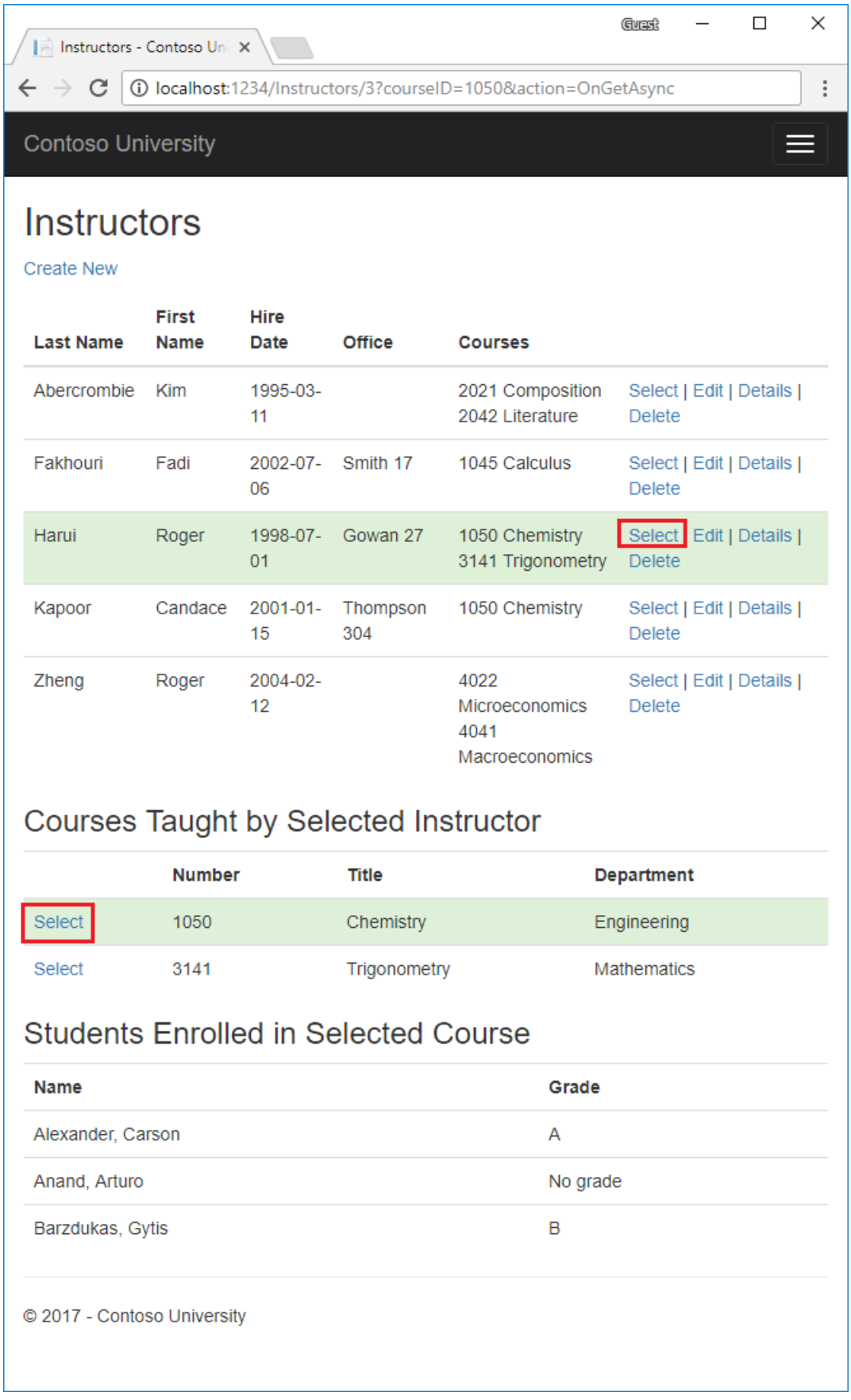
PREVIOUS TUTORIAL | NEXT TUTORIAL

In this tutorial, related data is read and displayed. Related data is data that EF Core loads into navigation properties.

If you run into problems you can't solve, download or view the completed app. Download instructions.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy Loading of related data

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When the entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than was the case for some queries in EF6 where there was a single query.

Eager loading is specified with the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

Eager loading sends multiple queries when a collection navigation is included:

- One query for the main query
- One query for each collection "edge" in the load tree.

- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "fixes up" means that EF Core automatically populates the navigation properties. Separate queries with `Load` is more like explicit loading than eager loading.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the DB. With explicit loading, the code specifies the navigation properties to be loaded. Use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Lazy loading. Lazy loading was added to EF Core in version 2.1. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the DB each time a navigation property is accessed for the first time.

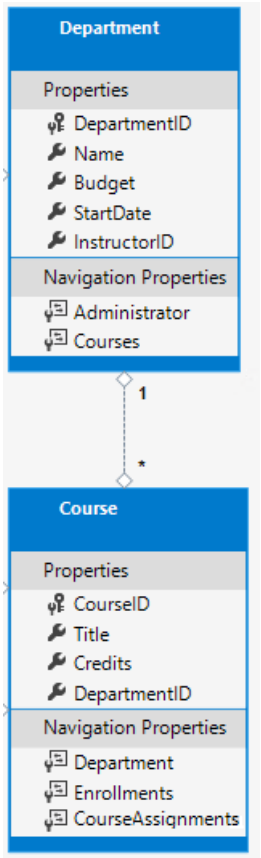
- The `Select` operator loads only the related data needed.

Create a Course page that displays department name

The Course entity includes a navigation property that contains the `Department` entity. The `Department` entity contains the department that the course is assigned to.

To display the name of the assigned department in a list of courses:

- Get the `Name` property from the `Department` entity.
- The `Department` entity comes from the `Course.Department` navigation property.



Scaffold the Course model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Course` for the model class.

The preceding command scaffolds the `Course` model. Open the project in Visual Studio.

Open *Pages/Courses/Index.cshtml.cs* and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The preceding code adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities are not tracked because they're not updated in the current context.

Update *Pages/Courses/Index.cshtml* with the following highlighted markup:

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel
@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

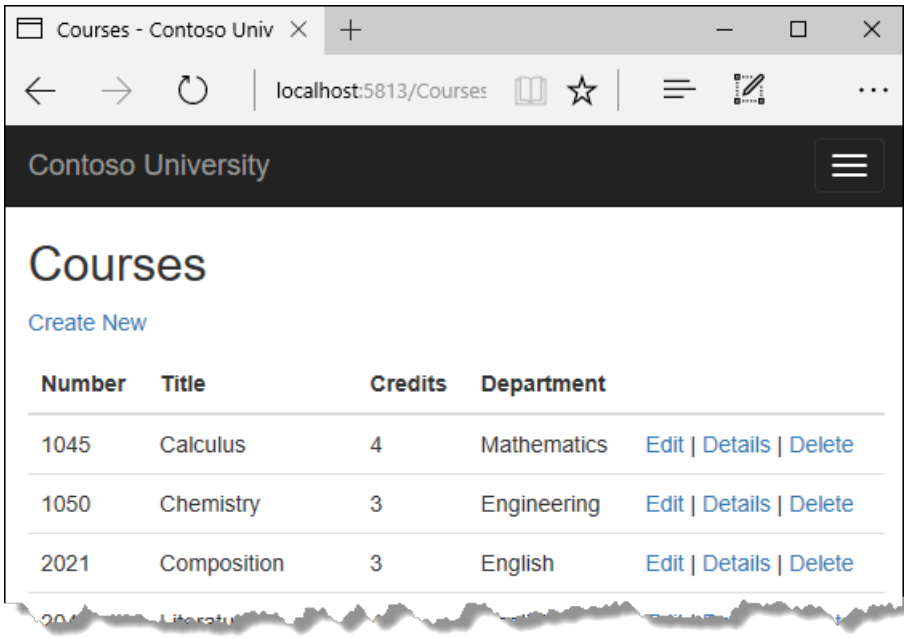
<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Course)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The following changes have been made to the scaffolded code:

- Changed the heading from Index to Courses.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The `Select` operator loads only the related data needed. For single items, like the `Department.Name` it uses a SQL INNER JOIN. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
            .Select(p => new CourseViewModel
            {
                CourseID = p.CourseID,
                Title = p.Title,
                Credits = p.Credits,
                DepartmentName = p.Department.Name
            }).ToListAsync();
}
```

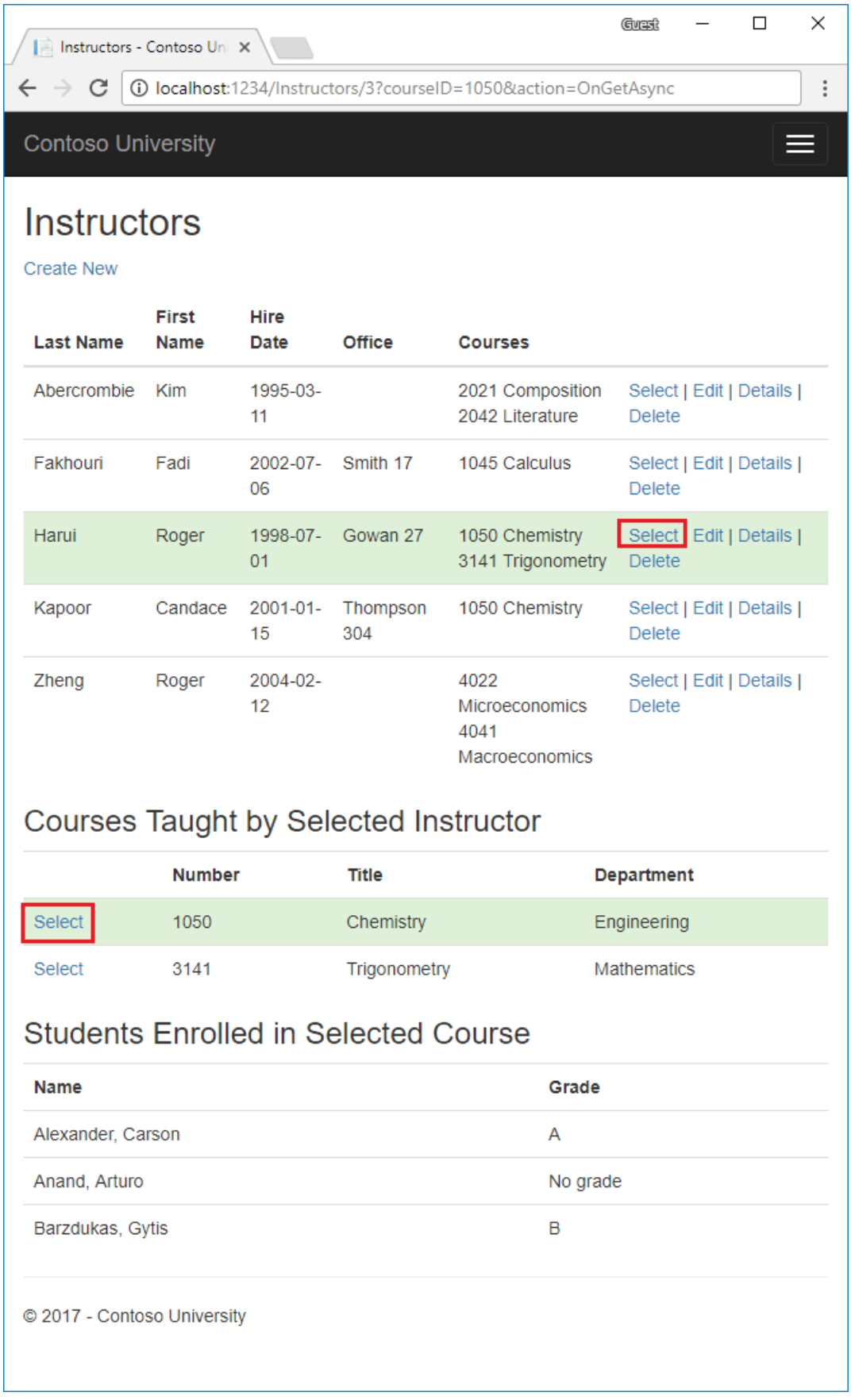
The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelect.cshtml and IndexSelect.cshtml.cs for a complete example.

Create an Instructors page that shows Courses and Enrollments

In this section, the Instructors page is created.



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor (Harui in the preceding image), related `Course` entities are displayed. The

`Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.

- When the user selects a course (Chemistry in the preceding image), related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model for the Instructor Index view

The instructors page shows data from three different tables. A view model is created that includes the three entities representing the three tables.

In the *SchoolViewModels* folder, create *InstructorIndexData.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Scaffold the Instructor model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Instructor` for the model class.

The preceding command scaffolds the `Instructor` model. Run the app and navigate to the instructors page.

Replace *Pages/Instructors/Index.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData Instructor { get; set; }
        public int InstructorID { get; set; }

        public async Task OnGetAsync(int? id)
        {
            Instructor = new InstructorIndexData();
            Instructor.Instructors = await _context.Instructors
                  .Include(i => i.OfficeAssignment)
                  .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                  .AsNoTracking()
                  .OrderBy(i => i.LastName)
                  .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the *Pages/Instructors/Index.cshtml.cs* file:

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The query has two includes:

- `OfficeAssignment`: Displayed in the instructors view.
- `CourseAssignments`: Which brings in the courses taught.

Update the instructors Index page

Update *Pages/Instructors/Index.cshtml* with the following markup:

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructor.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding markup makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `http://localhost:1234/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the previous URL is:

  `http://localhost:1234/Instructors/2`

- Page title is **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- Added a **Courses** column that displays courses taught by each instructor. See Explicit line transition for more about this razor syntax.
- Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Added a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link. The row style changes.

Add courses taught by selected instructor

Update the `OnGetAsync` method in *Pages/Instructors/Index.cshtml.cs* with the following code:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }
}
```

Add `public int CourseID { get; set; }`

```
public class IndexModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public IndexModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    public InstructorIndexData Instructor { get; set; }
    public int InstructorID { get; set; }
    public int CourseID { get; set; }

    public async Task OnGetAsync(int? id, int? courseID)
    {
        Instructor = new InstructorIndexData();
        Instructor.Instructors = await _context.Instructors
              .Include(i => i.OfficeAssignment)
              .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Department)
              .AsNoTracking()
              .OrderBy(i => i.LastName)
              .ToListAsync();

        if (id != null)
        {
            InstructorID = id.Value;
            Instructor instructor = Instructor.Instructors.Where(
                i => i.ID == id.Value).Single();
            Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
        }

        if (courseID != null)
        {
            CourseID = courseID.Value;
            Instructor.Enrollments = Instructor.Courses.Where(
                x => x.CourseID == courseID).Single().Enrollments;
        }
    }
```

Examine the updated query:

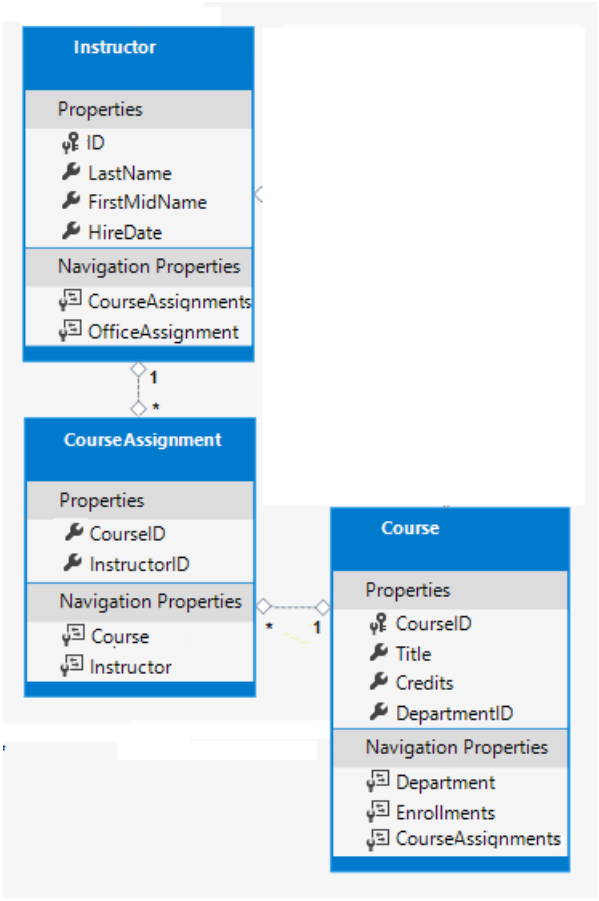
```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The preceding query adds the `Department` entities.

The following code executes when an instructor is selected (`id != null`). The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = Instructor.Instructors.Where(
        i => i.ID == id.Value).Single();
    Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection. In the preceding `Where` method, only a single `Instructor` entity is returned. The `Single` method converts the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `CourseAssignments` property. `CourseAssignments` provides access to the related `Course` entities.



The `Single` method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. Using `SingleOrDefault` on an empty collection:

- Results in an exception (from trying to find a `Courses` property on a null reference).
- The exception message would less clearly indicate the cause of the problem.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    Instructor.Enrollments = Instructor.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Add the following markup to the end of the *Pages/Instructors/Index.cshtml* Razor Page:

```
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.Instructor.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Instructor.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

The preceding markup displays a list of courses related to an instructor when an instructor is selected.

Test the app. Click on a **Select** link on the instructors page.

Show student data

In this section, the app is updated to show the student data for a selected course.

Update the query in the `OnGetAsync` method in *Pages/Instructors/Index.cshtml.cs* with the following code:

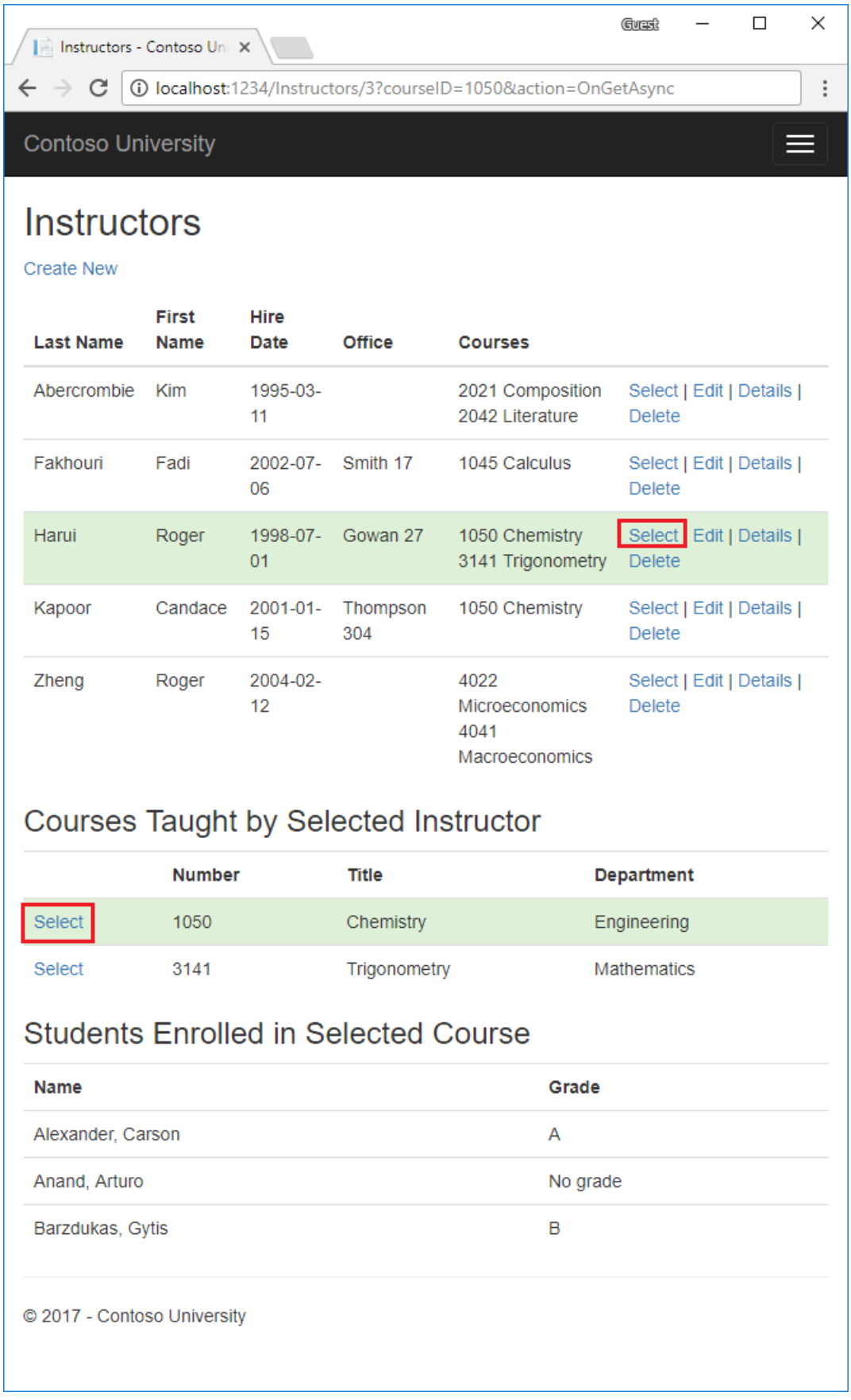
```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Update *Pages/Instructors/Index.cshtml*. Add the following markup to the end of the file:

```
@if (Model.Instructor.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Instructor.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding markup displays a list of the students who are enrolled in the selected course.

Refresh the page and select an instructor. Select a course to see the list of enrolled students and their grades.



Using Single

The `Single` method can pass in the `Where` condition instead of calling the `Where` method separately:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();

    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Enrollments)
                        .ThenInclude(i => i.Student)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Single(
            i => i.ID == id.Value);
        Instructor.Courses = instructor.CourseAssignments.Select(
            s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}
```

The preceding `Single` approach provides no benefits over using `Where`. Some developers prefer the `Single` approach style.

Explicit loading

The current code specifies eager loading for `Enrollments` and `Students`:

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Suppose users rarely want to see enrollments in a course. In that case, an optimization would be to only load the enrollment data if it's requested. In this section, the `OnGetAsync` is updated to use explicit loading of `Enrollments` and `Students`.

Update the `OnGetAsync` with the following code:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            //.Include(i => i.CourseAssignments)
            //    .ThenInclude(i => i.Course)
            //        .ThenInclude(i => i.Enrollments)
            //            .ThenInclude(i => i.Student)
         // .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();


    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        var selectedCourse = Instructor.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        Instructor.Enrollments = selectedCourse.Enrollments;
    }
}
```

The preceding code drops the *ThenInclude* method calls for enrollment and student data. If a course is selected, the highlighted code retrieves:

- The `Enrollment` entities for the selected course.
- The `Student` entities for each `Enrollment`.

Notice the preceding code comments out `.AsNoTracking()`. Navigation properties can only be explicitly loaded for tracked entities.

Test the app. From a users perspective, the app behaves identically to the previous version.

The next tutorial shows how to update related data.

Additional resources

- YouTube version of this tutorial (part1)
- YouTube version of this tutorial (part2)

Part 7, Razor Pages with EF Core in ASP.NET Core - Update Related Data

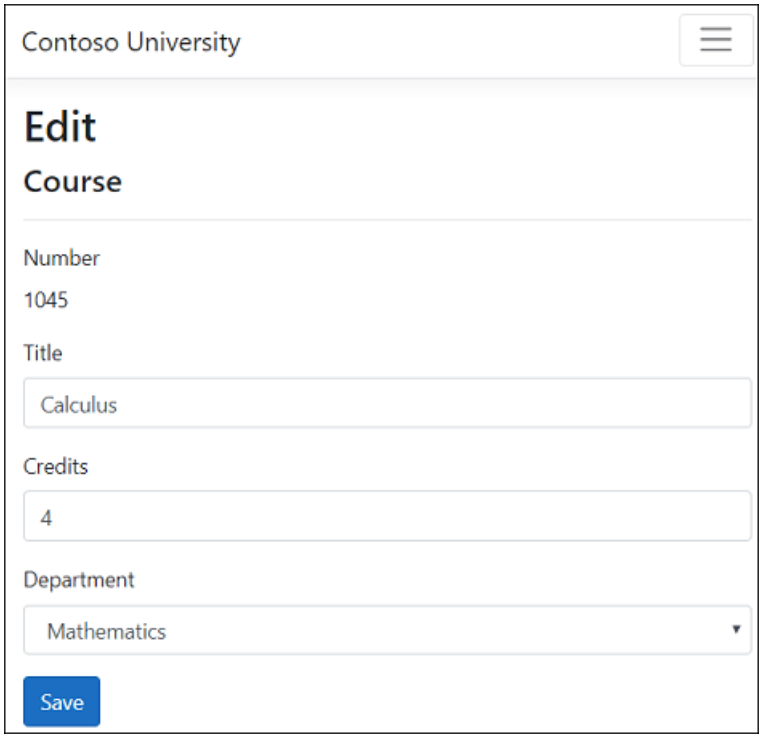
9/22/2020 • 32 minutes to read • Edit Online

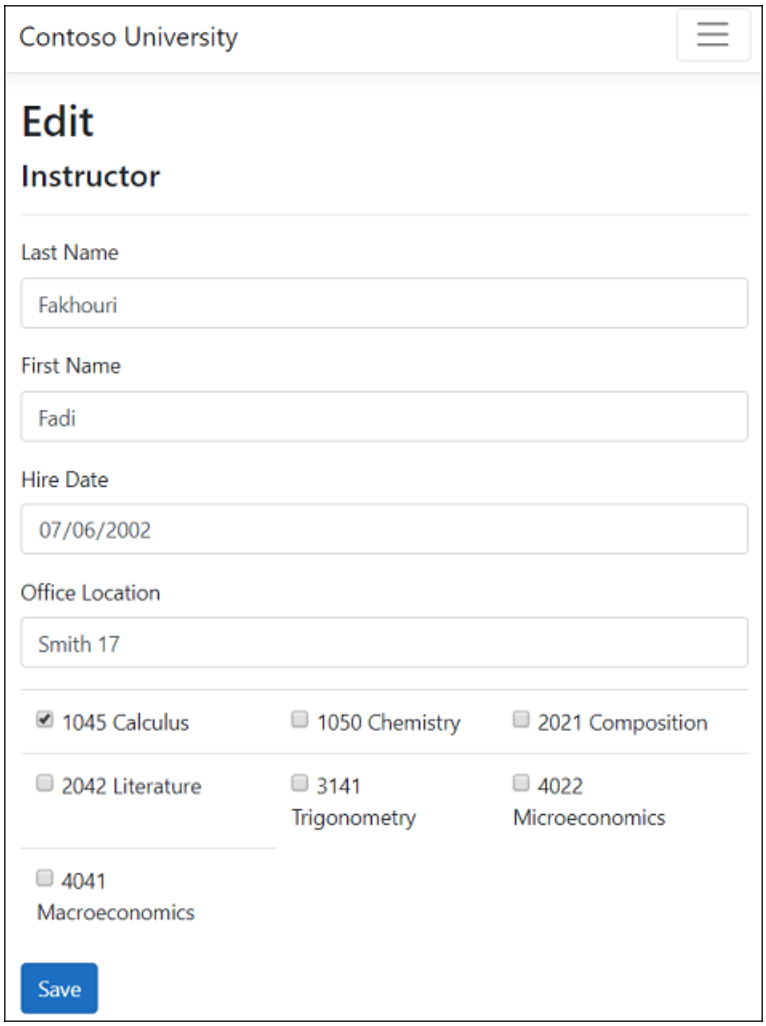
By Tom Dykstra, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to update related data. The following illustrations show some of the completed pages.





Update the Course Create and Edit pages

The scaffolded code for the Course Create and Edit pages has a Department drop-down list that shows Department ID (an integer). The drop-down should show the Department name, so both of these pages need a list of department names. To provide that list, use a base class for the Create and Edit pages.

Create a base class for Course Create and Edit

Create a *Pages/Courses/DepartmentNamePageModel.cs* file with the following code:

```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

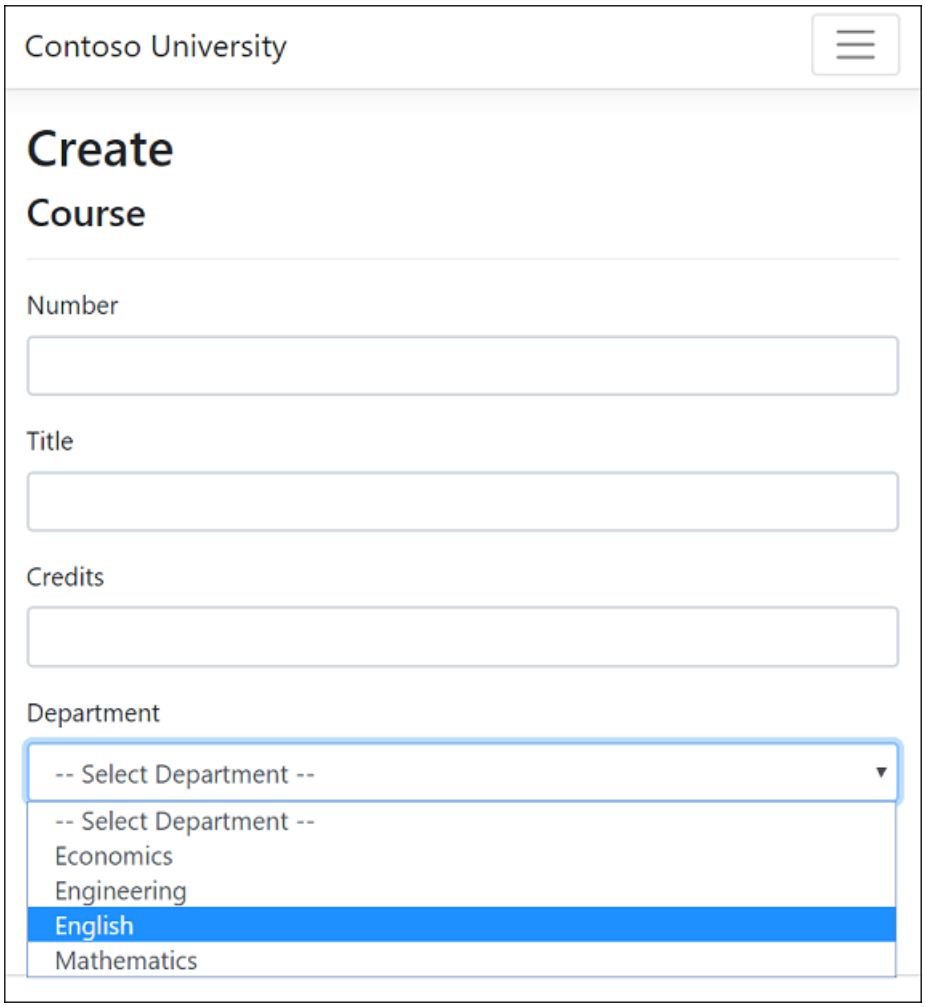
            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Update the Course Create page model

A Course is assigned to a Department. The base class for the Create and Edit pages provides a `SelectList` for selecting the department. The drop-down list that uses the `SelectList` sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update *Pages/Courses/Create.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses `TryUpdateModelAsync` to prevent overposting.
- Removes `ViewData["DepartmentID"]`. `DepartmentNameSL` from the base class is a strongly typed model and will be used by the Razor page. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Course Create Razor page

Update *Pages/Courses/Create.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" in the drop-down when no department has been selected yet, rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Course Edit page model

Update *Pages/Courses/Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (courseToUpdate == null)
            {
                return NotFound();
            }
```

```
            }

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which selects that department in the drop-down list.

Update the Course Edit Razor page

Update *Pages/Courses/Edit.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption for the Department drop-down from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for the course number to be included in the posted data when the user clicks **Save**.

Update the Course Details and Delete pages

AsNoTracking can improve performance when tracking isn't required.

Update the Course page models

Update *Pages/Courses/Delete.cshtml.cs* with the following code to add `AsNoTracking`:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .AsNoTracking()
                .Include(c => c.Department)
                .FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses.FindAsync(id);

            if (Course != null)
            {
                _context.Courses.Remove(Course);
                await _context.SaveChangesAsync();
            }

            return RedirectToPage("./Index");
        }
    }
}
```

Make the same change in the *Pages/Courses/Details.cshtml.cs* file:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class DetailsModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DetailsModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                 .AsNoTracking()
                 .Include(c => c.Department)
                 .FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }
            return Page();
        }
    }
}
```

Update the Course Razor pages

Update *Pages/Courses/Delete.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

```
@page
@model ContosoUniversity.Pages.Courses.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

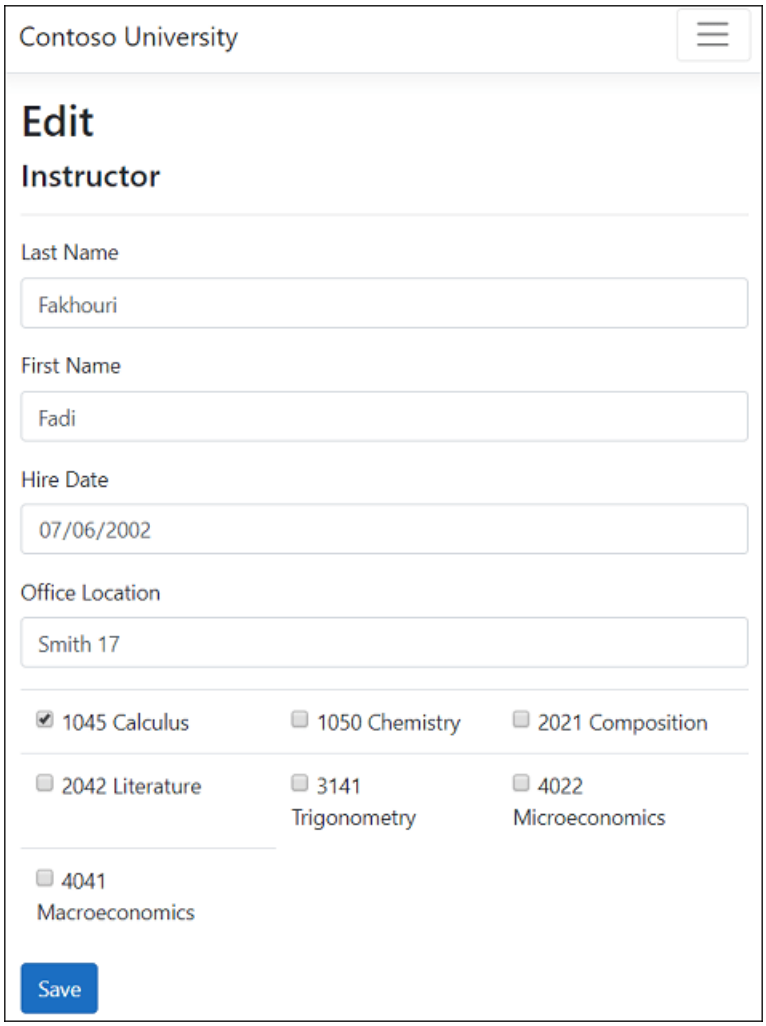
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Course.CourseID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

Test the Course pages

Test the create, edit, details, and delete pages.

Update the instructor Create and Edit pages

Instructors may teach any number of courses. The following image shows the instructor Edit page with an array of course checkboxes.



The checkboxes enable changes to courses an instructor is assigned to. A checkbox is displayed for every course in the database. Courses that the instructor is assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, a different UI might work better. But the method of managing a many-to-many relationship shown here wouldn't change. To create or delete relationships, you manipulate a join entity.

Create a class for assigned courses data

Create *SchoolViewModels/AssignedCourseData.cs* with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the check boxes for courses assigned to an instructor.

Create an Instructor page model base class

Create the *Pages/Instructors/InstructorCoursesPageModel.cs* base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
```

```
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            string[] selectedCourses, Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
            foreach (var course in context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.CourseAssignments.Add(
                            new CourseAssignment
                            {
                                InstructorID = instructorToUpdate.ID,
                                CourseID = course.CourseID
                            });
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        CourseAssignment courseToRemove
                            = instructorToUpdate
                                .CourseAssignments
                                .SingleOrDefault(i => i.CourseID == course.CourseID);
                        context.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class you will use for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used for efficient lookups.

Since the Razor page doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload with declared properties and `CourseAssignments` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```
if (selectedCourses == null)
{
    instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
    return;
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the page. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.CourseAssignments.Add(
            new CourseAssignment
            {
                InstructorID = instructorToUpdate.ID,
                CourseID = course.CourseID
            });
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        CourseAssignment courseToRemove
            = instructorToUpdate
                .CourseAssignments
                .SingleOrDefault(i => i.CourseID == course.CourseID);
        context.Remove(courseToRemove);
    }
}
```

Handle office location

Another relationship the edit page has to handle is the one-to-zero-or-one relationship that the Instructor entity has with the `OfficeAssignment` entity. The instructor edit code must handle the following scenarios:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.
- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the Instructor Edit page model

Update *Pages/Instructors/Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class EditModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            PopulateAssignedCourseData(_context, Instructor);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
        {
            if (id == null)
            {
                return NotFound();
            }

            var instructorToUpdate = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                .FirstOrDefaultAsync(s => s.ID == id);

            if (instructorToUpdate == null)
            {
```

```
            {
                return NotFound();
            }

            if (await TryUpdateModelAsync<Instructor>(
                instructorToUpdate,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                if (String.IsNullOrWhiteSpace(
                    instructorToUpdate.OfficeAssignment?.Location))
                {
                    instructorToUpdate.OfficeAssignment = null;
                }
                UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            PopulateAssignedCourseData(_context, instructorToUpdate);
            return Page();
        }
    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment`, `CourseAssignment`, and `CourseAssignment.Course` navigation properties.
- Updates the retrieved `Instructor` entity with values from the model binder. `TryUpdateModel` prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.
- Calls `PopulateAssignedCourseData` in `OnGetAsync` to provide information for the checkboxes using the `AssignedCourseData` view model class.
- Calls `UpdateInstructorCourses` in `OnPostAsync` to apply information from the checkboxes to the Instructor entity being edited.
- Calls `PopulateAssignedCourseData` and `UpdateInstructorCourses` in `OnPostAsync` if `TryUpdateModel` fails. These method calls restore the assigned course data entered on the page when it is redisplayed with an error message.

Update the Instructor Edit Razor page

Update *Pages/Instructors/Edit.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
```

```
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code creates an HTML table that has three columns. Each column has a checkbox and a caption containing the course number and title. The checkboxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each checkbox is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the checkboxes that are selected.

When the checkboxes are initially rendered, courses assigned to the instructor are selected.

Note: The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and

efficient.

Run the app and test the updated Instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Update the Instructor Create page

Update the Instructor Create page model and Razor page with code similar to the Edit page:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List<CourseAssignment>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
            var newInstructor = new Instructor();
            if (selectedCourses != null)
            {
                newInstructor.CourseAssignments = new List<CourseAssignment>();
                foreach (var course in selectedCourses)
                {
                    var courseToAdd = new CourseAssignment
                    {
                        CourseID = int.Parse(course)
                    };
                    newInstructor.CourseAssignments.Add(courseToAdd);
                }
            }

            if (await TryUpdateModelAsync<Instructor>(
                newInstructor,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel
```

```
@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>
```

```
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test the instructor Create page.

Update the Instructor Delete page

Update *Pages/Instructors/Delete.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            if (instructor == null)
            {
                return RedirectToPage("./Index");
            }

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `CourseAssignments` navigation property. `CourseAssignments` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

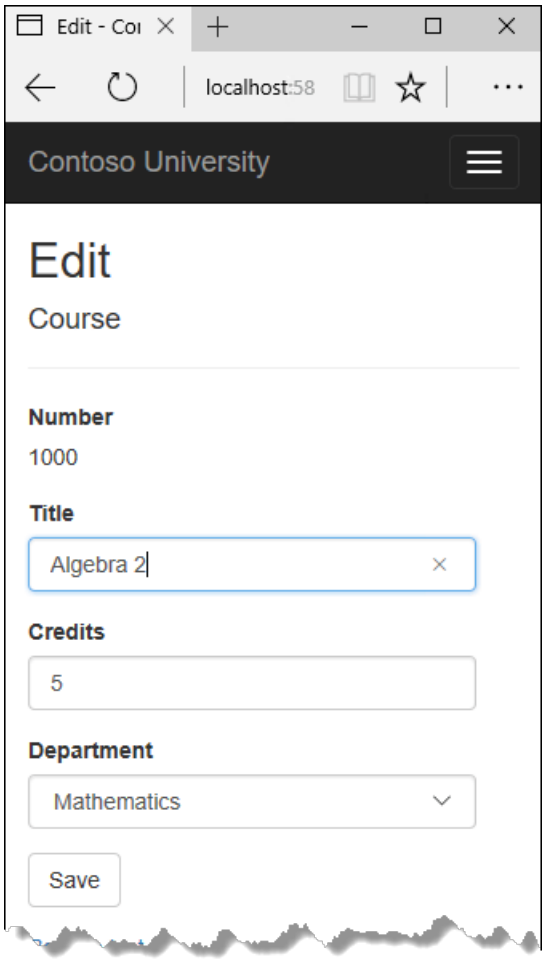
Run the app and test the Delete page.

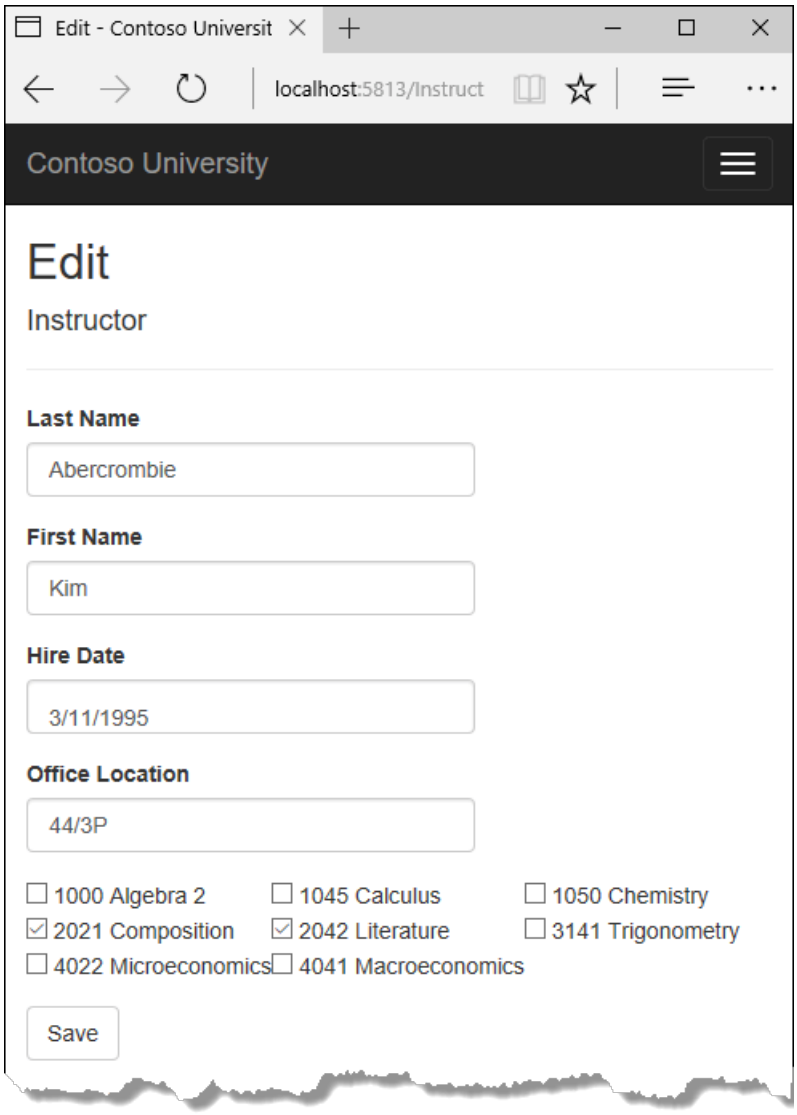
Next steps

PREVIOUS TUTORIAL NEXT TUTORIAL

This tutorial demonstrates updating related data. If you run into problems you can't solve, download or view the completed app. Download instructions.

The following illustrations shows some of the completed pages.





Examine and test the Create and Edit course pages. Create a new course. The department is selected by its primary key (an integer), not its name. Edit the new course. When you have finished testing, delete the new course.

Create a base class to share common code

The Courses/Create and Courses/Edit pages each need a list of department names. Create the *Pages/Courses/DepartmentNamePageModel.cshtml.cs* base class for the Create and Edit pages:

```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

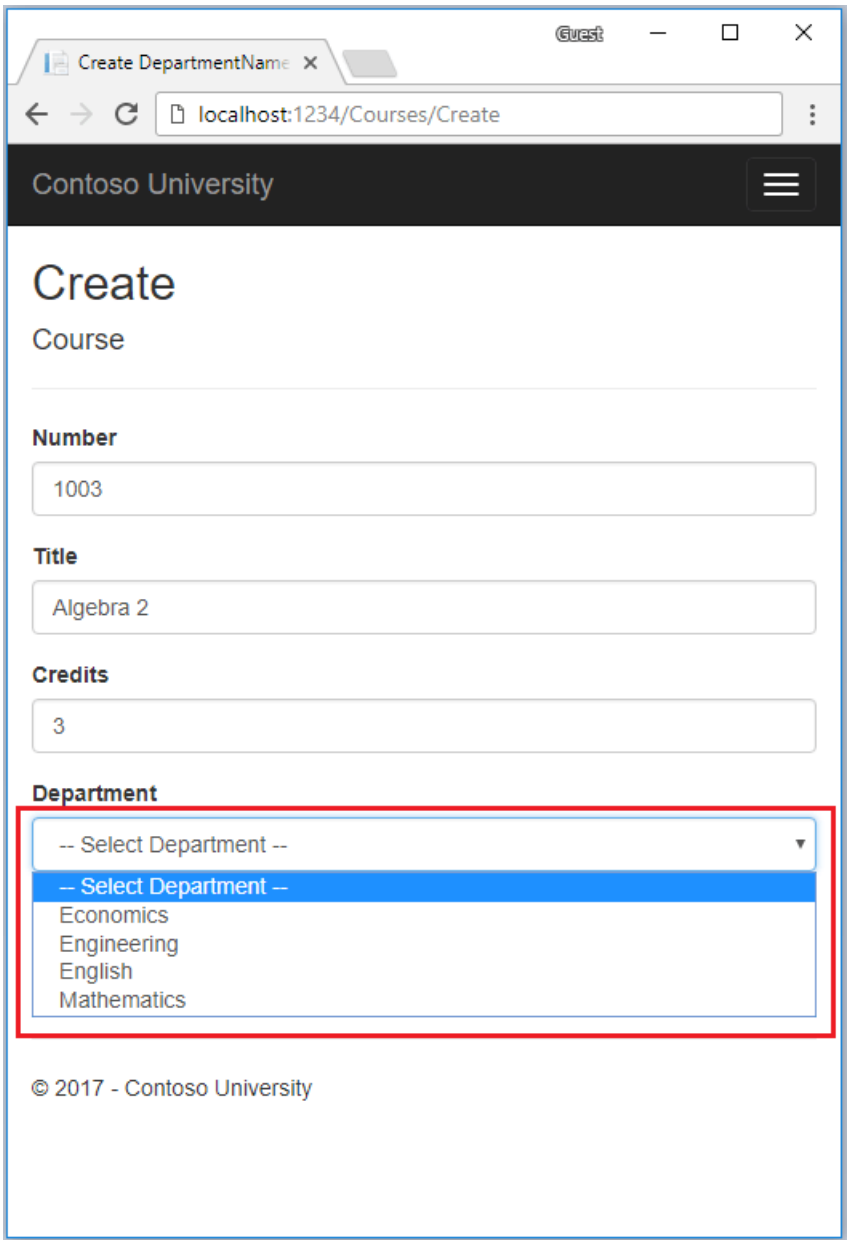
            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Customize the Courses Pages

When a new course entity is created, it must have a relationship to an existing department. To add a department while creating a course, the base class for Create and Edit contains a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update the Create page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses `TryUpdateModelAsync` to prevent overposting.
- Replaces `ViewData["DepartmentID"]` with `DepartmentNameSL` (from the base class).

`ViewData["DepartmentID"]` is replaced with the strongly typed `DepartmentNameSL`. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Courses Create page

Update *Pages/Courses/Create.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Courses Edit page.

Replace the code in *Pages/Courses/Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context,Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which select the department specified in the drop-

down list.

Update *Pages/Courses/Edit.cshtml* with the following markup:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for

the course number to be included in the posted data when the user clicks **Save**.

Test the updated code. Create, edit, and delete a course.

Add AsNoTracking to the Details and Delete page models

AsNoTracking can improve performance when tracking isn't required. Add `AsNoTracking` to the Delete and Details page model. The following code shows the updated Delete page model:

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Course Course { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .Include(c => c.Department)
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course != null)
        {
            _context.Courses.Remove(Course);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage("./Index");
    }
}
```

Update the `OnGetAsync` method in the *Pages/Courses/Details.cshtml.cs* file:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Course = await _context.Courses
         .AsNoTracking()
         .Include(c => c.Department)
         .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

Modify the Delete and Details pages

Update the Delete Razor page with the following markup:

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Department.DepartmentID)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

Test the Course pages

Test create, edit, details, and delete.

Update the instructor pages

The following sections update the instructor pages.

Add office location

When editing an instructor record, you may want to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity. The instructor code must handle:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.

- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the instructors Edit page model with the following code:

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");

    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property.
- Updates the retrieved `Instructor` entity with values from the model binder. `TryUpdateModel` prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.

Update the instructor Edit page

Update *Pages/Instructors/Edit.cshtml* with the office location:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

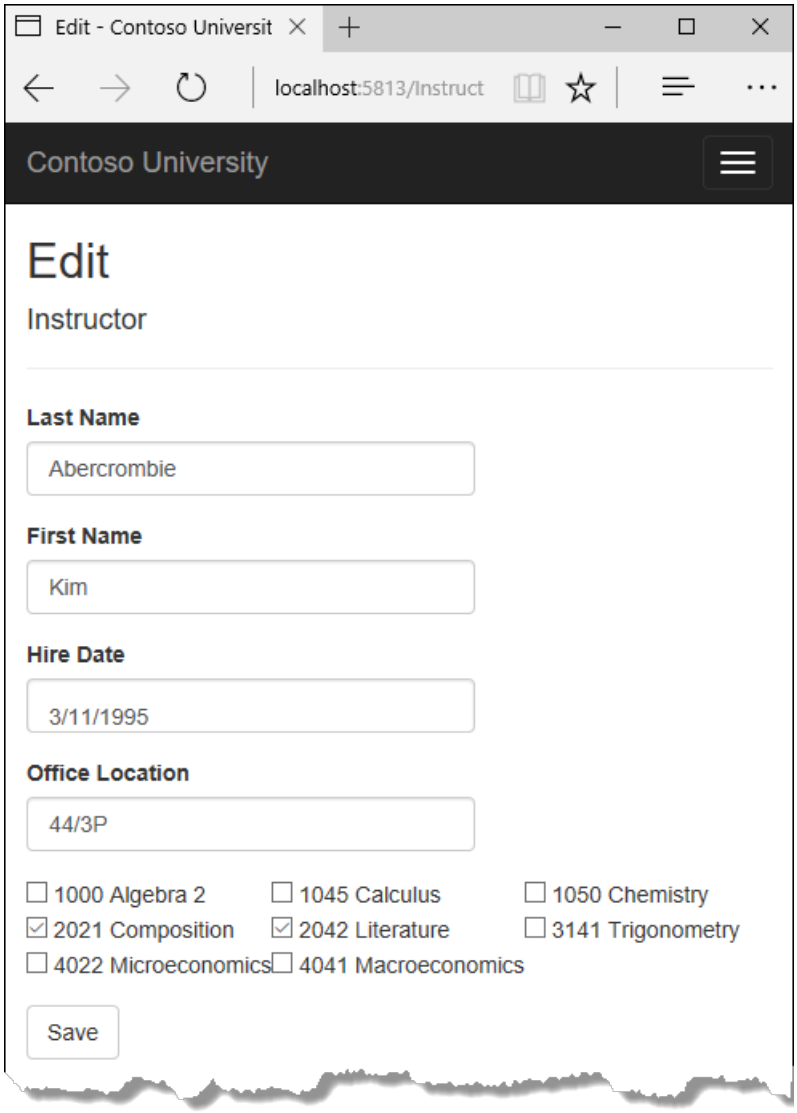
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Verify you can change an instructors office location.

Add Course assignments to the instructor Edit page

Instructors may teach any number of courses. In this section, you add the ability to change course assignments. The

following image shows the updated instructor Edit page:



`Course` and `Instructor` has a many-to-many relationship. To add and remove relationships, you add and remove entities from the `CourseAssignments` join entity set.

Check boxes enable changes to courses an instructor is assigned to. A check box is displayed for every course in the database. Courses that the instructor is assigned to are checked. The user can select or clear check boxes to change course assignments. If the number of courses were much greater:

- You'd probably use a different user interface to display the courses.
- The method of manipulating a join entity to create or delete relationships wouldn't change.

Add classes to support Create and Edit instructor pages

Create *SchoolViewModels/AssignedCourseData.cs* with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the check boxes for assigned courses by an instructor.

Create the *Pages/Instructors/InstructorCoursesPageModel.cshtml.cs* base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            string[] selectedCourses, Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
            foreach (var course in context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.CourseAssignments.Add(
                            new CourseAssignment
                            {
                                InstructorID = instructorToUpdate.ID,
                                CourseID = course.CourseID
                            });
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        CourseAssignment courseToRemove
                            = instructorToUpdate
                                .CourseAssignments
                                .SingleOrDefault(i => i.CourseID == course.CourseID);
```

```
                        context.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class you will use for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used to create efficient lookups.

Instructors Edit page model

Update the instructor Edit page model with the following code:

```
public class EditModel : InstructorCoursesPageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        PopulateAssignedCourseData(_context, Instructor);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(_context, instructorToUpdate);
        return Page();
    }
}
```

The preceding code handles office assignment changes.

Update the instructor Razor View:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
```

```
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

> **NOTE**
>
> When you paste the code in Visual Studio, line breaks are changed in a way that breaks the code. Press Ctrl+Z one time to undo the automatic formatting. Ctrl+Z fixes the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown. With the block of new code selected, press Tab three times to line up the new code with the existing code. Vote on or review the status of this bug with this link.

The preceding code creates an HTML table that has three columns. Each column has a check box and a caption containing the course number and title. The check boxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each check box is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the check boxes that are selected.

When the check boxes are initially rendered, courses assigned to the instructor have checked attributes.

Run the app and test the updated instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Note: The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Update the instructors Create page

Update the instructor Create page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List<CourseAssignment>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData( context, instructor);
```

```
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var newInstructor = new Instructor();
            if (selectedCourses != null)
            {
                newInstructor.CourseAssignments = new List<CourseAssignment>();
                foreach (var course in selectedCourses)
                {
                    var courseToAdd = new CourseAssignment
                    {
                        CourseID = int.Parse(course)
                    };
                    newInstructor.CourseAssignments.Add(courseToAdd);
                }
            }

            if (await TryUpdateModelAsync<Instructor>(
                newInstructor,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

The preceding code is similar to the *Pages/Instructors/Edit.cshtml.cs* code.

Update the instructor Create Razor page with the following markup:

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
```

```
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test the instructor Create page.

Update the Delete page

Update the Delete page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.SingleAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `CourseAssignments` navigation property. `CourseAssignments` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Additional resources

- YouTube version of this tutorial (Part 1)
- YouTube version of this tutorial (Part 2)

PREVIOUS NEXT

Part 8, Razor Pages with EF Core in ASP.NET Core - Concurrency

9/22/2020 • 36 minutes to read • Edit Online

By Rick Anderson, Tom Dykstra, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to handle conflicts when multiple users update an entity concurrently (at the same time).

Concurrency conflicts

A concurrency conflict occurs when:

- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the database.

If concurrency detection isn't enabled, whoever updates the database last overwrites the other user's changes. If this risk is acceptable, the cost of programming for concurrency might outweigh the benefit.

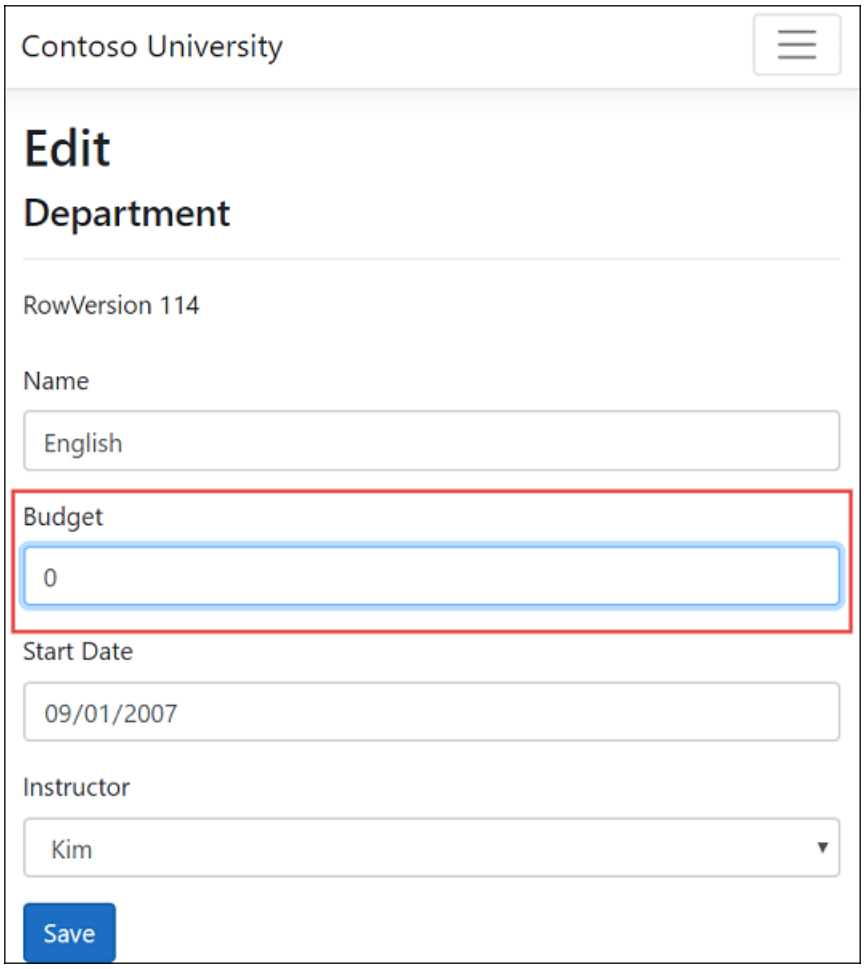
Pessimistic concurrency (locking)

One way to prevent concurrency conflicts is to use database locks. This is called pessimistic concurrency. Before the app reads a database row that it intends to update, it requests a lock. Once a row is locked for update access, no other users are allowed to lock the row until the first lock is released.

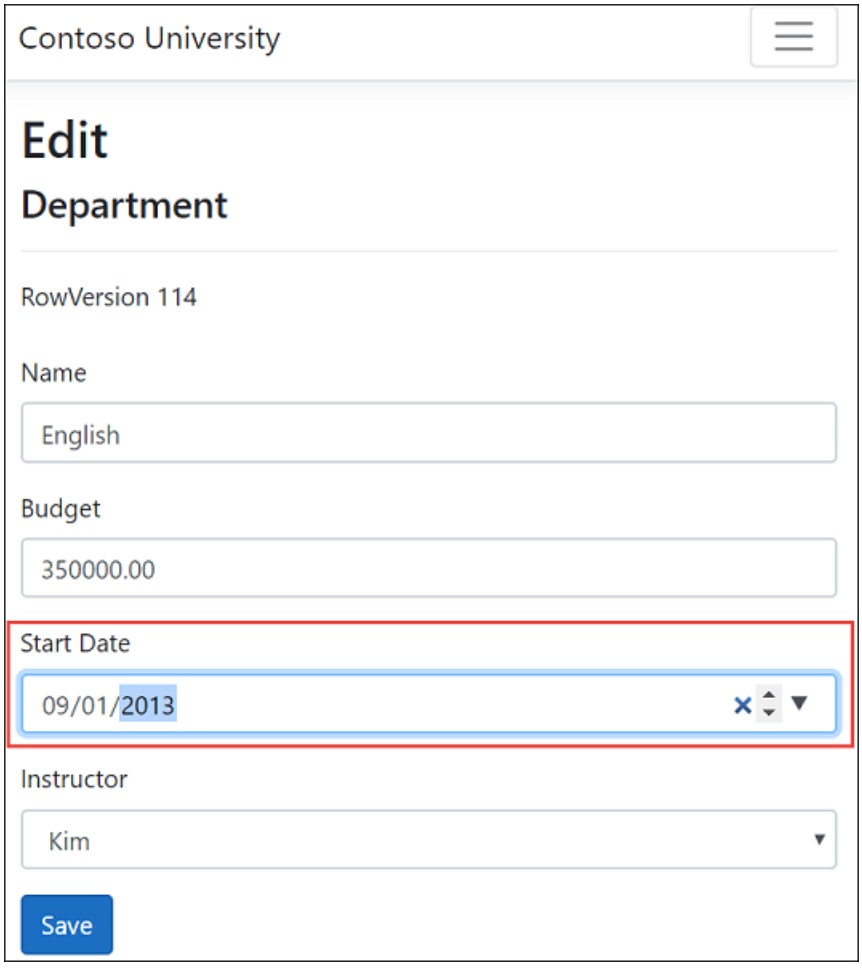
Managing locks has disadvantages. It can be complex to program and can cause performance problems as the number of users increases. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show how to implement it.

Optimistic concurrency

Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change take effect, since the browser displays the Index page with zero as the

Budget amount.

John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

  In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach has some disadvantages:

  - Can't avoid data loss if competing changes are made to the same property.
  - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.
  - Can increase app complexity compared to concurrency detection on an entity.
- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the fetched $350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) If you don't do any coding for concurrency handling, Client Wins happens automatically.
- You can prevent John's change from being updated in the database. Typically, the app would:

  - Display an error message.
  - Show the current state of the data.
  - Allow the user to reapply the changes.

  This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Conflict detection in EF Core

EF Core throws `DbConcurrencyException` exceptions when it detects conflicts. The data model has to be configured to enable conflict detection. Options for enabling conflict detection include the following:

- Configure EF Core to include the original values of columns configured as concurrency tokens in the Where clause of Update and Delete commands.

  When `SaveChanges` is called, the Where clause looks for the original values of any properties annotated with the ConcurrencyCheckAttribute attribute. The update statement won't find a row to update if any of the concurrency token properties changed since the row was first read. EF Core interprets that as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require large amounts of state. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.
- In the database table, include a tracking column that can be used to determine when a row has been changed.

  In a SQL Server database, the data type of the tracking column is `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version number). If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value. In that case, the Update or Delete statement can't find the row to update because of the Where

clause. EF Core throws a concurrency exception when no rows are affected by an Update or Delete command.

Add a tracking property

In *Models/Department.cs*, add a tracking property named RowVersion:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The TimestampAttribute attribute is what identifies the column as a concurrency tracking column. The fluent API is an alternative way to specify the tracking property:

```
modelBuilder.Entity<Department>()
  .Property<byte[]>("RowVersion")
  .IsRowVersion();
```

- Visual Studio
- Visual Studio Code

For a SQL Server database, the `[Timestamp]` attribute on an entity property defined as byte array:

- Causes the column to be included in DELETE and UPDATE WHERE clauses.
- Sets the column type in the database to rowversion.

The database generates a sequential row version number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the fetched row version value. If the row being updated has changed since it was fetched:

- The current row version value doesn't match the fetched value.

- The `Update` or `Delete` commands don't find a row because the `Where` clause looks for the fetched row version value.
- A `DbUpdateConcurrencyException` is thrown.

The following code shows a portion of the T-SQL generated by EF Core when the Department name is updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

The preceding highlighted code shows the `WHERE` clause containing `RowVersion`. If the database `RowVersion` doesn't equal the `RowVersion` parameter (`@p2`), no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT returns the number of rows affected by the last statement. If no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

Update the database

Adding the `RowVersion` property changes the data model, which requires a migration.

Build the project.

- Visual Studio
- Visual Studio Code

- Run the following command in the PMC:

  ```
  Add-Migration RowVersion
  ```

This command:

- Creates the *Migrations/{time stamp}_RowVersion.cs* migration file.
- Updates the *Migrations/SchoolContextModelSnapshot.cs* file. The update adds the following highlighted code to the `BuildModel` method:

```
modelBuilder.Entity("ContosoUniversity.Models.Department", b =>
    {
        b.Property<int>("DepartmentID")
            .ValueGeneratedOnAdd()
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

        b.Property<decimal>("Budget")
            .HasColumnType("money");

        b.Property<int?>("InstructorID");

        b.Property<string>("Name")
            .HasMaxLength(50);

        b.Property<byte[]>("RowVersion")
            .IsConcurrencyToken()
            .ValueGeneratedOnAddOrUpdate();

        b.Property<DateTime>("StartDate");

        b.HasKey("DepartmentID");

        b.HasIndex("InstructorID");

        b.ToTable("Department");
    });
```

- Visual Studio
- Visual Studio Code
- Run the following command in the PMC:

```
Update-Database
```

Scaffold Department pages

- Visual Studio
- Visual Studio Code
- Follow the instructions in Scaffold Student pages with the following exceptions:
- Create a *Pages/Departments* folder.
- Use `Department` for the model class.
  - Use the existing context class instead of creating a new one.

Build the project.

Update the Index page

The scaffolding tool created a `RowVersion` column for the Index page, but that field wouldn't be displayed in a production app. In this tutorial, the last byte of the `RowVersion` is displayed to help show how concurrency handling works. The last byte isn't guaranteed to be unique by itself.

Update *Pages\Departments\Index.cshtml* page:

- Replace Index with Departments.

- Change the code containing `RowVersion` to show just the last byte of the byte array.
- Replace FirstMidName with FullName.

The following code shows the updated page:

```
@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Name)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Budget)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].StartDate)
                </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Administrator)
            </th>
            <th>
                RowVersion
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Department)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    @item.RowVersion[7]
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the Edit page model

Update *Pages\Departments\Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (departmentToUpdate == null)
            {
                return HandleDeletedDepartment();
            }

            _context.Entry(departmentToUpdate)
                .Property("RowVersion").OriginalValue = Department.RowVersion;

            if (
```

```
            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
            {
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues = (Department)exceptionEntry.Entity;
                    var databaseEntry = exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable to save. " +
                            "The department was deleted by another user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await setDbErrorMessage(dbValues, clientValues, _context);

                    // Save the current RowVersion so next postback
                    // matches unless an new concurrency issue happens.
                    Department.RowVersion = (byte[])dbValues.RowVersion;
                    // Clear the model error for the next postback.
                    ModelState.Remove("Department.RowVersion");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

        private IActionResult HandleDeletedDepartment()
        {
            var deletedDepartment = new Department();
            // ModelState contains the posted data because of the deletion error
            // and will overide the Department instance values when displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another user.");
            InstructorNameSL = new SelectList(_context.Instructors, "ID", "FullName", Department.InstructorID);
            return Page();
        }

        private async Task setDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
            }
```

```
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in the database "
              + "have been displayed. If you still want to edit this record, click "
              + "the Save button again.");
        }
    }
}
```

The OriginalValue is updated with the `rowVersion` value from the entity when it was fetched in the `OnGet` method. EF Core generates a SQL UPDATE command with a WHERE clause containing the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), a `DbUpdateConcurrencyException` exception is thrown.

```
public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        return HandleDeletedDepartment();
    }

    _context.Entry(departmentToUpdate)
        .Property("RowVersion").OriginalValue = Department.RowVersion;
```

In the preceding highlighted code:

- The value in `Department.RowVersion` is what was in the entity when it was originally fetched in the Get request for the Edit page. The value is provided to the `OnPost` method by a hidden field in the Razor page that displays the entity to be edited. The hidden field value is copied to `Department.RowVersion` by the model binder.
- `OriginalValue` is what EF Core will use in the Where clause. Before the highlighted line of code executes, `OriginalValue` has the value that was in the database when `FirstOrDefaultAsync` was called in this method, which might be different from what was displayed on the Edit page.
- The highlighted code makes sure that EF Core uses the original `RowVersion` value from the displayed `Department` entity in the SQL UPDATE statement's Where clause.

When a concurrency error happens, the following highlighted code gets the client values (the values posted to this method) and the database values.

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await setDbErrorMessage(dbValues, clientValues, _context);

        // Save the current RowVersion so next postback
        // matches unless an new concurrency issue happens.
        Department.RowVersion = (byte[])dbValues.RowVersion;
        // Clear the model error for the next postback.
        ModelState.Remove("Department.RowVersion");
    }
```

The following code adds a custom error message for each column that has database values different from what was posted to `OnPostAsync`:

```
private async Task setDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database "
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

The following highlighted code sets the `RowVersion` value to the new value retrieved from the database. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await setDbErrorMessage(dbValues, clientValues, _context);

        // Save the current RowVersion so next postback
        // matches unless an new concurrency issue happens.
        Department.RowVersion = (byte[])dbValues.RowVersion;
        // Clear the model error for the next postback.
        ModelState.Remove("Department.RowVersion");
    }
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Edit page

Update *Pages/Departments/Edit.cshtml* with the following code:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.RowVersion" />
            <div class="form-group">
                <label>RowVersion</label>
                @Model.Department.RowVersion[7]
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label"></label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label"></label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label"></label>
                <input asp-for="Department.StartDate" class="form-control" />
                <span asp-validation-for="Department.StartDate" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
                <select asp-for="Department.InstructorID" class="form-control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `RowVersion` must be added so postback binds the value.
- Displays the last byte of `RowVersion` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

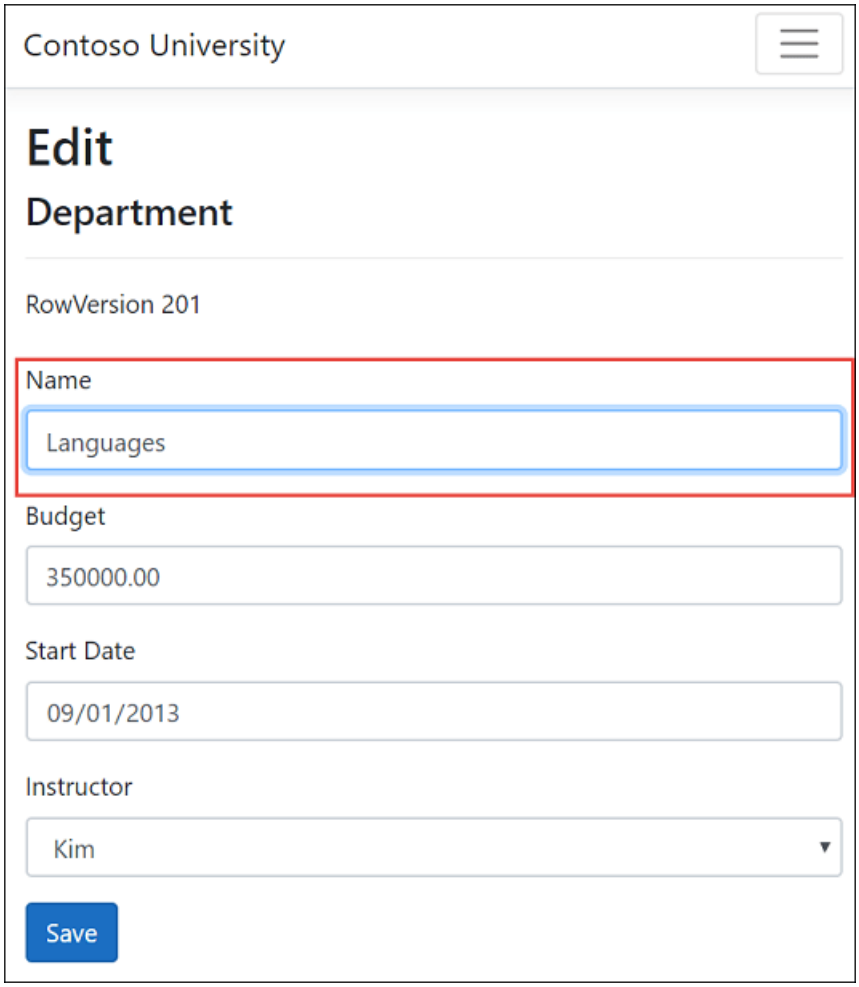
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

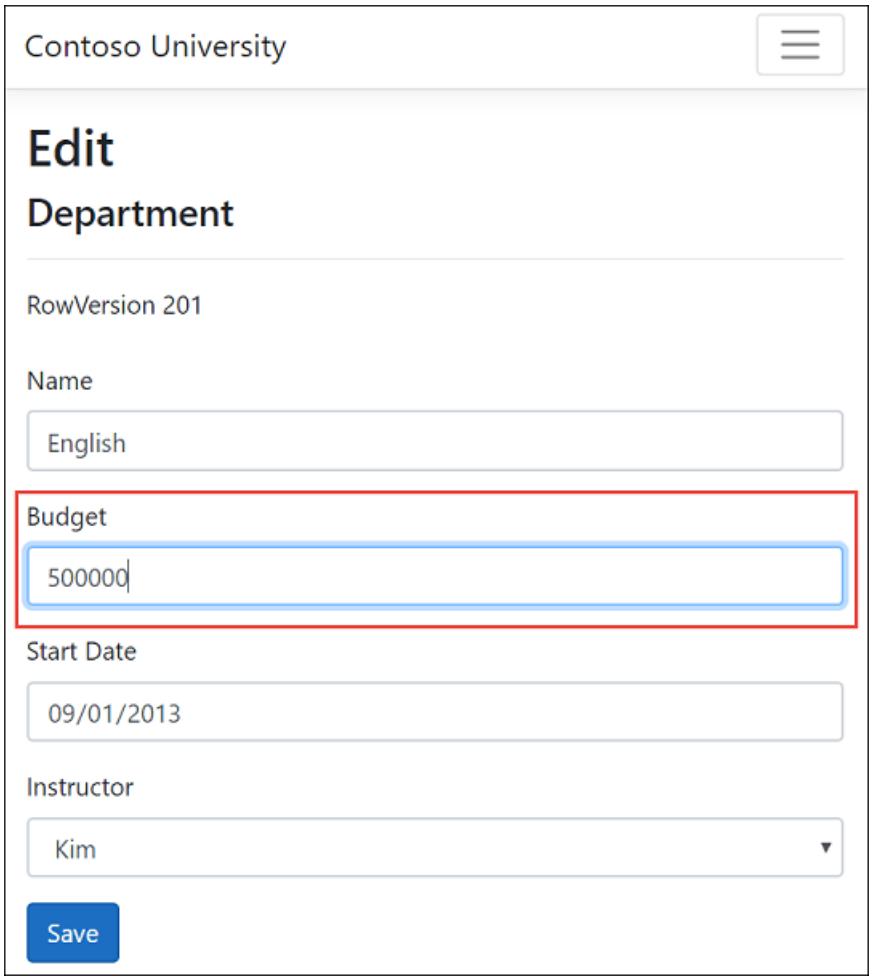
The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.

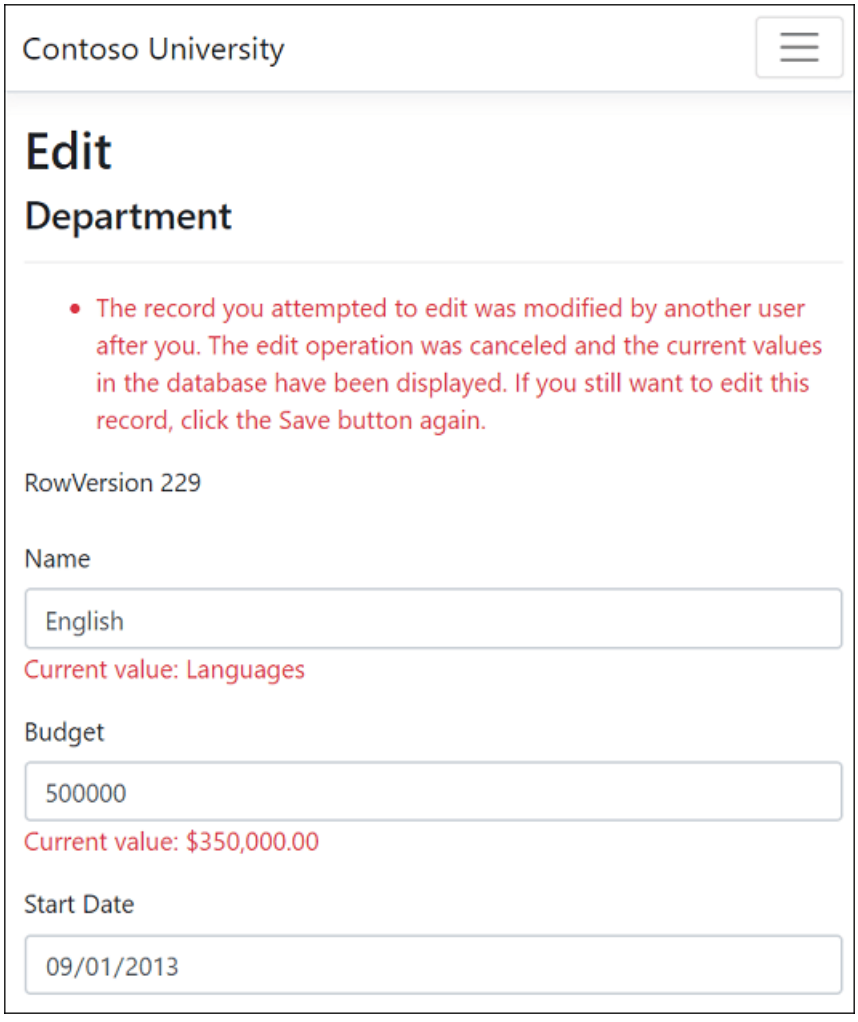


The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

Change a different field in the second browser tab.



Click **Save**. You see error messages for all fields that don't match the database values:



This browser window didn't intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.

Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page model

Update *Pages/Departments/Delete.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool? concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                 return NotFound();
            }

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to delete "
                  + "was modified by another user after you selected delete. "
                  + "The delete operation was canceled and the current values in the "
                  + "database have been displayed. If you still want to delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.rowVersion value is from when the entity
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
```

```
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched. `Department.RowVersion` is the row version when the entity was fetched. When EF Core creates the SQL DELETE command, it includes a WHERE clause with `RowVersion`. If the SQL DELETE command results in zero rows affected:

- The `RowVersion` in the SQL DELETE command doesn't match `RowVersion` in the database.
- A DbUpdateConcurrencyException exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete page

Update *Pages/Departments/Delete.cshtml* with the following code:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.RowVersion)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.RowVersion[7])
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-danger" /> |
            <a asp-page="./Index">Back to List</a>
        </div>
</form>
</div>
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Changes `RowVersion` to display the last byte.

- Adds a hidden row version. `RowVersion` must be added so postback binds the value.

Test concurrency conflicts

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the database. Clicking **Delete** deletes the entity, unless `RowVersion` has been updated.

Additional resources

- Concurrency Tokens in EF Core
- Handle concurrency in EF Core
- Debugging ASP.NET Core 2.x source

Next steps

This is the last tutorial in the series. Additional topics are covered in the MVC version of this tutorial series.

PREVIOUS TUTORIAL

This tutorial shows how to handle conflicts when multiple users update an entity concurrently (at the same time). If you run into problems you can't solve, download or view the completed app. Download instructions.

Concurrency conflicts

A concurrency conflict occurs when:

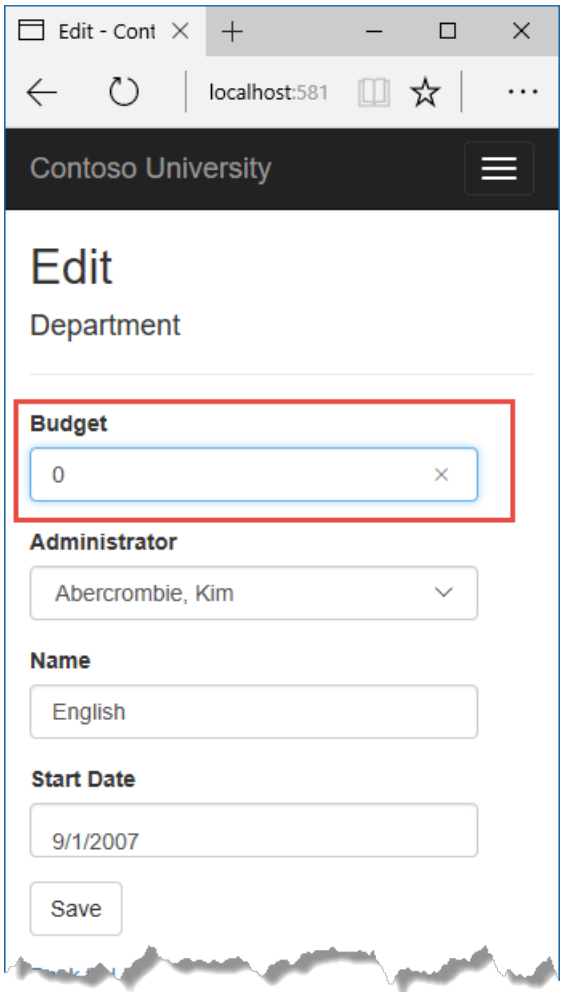
- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the DB.

If concurrency detection isn't enabled, when concurrent updates occur:

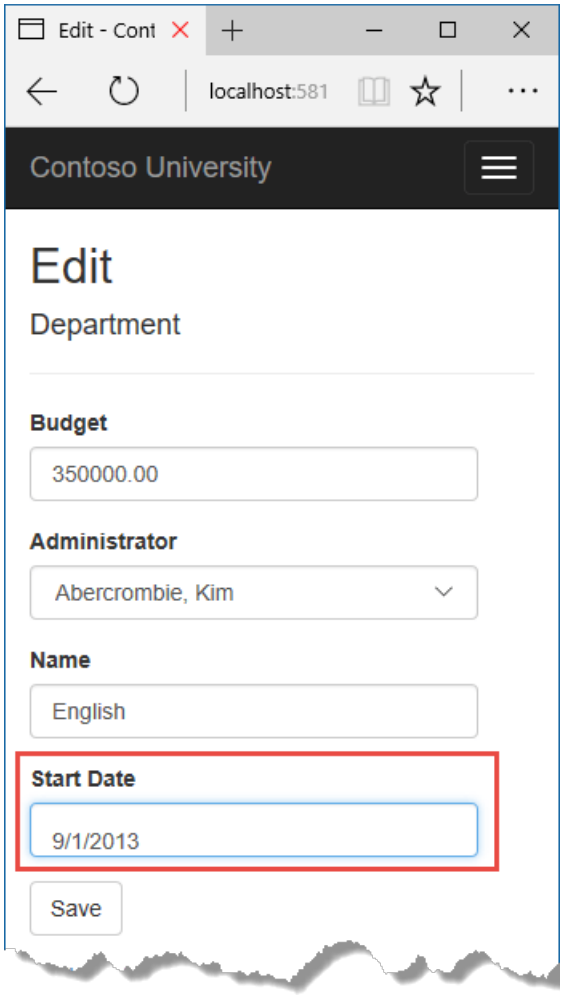
- The last update wins. That is, the last update values are saved to the DB.
- The first of the current updates are lost.

Optimistic concurrency

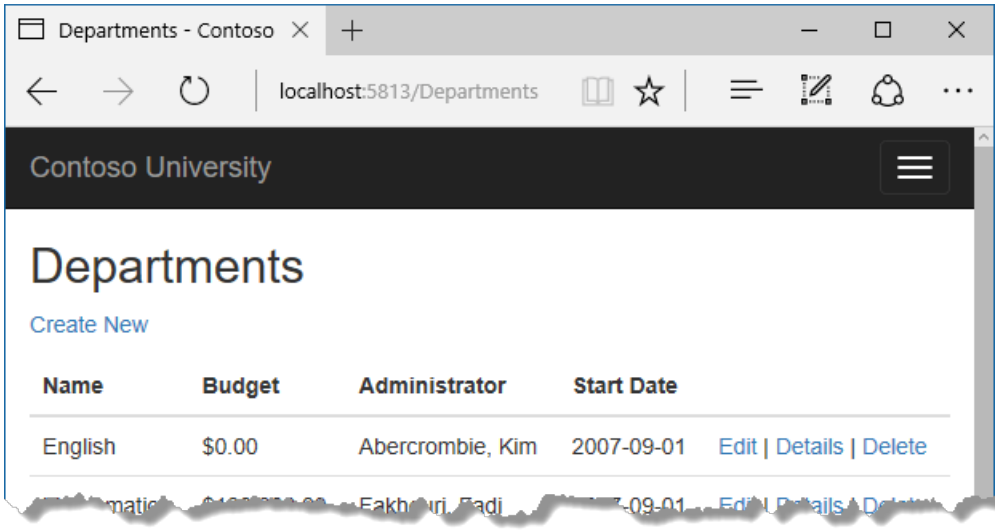
Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser displays the Index page.



John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Optimistic concurrency includes the following options:

- You can keep track of which property a user has modified and update only the corresponding columns in the DB.

  In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach:

  - Can't avoid data loss if competing changes are made to the same property.
  - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.
  - Can increase app complexity compared to concurrency detection on an entity.
- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the fetched $350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) If you don't do any coding for concurrency handling, Client Wins happens automatically.
- You can prevent John's change from being updated in the DB. Typically, the app would:

  - Display an error message.
  - Show the current state of the data.
  - Allow the user to reapply the changes.

  This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Handling concurrency

When a property is configured as a concurrency token:

- EF Core verifies that property has not been modified after it was fetched. The check occurs when SaveChanges or SaveChangesAsync is called.
- If the property has been changed after it was fetched, a DbUpdateConcurrencyException is thrown.

The DB and data model must be configured to support throwing `DbUpdateConcurrencyException`.

Detecting concurrency conflicts on a property

Concurrency conflicts can be detected at the property level with the ConcurrencyCheck attribute. The attribute can be applied to multiple properties on the model. For more information, see Data Annotations-ConcurrencyCheck.

The `[ConcurrencyCheck]` attribute isn't used in this tutorial.

Detecting concurrency conflicts on a row

To detect concurrency conflicts, a rowversion tracking column is added to the model. `rowversion`:

- Is SQL Server specific. Other databases may not provide a similar feature.
- Is used to determine that an entity has not been changed since it was fetched from the DB.

The DB generates a sequential `rowversion` number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the fetched value of `rowversion`. If the row being updated has changed:

- `rowversion` doesn't match the fetched value.
- The `Update` or `Delete` commands don't find a row because the `Where` clause includes the fetched `rowversion`.
- A `DbUpdateConcurrencyException` is thrown.

In EF Core, when no rows have been updated by an `Update` or `Delete` command, a concurrency exception is thrown.

Add a tracking property to the Department entity

In *Models/Department.cs*, add a tracking property named RowVersion:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Timestamp attribute specifies that this column is included in the `Where` clause of `Update` and `Delete` commands. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` type replaced it.

The fluent API can also specify the tracking property:

```
modelBuilder.Entity<Department>()
  .Property<byte[]>("RowVersion")
  .IsRowVersion();
```

The following code shows a portion of the T-SQL generated by EF Core when the Department name is updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

The preceding highlighted code shows the `WHERE` clause containing `RowVersion`. If the DB `RowVersion` doesn't equal the `RowVersion` parameter (`@p2`), no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT returns the number of rows affected by the last statement. In no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

You can see the T-SQL EF Core generates in the output window of Visual Studio.

Update the DB

Adding the `RowVersion` property changes the DB model, which requires a migration.

Build the project. Enter the following in a command window:

```
dotnet ef migrations add RowVersion
dotnet ef database update
```

The preceding commands:

- Adds the *Migrations/{time stamp}_RowVersion.cs* migration file.
- Updates the *Migrations/SchoolContextModelSnapshot.cs* file. The update adds the following highlighted code to the `BuildModel` method:

  ```
  ```

- Runs migrations to update the DB.

Scaffold the Departments model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Department` for the model class.

The preceding command scaffolds the `Department` model. Open the project in Visual Studio.

Build the project.

Update the Departments Index page

The scaffolding engine created a `RowVersion` column for the Index page, but that field shouldn't be displayed. In this tutorial, the last byte of the `RowVersion` is displayed to help understand concurrency. The last byte isn't guaranteed to be unique. A real app wouldn't display `RowVersion` or the last byte of `RowVersion`.

Update the Index page:

- Replace Index with Departments.
- Replace the markup containing `RowVersion` with the last byte of `RowVersion`.
- Replace FirstMidName with FullName.

The following markup shows the updated page:

```
@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Name)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Budget)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].StartDate)
                </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Administrator)
            </th>
            <th>
                RowVersion
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Department) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Budget)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.StartDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Administrator.FullName)
            </td>
            <td>
                @item.RowVersion[7]
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.DepartmentID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.DepartmentID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Update the Edit page model

Update *Pages\Departments\Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Data;
```

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            // null means Department was deleted by another user.
            if (departmentToUpdate == null)
            {
                return HandleDeletedDepartment();
            }

            // Update the RowVersion to the value when this entity was
            // fetched. If the entity has been updated after it was
            // fetched, RowVersion won't match the DB RowVersion and
            // a DbUpdateConcurrencyException is thrown.
            // A second postback will make them match, unless a new
            // concurrency issue happens.
            _context.Entry(departmentToUpdate)
                .Property("RowVersion").OriginalValue = Department.RowVersion;
```

```
            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
            {
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues = (Department)exceptionEntry.Entity;
                    var databaseEntry = exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable to save. " +
                            "The department was deleted by another user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await setDbErrorMessage(dbValues, clientValues, _context);

                    // Save the current RowVersion so next postback
                    // matches unless an new concurrency issue happens.
                    Department.RowVersion = (byte[])dbValues.RowVersion;
                    // Must clear the model error for the next postback.
                    ModelState.Remove("Department.RowVersion");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

        private IActionResult HandleDeletedDepartment()
        {
            var deletedDepartment = new Department();
            // ModelState contains the posted data because of the deletion error and will overide the
Department instance values when displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another user.");
            InstructorNameSL = new SelectList(_context.Instructors, "ID", "FullName", Department.InstructorID);
            return Page();
        }

        private async Task setDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
            }
```

```
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in the database "
              + "have been displayed. If you still want to edit this record, click "
              + "the Save button again.");
        }
    }
}
```

To detect a concurrency issue, the OriginalValue is updated with the `rowVersion` value from the entity it was fetched. EF Core generates a SQL UPDATE command with a WHERE clause containing the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), a `DbUpdateConcurrencyException` exception is thrown.

```
public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    // null means Department was deleted by another user.
    if (departmentToUpdate == null)
    {
        return HandleDeletedDepartment();
    }

    // Update the RowVersion to the value when this entity was
    // fetched. If the entity has been updated after it was
    // fetched, RowVersion won't match the DB RowVersion and
    // a DbUpdateConcurrencyException is thrown.
    // A second postback will make them match, unless a new
    // concurrency issue happens.
    _context.Entry(departmentToUpdate)
        .Property("RowVersion").OriginalValue = Department.RowVersion;
```

In the preceding code, `Department.RowVersion` is the value when the entity was fetched. `OriginalValue` is the value in the DB when `FirstOrDefaultAsync` was called in this method.

The following code gets the client values (the values posted to this method) and the DB values:

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless an new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
```

The following code adds a custom error message for each column that has DB values different from what was posted to `OnPostAsync`:

```
private async Task setDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database "
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

The following highlighted code sets the `RowVersion` value to the new value retrieved from the DB. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless an new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Edit page

Update *Pages/Departments/Edit.cshtml* with the following markup:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.RowVersion" />
            <div class="form-group">
                <label>RowVersion</label>
                @Model.Department.RowVersion[7]
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label"></label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label"></label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label"></label>
                <input asp-for="Department.StartDate" class="form-control" />
                <span asp-validation-for="Department.StartDate" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
                <select asp-for="Department.InstructorID" class="form-control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `RowVersion` must be added so post back binds the value.
- Displays the last byte of `RowVersion` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

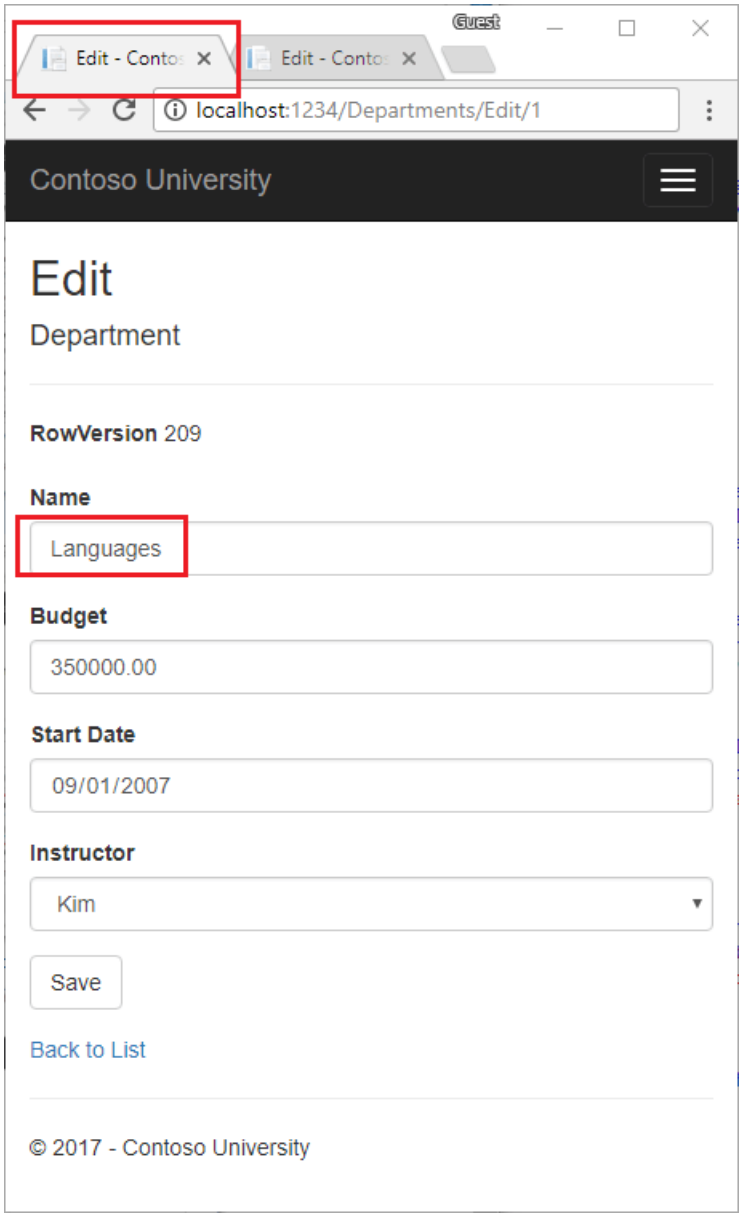
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

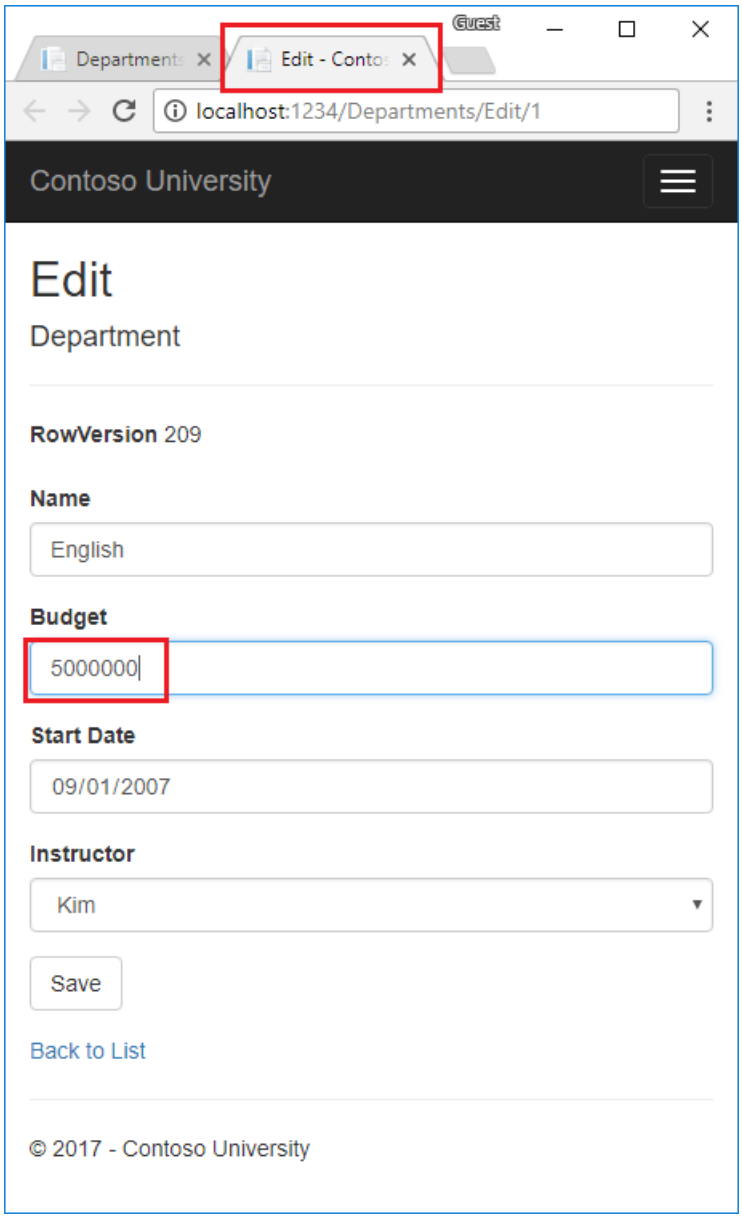
The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.

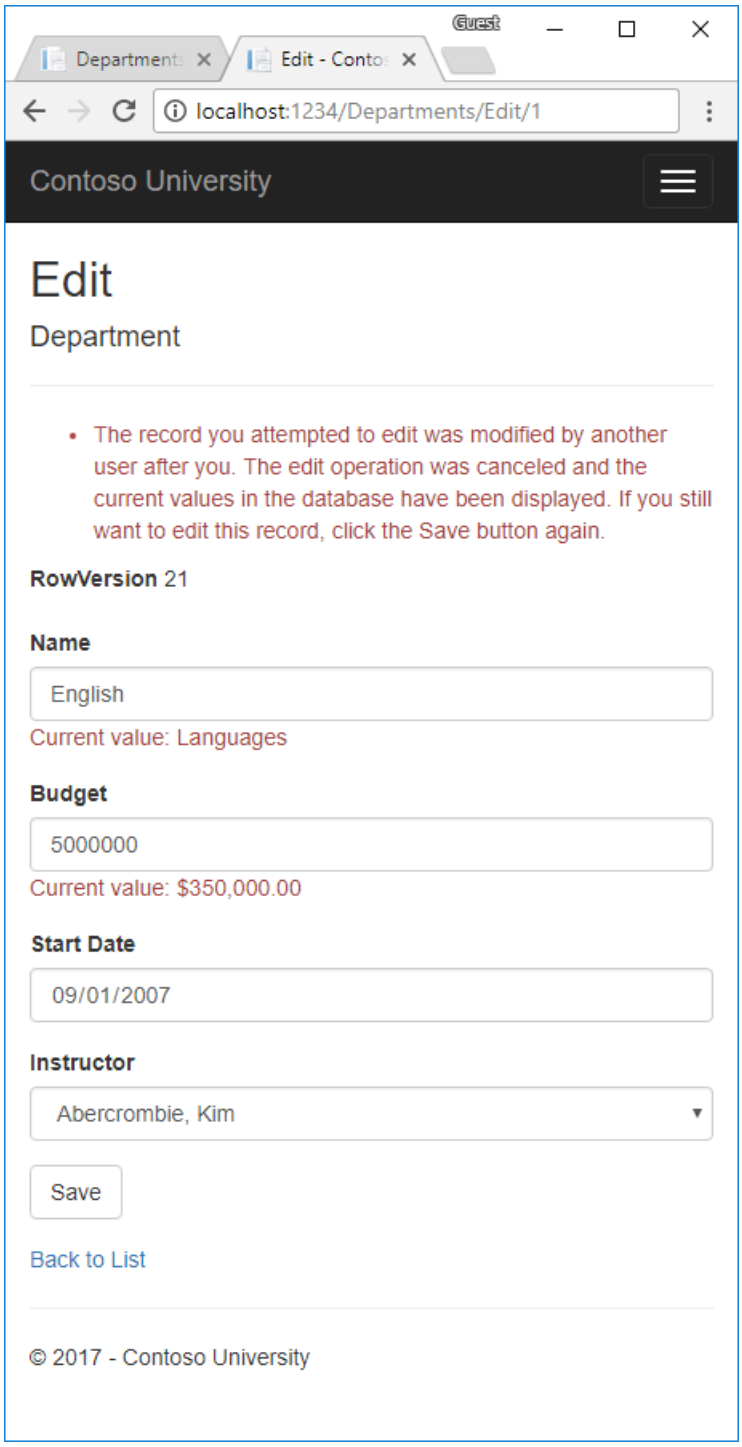


The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

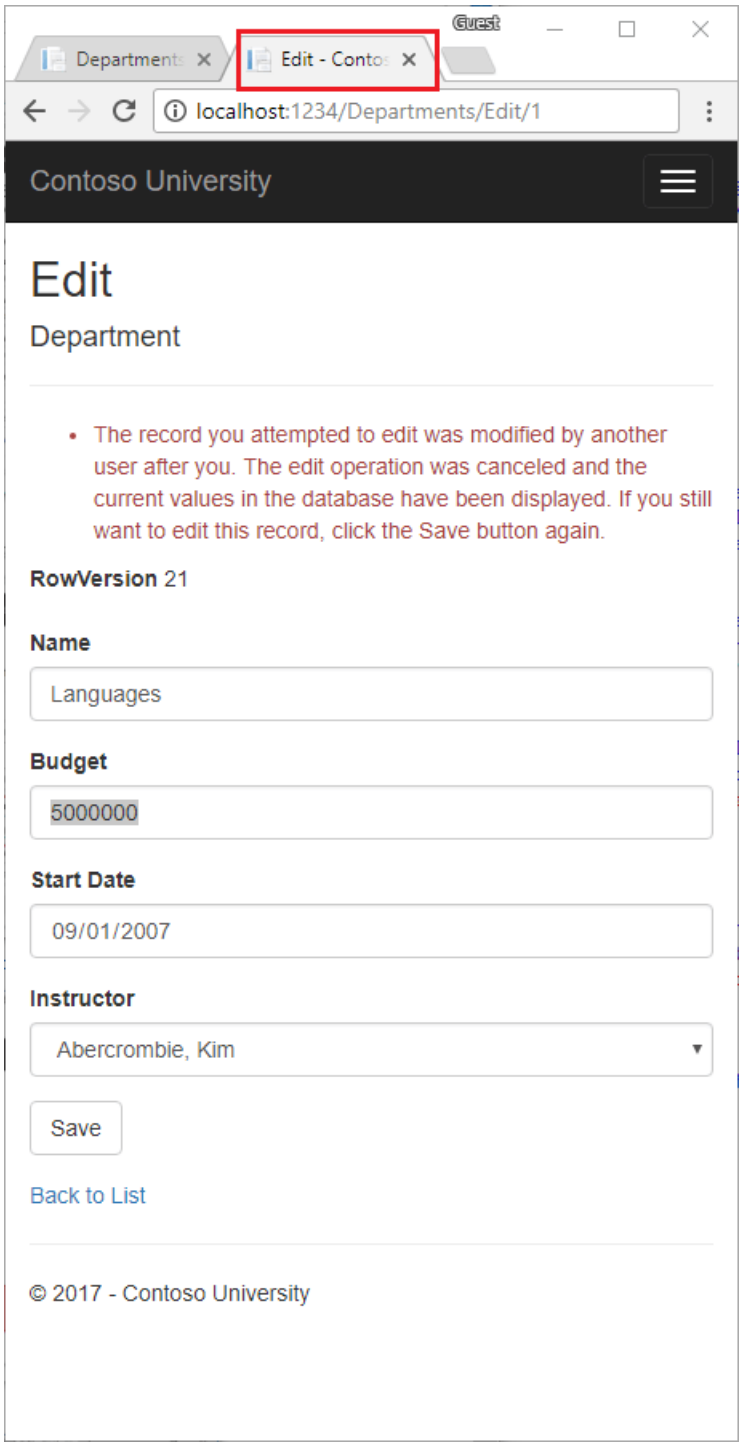
Change a different field in the second browser tab.



Click **Save**. You see error messages for all fields that don't match the DB values:



This browser window didn't intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page

Update the Delete page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
```

```
        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool? concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                 return NotFound();
            }

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to delete "
                  + "was modified by another user after you selected delete. "
                  + "The delete operation was canceled and the current values in the "
                  + "database have been displayed. If you still want to delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.rowVersion value is from when the entity
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched. `Department.RowVersion` is the row version when the entity was fetched. When EF Core creates the SQL DELETE command, it includes a WHERE clause with `RowVersion`. If the SQL DELETE command results in zero rows affected:

- The `RowVersion` in the SQL DELETE command doesn't match `RowVersion` in the DB.
- A DbUpdateConcurrencyException exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete page

Update *Pages/Departments/Delete.cshtml* with the following code:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.RowVersion)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.RowVersion[7])
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-page="./Index">Back to List</a>
        </div>
</form>
</div>
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.
- Replaces FirstMidName with FullName in the **Administrator** field.

- Changes `RowVersion` to display the last byte.
- Adds a hidden row version. `RowVersion` must be added so post back binds the value.

Test concurrency conflicts with the Delete page

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the DB. Clicking **Delete** deletes the entity, unless `RowVersion` has been updated.

See Inheritance on how to inherit a data model.

Additional resources

- Concurrency Tokens in EF Core
- Handle concurrency in EF Core
- YouTube version of this tutorial(Handling Concurrency Conflicts)
- YouTube version of this tutorial(Part 2)
- YouTube version of this tutorial(Part 3)

PREVIOUS

ASP.NET Core MVC with EF Core - tutorial series

9/22/2020 • 2 minutes to read • Edit Online

This tutorial has **not** been updated to ASP.NET Core 3.0. The Razor Pages version has been updated. For information on when this might be updated, see this GitHub issue.

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model that was introduced in ASP.NET Core 2.0. For new development, we recommend Razor Pages over MVC with controllers and views. There is a Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- A version available for ASP.NET Core 3.0

1. Get started
2. Create, Read, Update, and Delete operations
3. Sorting, filtering, paging, and grouping
4. Migrations
5. Create a complex data model
6. Reading related data
7. Updating related data
8. Handle concurrency conflicts
9. Inheritance
10. Advanced topics

Tutorial: Get started with EF Core in an ASP.NET MVC web app

9/22/2020 • 21 minutes to read • Edit Online

This tutorial has **not** been updated to ASP.NET Core 3.0. The Razor Pages version has been updated. Most of the code changes for the ASP.NET Core 3.0 and later version of this tutorial:

- Are in the *Startup.cs* and *Program.cs* files.
- Can be found in the Razor Pages version.

For information on when this might be updated, see this GitHub issue.

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model that was introduced in ASP.NET Core 2.0. For new development, we recommend Razor Pages over MVC with controllers and views. There is a Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- A version available for ASP.NET Core 3.0

The Contoso University sample web application demonstrates how to create ASP.NET Core 2.2 MVC web applications using Entity Framework (EF) Core 2.2 and Visual Studio 2017 or 2019.

The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This is the first in a series of tutorials that explain how to build the Contoso University sample application from scratch.

In this tutorial, you:

- Create an ASP.NET Core MVC web app
- Set up the site style
- Learn about EF Core NuGet packages
- Create the data model
- Create the database context
- Register the context for dependency injection
- Initialize the database with test data
- Create a controller and views
- View the database

Prerequisites

- .NET Core SDK 2.2
- Visual Studio 2019 with the following workloads:

- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload

Troubleshooting

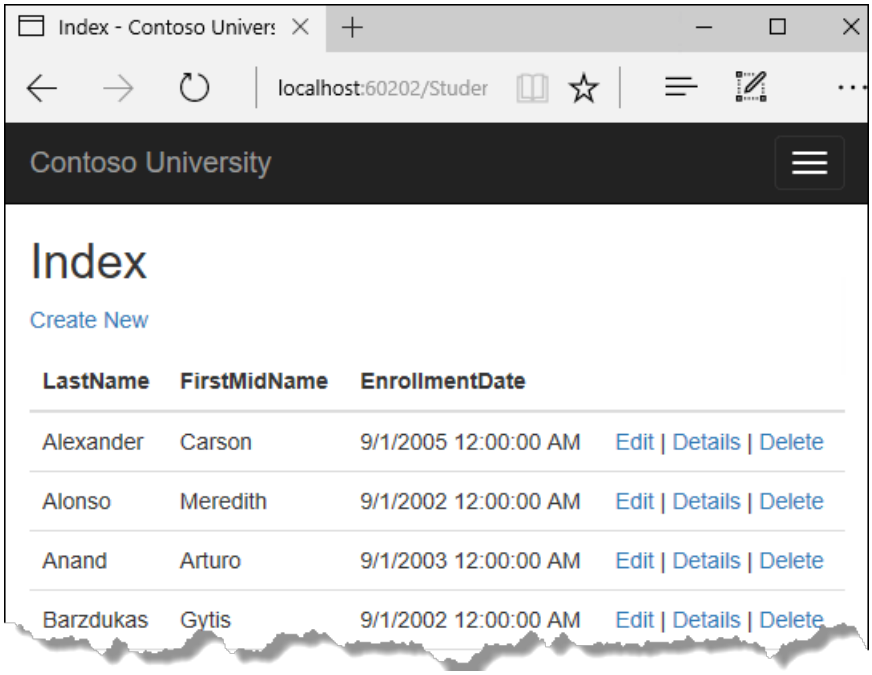
If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the completed project. For a list of common errors and how to solve them, see the Troubleshooting section of the last tutorial in the series. If you don't find what you need there, you can post a question to StackOverflow.com for ASP.NET Core or EF Core.

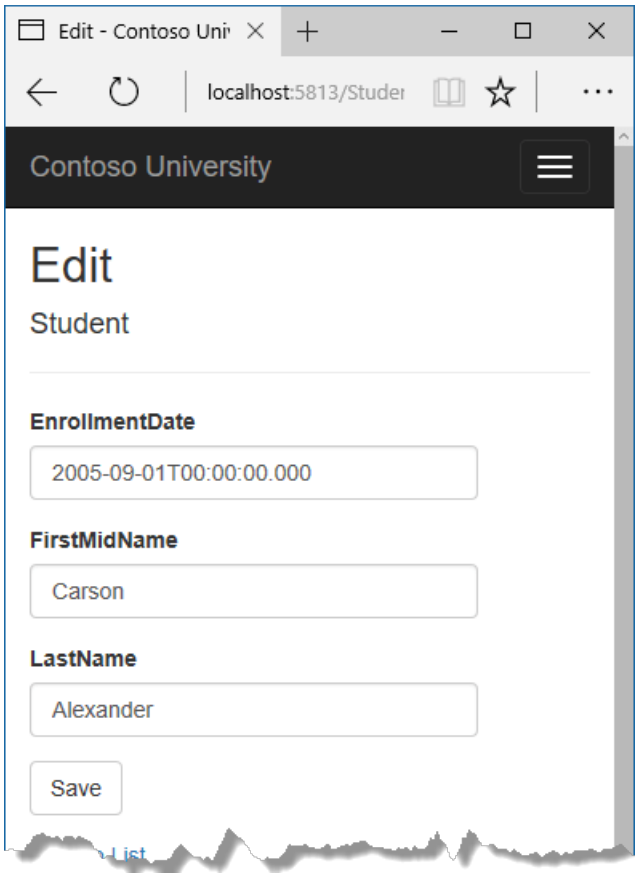
> **TIP**
> This is a series of 10 tutorials, each of which builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. Then if you run into problems, you can start over from the previous tutorial instead of going back to the beginning of the whole series.

Contoso University web app

The application you'll be building in these tutorials is a simple university web site.

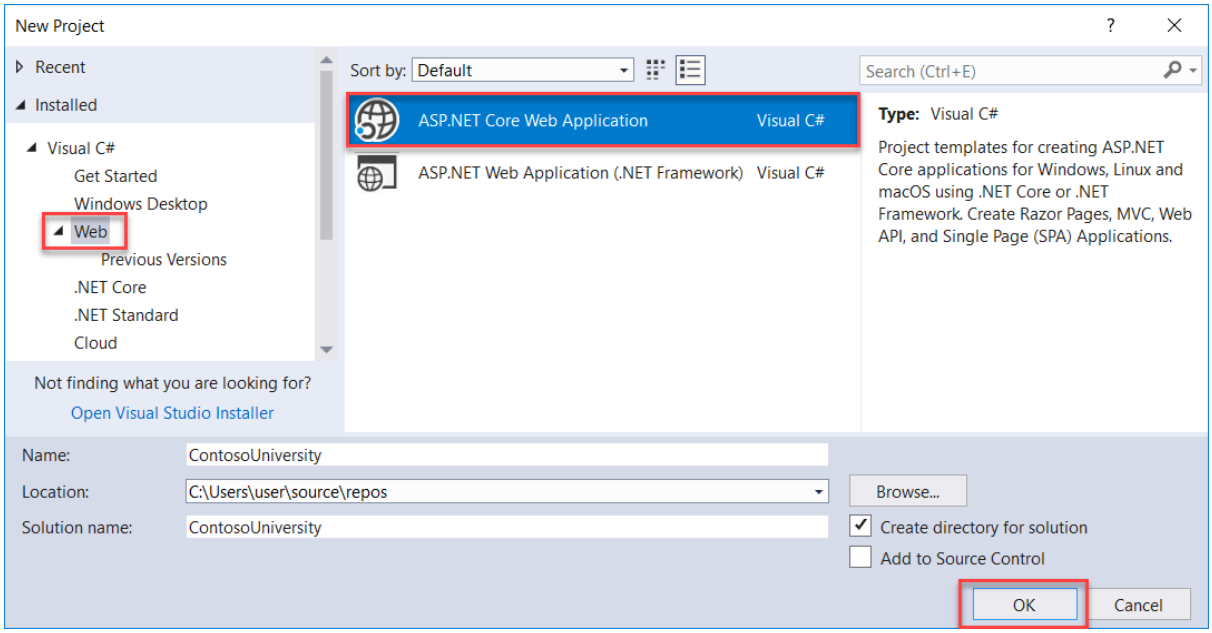
Users can view and update student, course, and instructor information. Here are a few of the screens you'll create.





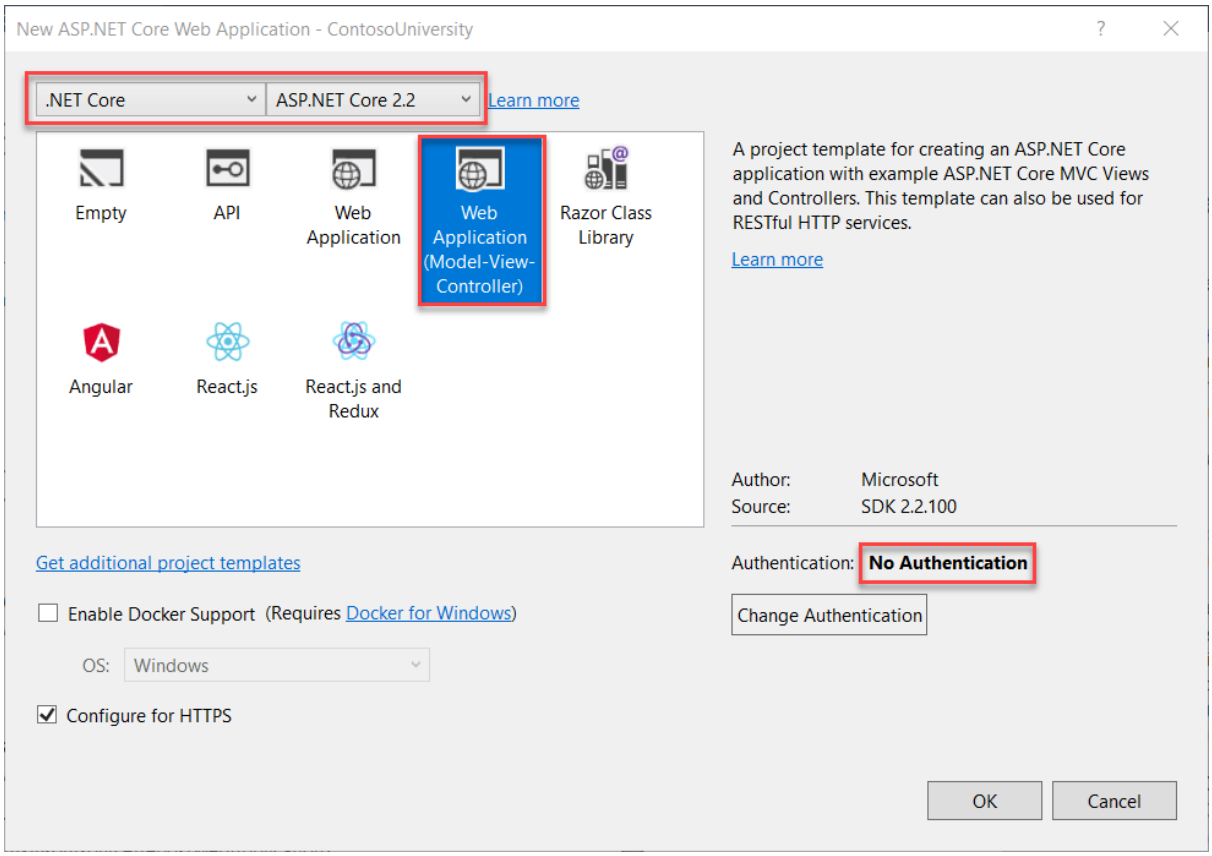
Create web app

- Open Visual Studio.
- From the **File** menu, select **New** > **Project**.
- From the left pane, select **Installed** > **Visual C#** > **Web**.
- Select the **ASP.NET Core Web Application** project template.
- Enter **ContosoUniversity** as the name and click **OK**.



- Wait for the **New ASP.NET Core Web Application** dialog to appear.
- Select **.NET Core**, **ASP.NET Core 2.2** and the **Web Application (Model-View-Controller)** template.

- Make sure **Authentication** is set to **No Authentication**.
- Select **OK**



Set up the site style

A few simple changes will set up the site menu, layout, and home page.

Open *Views/Shared/_Layout.cshtml* and make the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Add menu entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Privacy** menu entry.

The changes are highlighted.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute"
              crossorigin="anonymous"
              integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE="/>
    </environment>
    <link rel="stylesheet" href="~/css/site.css" />
```

```
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow
mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">Contoso
University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Students" asp-
action="Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Courses" asp-
action="Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Instructors" asp-
action="Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Departments" asp-
action="Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Contoso University - <a asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
        </div>
    </footer>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">
        </script>
```

```
        <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/js/bootstrap.bundle.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha256-E/V4cWE4qvAeO5MOhjtGtqDzPndRO1LBk8lJ/PR7CA4=">
        </script>
    </environment>
    <script src="~/js/site.js" asp-append-version="true"></script>

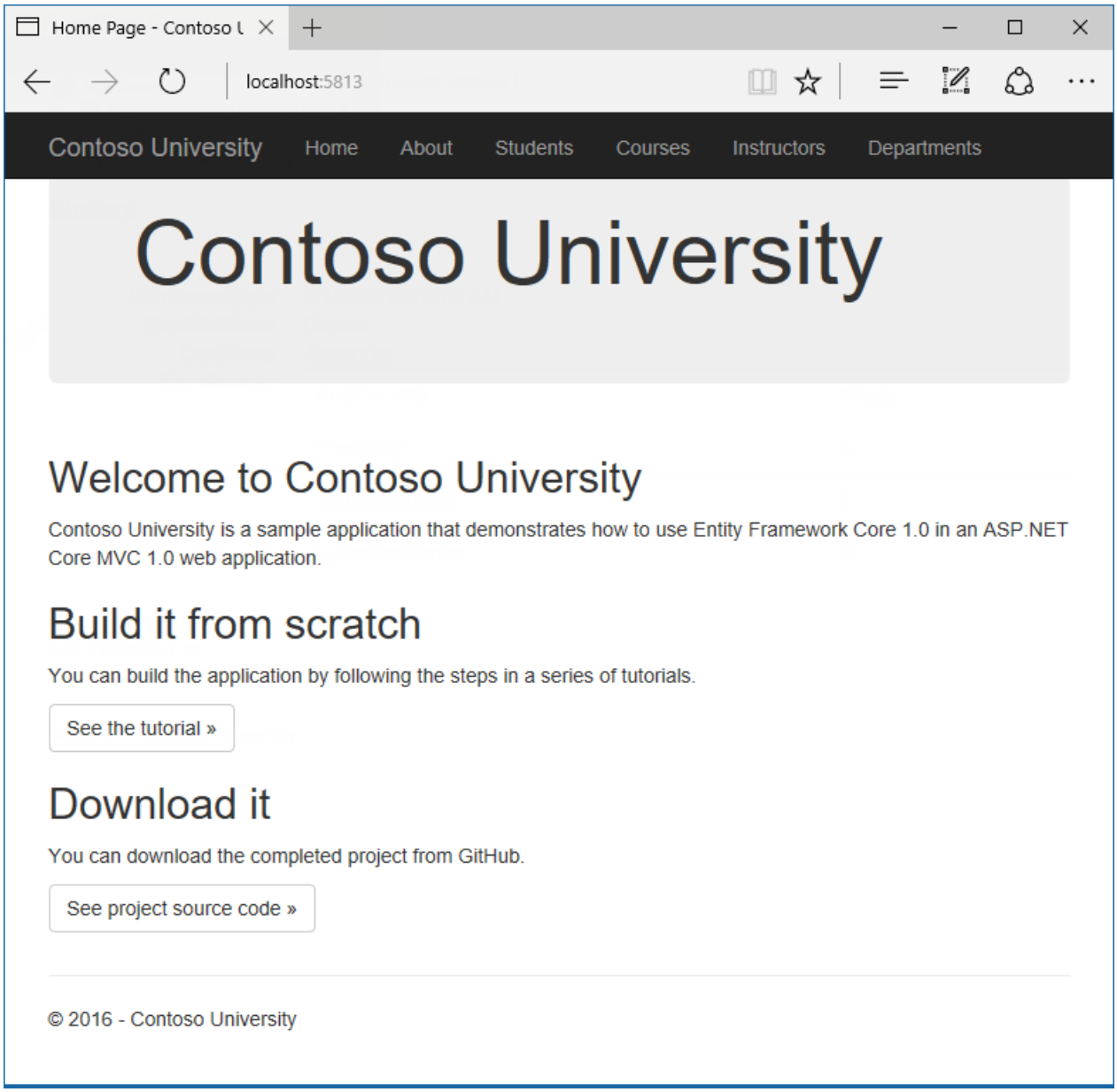
    @RenderSection("Scripts", required: false)
</body>
</html>
```

In *Views/Home/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this application:

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default" href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-mvc/intro/samples/cu-final">See
project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu. You see the home page with tabs for the pages you'll create in these tutorials.



About EF Core NuGet packages

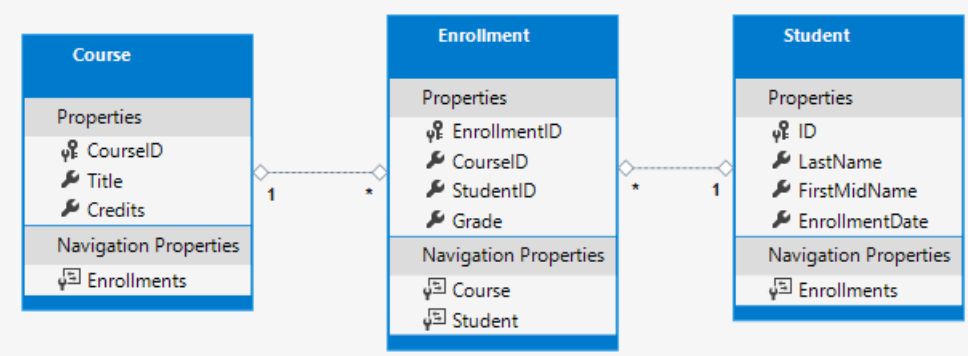
To add EF Core support to a project, install the database provider that you want to target. This tutorial uses SQL Server, and the provider package is Microsoft.EntityFrameworkCore.SqlServer. This package is included in the Microsoft.AspNetCore.App metapackage, so you don't need to reference the package.

The EF SQL Server package and its dependencies (`Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`) provide runtime support for EF. You'll add a tooling package later, in the Migrations tutorial.

For information about other database providers that are available for Entity Framework Core, see Database providers.

Create the data model

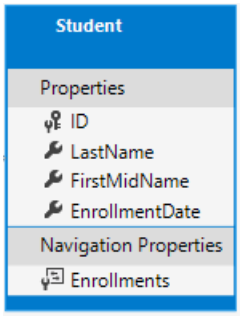
Next you'll create entity classes for the Contoso University application. You'll start with the following three entities.



There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections you'll create a class for each one of these entities.

The Student entity



In the *Models* folder, create a class file named *Student.cs* and replace the template code with the following code.

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

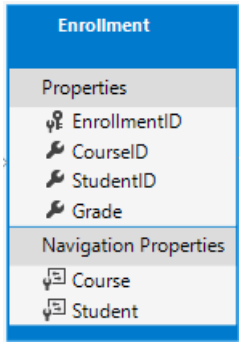
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or `classnameID` as the primary key.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student entity` will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given Student row in the database has two related Enrollment rows (rows that contain that student's primary key value in their StudentID foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection<T>`. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The Enrollment entity



In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property will be the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself as you saw in the `Student` entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a later tutorial, you'll see how using ID without classname makes it easier to implement inheritance in the data model.

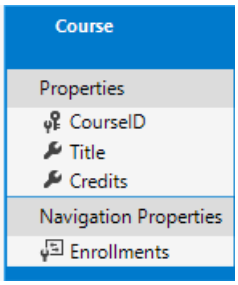
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a foreign key property if it's named `<navigation property name><primary key property name>` (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply `<primary key property name>` (for example, `CourseID` since the `Course` entity's primary key is `CourseID`).

The Course entity



In the *Models* folder, create *Course.cs* and replace the existing code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the `DatabaseGenerated` attribute in a later tutorial in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the database context

The main class that coordinates Entity Framework functionality for a given data model is the database context class. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. In your code you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named *Data*.

In the *Data* folder create a new class file named *SchoolContext.cs*, and replace the template code with the following code:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

This code creates a `DbSet` property for each entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.

You could've omitted the `DbSet<Enrollment>` and `DbSet<Course>` statements and it would work the same. The Entity Framework would include them implicitly because the `Student` entity references the `Enrollment` entity and the `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural (Students rather than Student), but developers disagree about whether table names should be pluralized or not. For these tutorials you'll override the default behavior by specifying singular table names in the DbContext. To do that, add the following highlighted code after the last DbSet property.

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Register the SchoolContext

ASP.NET Core implements dependency injection by default. Services (such as the EF database context) are registered with dependency injection during application startup. Components that require these services (such as MVC controllers) are provided these services via constructor parameters. You'll see the controller constructor code that gets a context instance later in this tutorial.

To register `SchoolContext` as a service, open *Startup.cs*, and add the highlighted lines to the `ConfigureServices` method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Add `using` statements for `ContosoUniversity.Data` and `Microsoft.EntityFrameworkCore` namespaces, and then build the project.

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Http;
```

Open the *appsettings.json* file and add a connection string as shown in the following example.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

SQL Server Express LocalDB

The connection string specifies a SQL Server LocalDB database. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for application development, not production use. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB creates *.mdf* database files in the `C:/Users/<user>` directory.

Initialize DB with test data

The Entity Framework will create an empty database for you. In this section, you write a method that's called after the database is created in order to populate it with test data.

Here you'll use the `EnsureCreated` method to automatically create the database. In a later tutorial you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the *Data* folder, create a new class file named *DbInitializer.cs* and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-
01")},
            new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
            new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
            new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
            new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
```

```
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollments.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database, and if not, it assumes the database is new and needs to be seeded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

In *Program.cs*, modify the `Main` method to do the following on application startup:

- Get a database context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method is done.

```
public static void Main(string[] args)
{
     var host = CreateWebHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}
```

Add `using` statements:

```
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;
```

In older tutorials, you may see similar code in the `Configure` method in *Startup.cs*. We recommend that you use the `Configure` method only to set up the request pipeline. Application startup code belongs in the `Main` method.

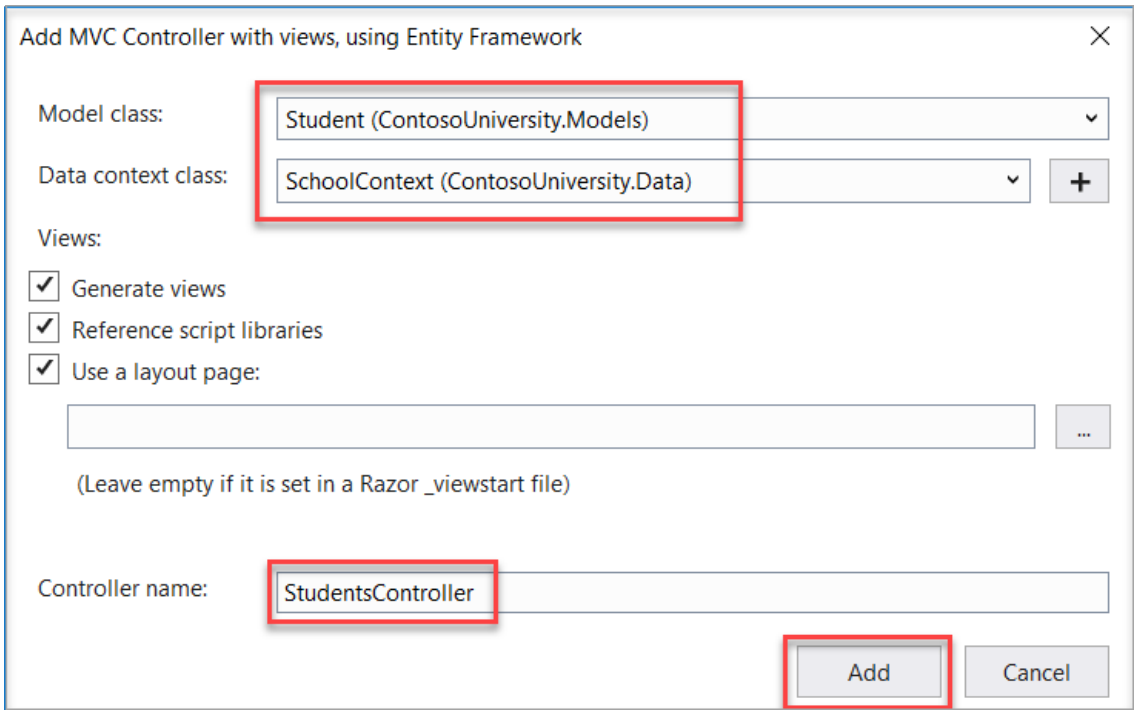
Now the first time you run the application, the database will be created and seeded with test data. Whenever you change your data model, you can delete the database, update your seed method, and start afresh with a new database the same way. In later tutorials, you'll see how to modify the database when the data model changes, without deleting and re-creating it.

Create controller and views

Next, you'll use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding. Scaffolding differs from code generation in that the scaffolded code is a starting point that you can modify to suit your own requirements, whereas you typically don't modify generated code. When you need to customize generated code, you use partial classes or you regenerate the code when things change.

- Right-click the **Controllers** folder in **Solution Explorer** and select **Add > New Scaffolded Item**.
- In the **Add Scaffold** dialog box:
  - Select **MVC controller with views, using Entity Framework**.
  - Click **Add**. The **Add MVC Controller with views, using Entity Framework** dialog box appears.



  - In **Model class** select **Student**.
  - In **Data context class** select **SchoolContext**.
  - Accept the default **StudentsController** as the name.
  - Click **Add**.

  When you click **Add**, the Visual Studio scaffolding engine creates a *StudentsController.cs* file and a set of views (*.cshtml* files) that work with the controller.

(The scaffolding engine can also create the database context for you if you don't create it manually first as you did earlier for this tutorial. You can specify a new context class in the **Add Controller** box by clicking the plus sign to the right of **Data context class**. Visual Studio will then create your `DbContext` class as well as the controller and views.)

You'll notice that the controller takes a `SchoolContext` as a constructor parameter.

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
```

ASP.NET Core dependency injection takes care of passing an instance of `SchoolContext` into the controller. You configured that in the *Startup.cs* file earlier.

The controller contains an `Index` action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the `Students` property of the database context instance:

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

You'll learn about the asynchronous programming elements in this code later in the tutorial.

The *Views/Students/Index.cshtml* view displays this list in a table:

```
@model IEnumerable<ContosoUniversity.Models.Student>

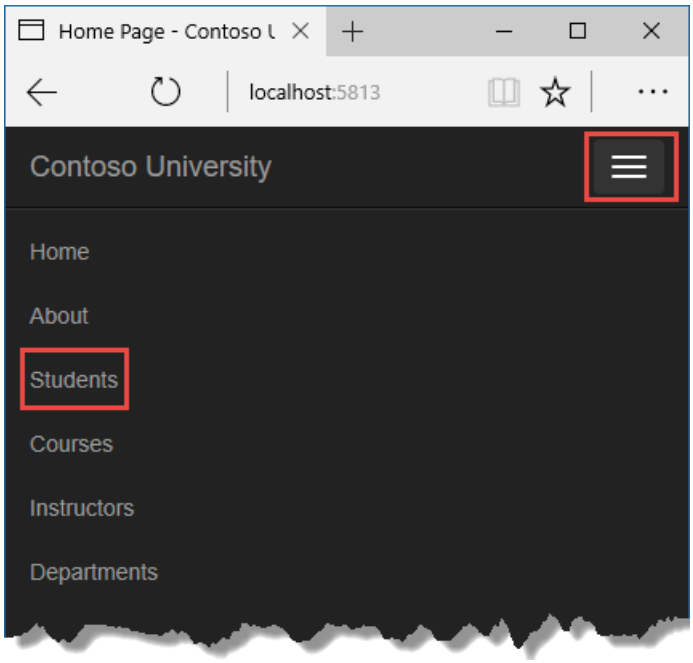
@{
    ViewData["Title"] = "Index";
}

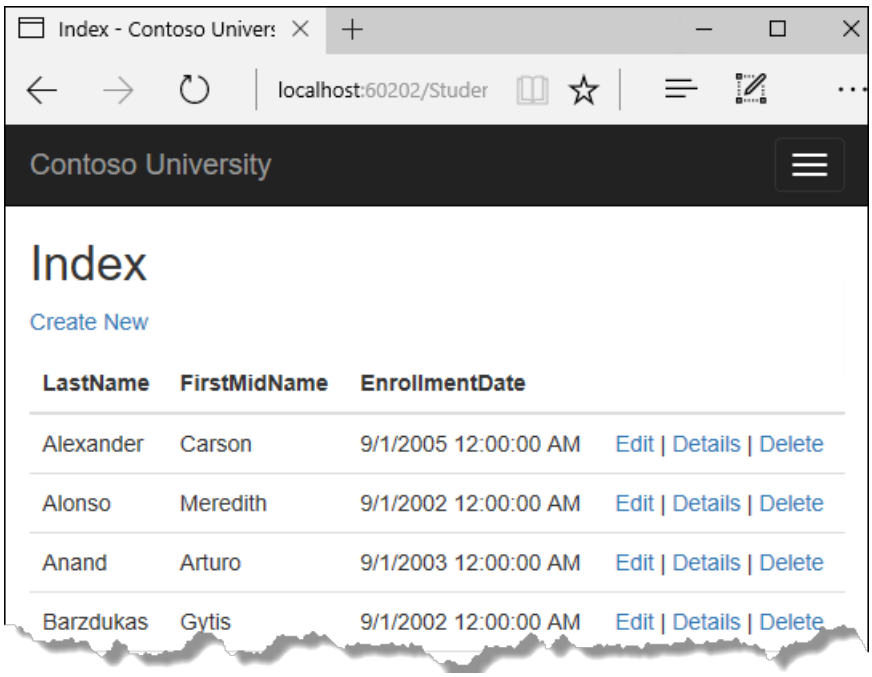
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.LastName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.EnrollmentDate)
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu.

Click the Students tab to see the test data that the `DbInitializer.Initialize` method inserted. Depending on how narrow your browser window is, you'll see the `Students` tab link at the top of the page or you'll have to click the navigation icon in the upper right corner to see the link.





View the database

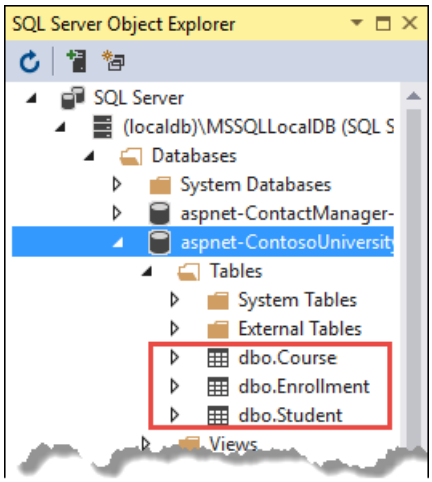
When you started the application, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database and so it created one, then the remainder of the `Initialize` method code populated the database with data. You can use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio.

Close the browser.

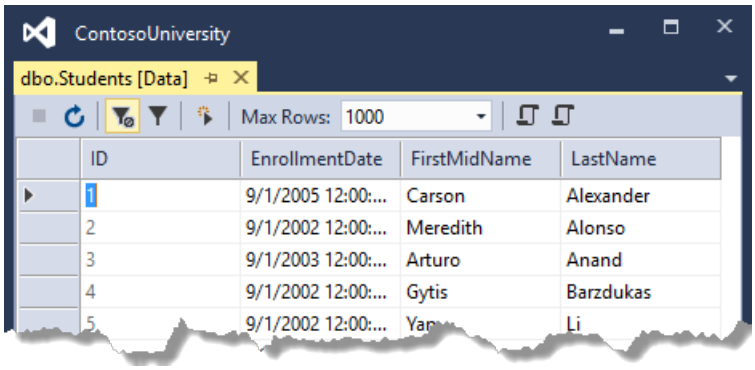
If the SSOX window isn't already open, select it from the **View** menu in Visual Studio.

In SSOX, click **(localdb)\MSSQLLocalDB > Databases**, and then click the entry for the database name that's in the connection string in your *appsettings.json* file.

Expand the **Tables** node to see the tables in your database.



Right-click the **Student** table and click **View Data** to see the columns that were created and the rows that were inserted into the table.



The *.mdf* and *.ldf* database files are in the *C:\Users\<yourusername>* folder.

Because you're calling `EnsureCreated` in the initializer method that runs on app start, you could now make a change to the `Student` class, delete the database, run the application again, and the database would automatically be re-created to match your change. For example, if you add an `EmailAddress` property to the `Student` class, you'll see a new `EmailAddress` column in the re-created table.

Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of conventions, or assumptions that the Entity Framework makes.

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named ID or classnameID are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named *<navigation property name><primary key property name>* (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply *<primary key property name>* (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

Conventional behavior can be overridden. For example, you can explicitly specify table names, as you saw earlier in this tutorial. And you can set column names and set any property as primary key or foreign key, as you'll see in a later tutorial in this series.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the `async` keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that's returned.
- The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when you are writing asynchronous code that uses the Entity Framework:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes, for example, `ToListAsync`, `SingleOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include, for example, statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF context isn't thread safe: don't try to do multiple operations in parallel. When you call any async EF method, always use the `await` keyword.
- If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

For more information about asynchronous programming in .NET, see Async Overview.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Created ASP.NET Core MVC web app
- Set up the site style

- Learned about EF Core NuGet packages
- Created the data model
- Created the database context
- Registered the SchoolContext
- Initialized DB with test data
- Created controller and views
- Viewed the database

In the following tutorial, you'll learn how to perform basic CRUD (create, read, update, delete) operations.

Advance to the next tutorial to learn how to perform basic CRUD (create, read, update, delete) operations.

Implement basic CRUD functionality

Tutorial: Implement CRUD Functionality - ASP.NET MVC with EF Core

9/22/2020 • 19 minutes to read • Edit Online

In the previous tutorial, you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial, you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

> **NOTE**
> It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about repositories with EF, see the last tutorial in this series.

In this tutorial, you:

- Customize the Details page
- Update the Create page
- Update the Edit page
- Update the Delete page
- Close database connections

Prerequisites

- Get started with EF Core and ASP.NET Core MVC

Customize the Details page

The scaffolded code for the Students Index page left out the `Enrollments` property, because that property holds a collection. In the **Details** page, you'll display the contents of the collection in an HTML table.

In *Controllers/StudentsController.cs*, the action method for the Details view uses the `SingleOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls `Include`. `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .Include(s => s.Enrollments)
            .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. You'll learn more about these methods in the read related data tutorial.

The `AsNoTracking` method improves performance in scenarios where the entities returned won't be updated in the current context's lifetime. You'll learn more about `AsNoTracking` at the end of this tutorial.

Route data

The key value that's passed to the `Details` method comes from *route data*. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the `Index` method `id` parameter if you pass it as a query string value:

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the `id` parameter matches the default route, so `id` is added to the route data.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, `studentID` doesn't match a parameter in the default route, so it's added as a query string.

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

For more information about tag helpers, see Tag Helpers in ASP.NET Core.

Add enrollments to the Details view

Open *Views/Students/Details.cshtml*. Each field is displayed using `DisplayNameFor` and `DisplayFor` helpers, as shown in the following example:

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd class="col-sm-10">
    @Html.DisplayFor(model => model.LastName)
</dd>
```

After the last field and immediately before the closing `</dl>` tag, add the following code to display a list of enrollments:

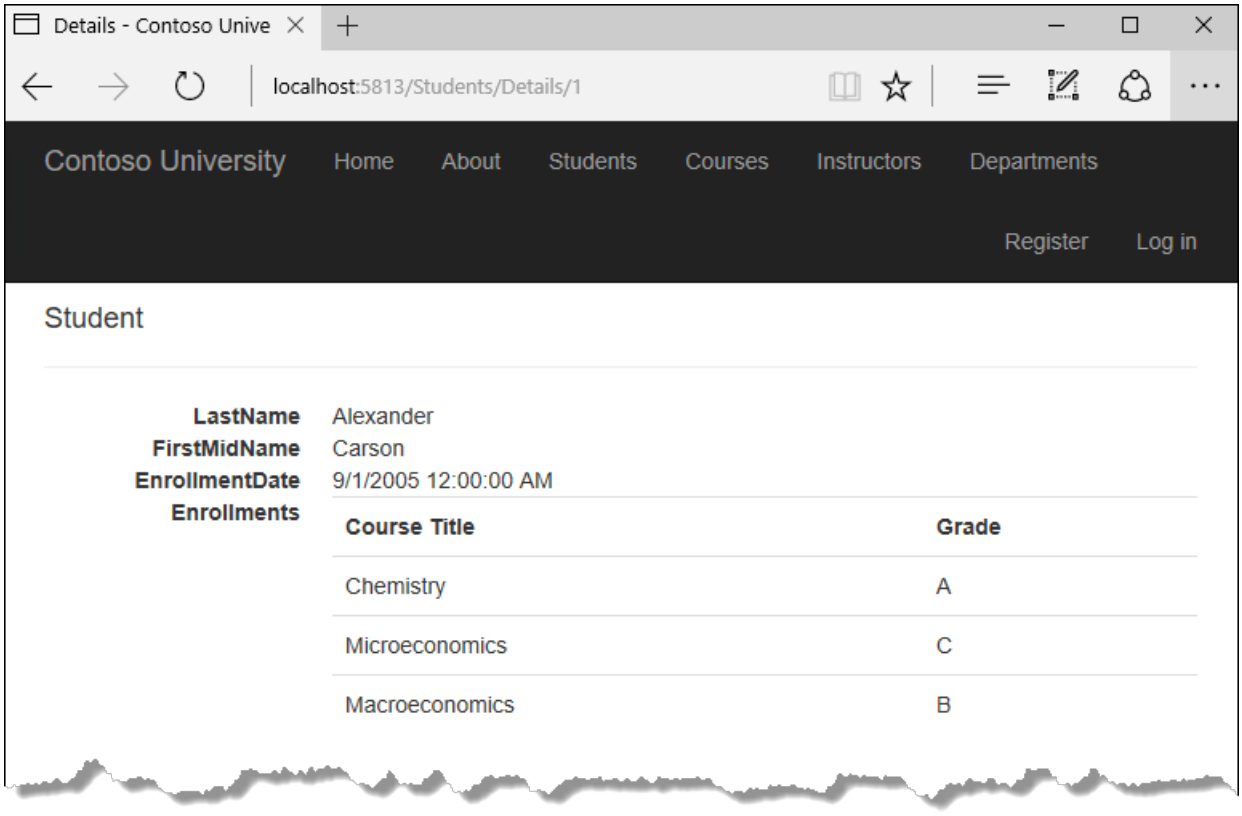
```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd class="col-sm-10">
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. You see the list of courses and grades

for the selected student:



Update the Create page

In *StudentsController.cs*, modify the HttpPost `Create` method by adding a try-catch block and removing ID from the `Bind` attribute.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

This code adds the Student entity created by the ASP.NET Core MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET Core MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed `ID` from the `Bind` attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user doesn't set the ID value.

Other than the `Bind` attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from `DbUpdateException` is caught while the changes are being saved, a generic error message is displayed. `DbUpdateException` exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in Monitoring and Telemetry (Building Real-World Cloud Apps with Azure).

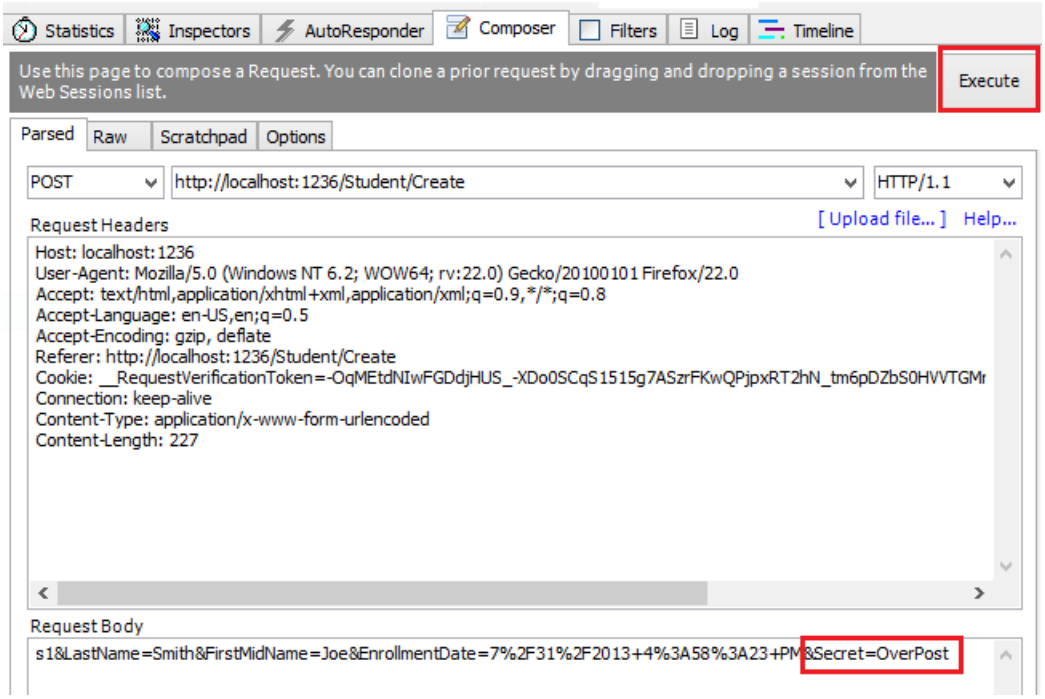
The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the FormTagHelper and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Security note about overposting

The `Bind` attribute that the scaffolded code includes on the `Create` method is one way to protect against overposting in create scenarios. For example, suppose the Student entity includes a `Secret` property that you don't want this web page to set.

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a Student instance, the model binder would pick up that `Secret` form value and use it to create the Student entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to set that property.

You can prevent overposting in edit scenarios by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That's the method used in these tutorials.

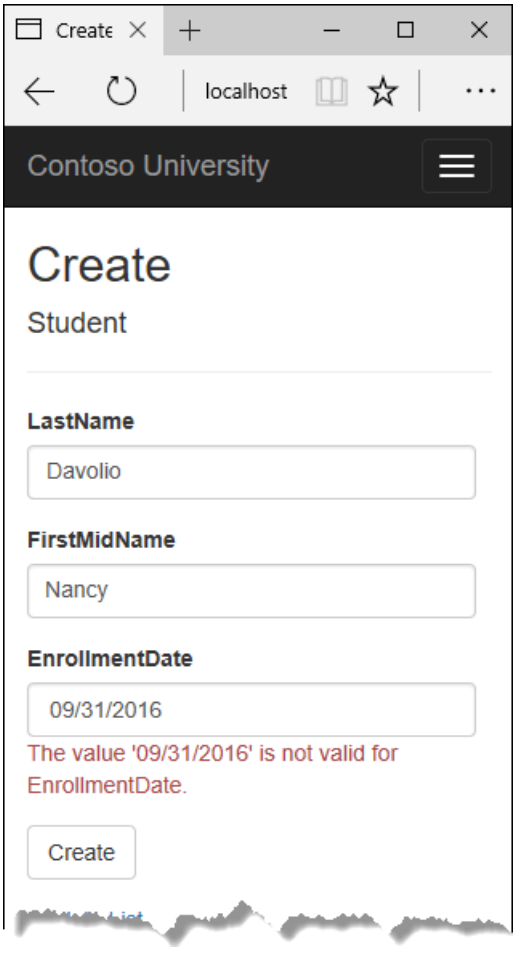
An alternative way to prevent overposting that's preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use `_context.Entry` on the entity instance to set its state to `Unchanged`, and then set `Property("PropertyName").IsModified` to true on each entity property that's included in the view model. This method works in both edit and create scenarios.

Test the Create page

The code in *Views/Students/Create.cshtml* uses `label`, `input`, and `span` (for validation messages) tag helpers for each field.

Run the app, select the **Students** tab, and click **Create New**.

Enter names and a date. Try entering an invalid date if your browser lets you do that. (Some browsers force you to use a date picker.) Then click **Create** to see the error message.



This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the `Create` method.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

Update the Edit page

In *StudentController.cs*, the HttpGet `Edit` method (the one without the `HttpPost` attribute) uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the `Details` method. You don't need to change this method.

Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.FirstOrDefaultAsync(s => s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}
```

These changes implement a security best practice to prevent overposting. The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a `Modified` flag. That code isn't recommended for many scenarios because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter.

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity based on user input in the posted form data. The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

As a best practice to prevent overposting, the fields that you want to be updateable by the **Edit** page are declared in the `TryUpdateModel` parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the HttpPost `Edit` method is the same as the HttpGet `Edit` method; therefore you've renamed the method `EditPost`.

Alternative HttpPost Edit code: Create and attach

The recommended HttpPost edit code ensures that only changed columns get updated and preserves data in properties that you don't want included for model binding. However, the read-first approach requires an extra database read, and can result in more complex code for handling concurrency conflicts. An alternative is to attach an entity created by the model binder to the EF context and mark it as modified. (Don't update your project with this code, it's only shown to illustrate an optional approach.)

```
public async Task<IActionResult> Edit(int id, [Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student
student)
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}
```

You can use this approach when the web page UI includes all of the fields in the entity and can update any of them.

The scaffolded code uses the create-and-attach approach but only catches `DbUpdateConcurrencyException` exceptions and returns 404 error codes. The example shown catches any database update exception and displays an error message.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the `Add` method, that entity's state is set to `Added`. Then when you call the `SaveChanges` method, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`. The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.
- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. You read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

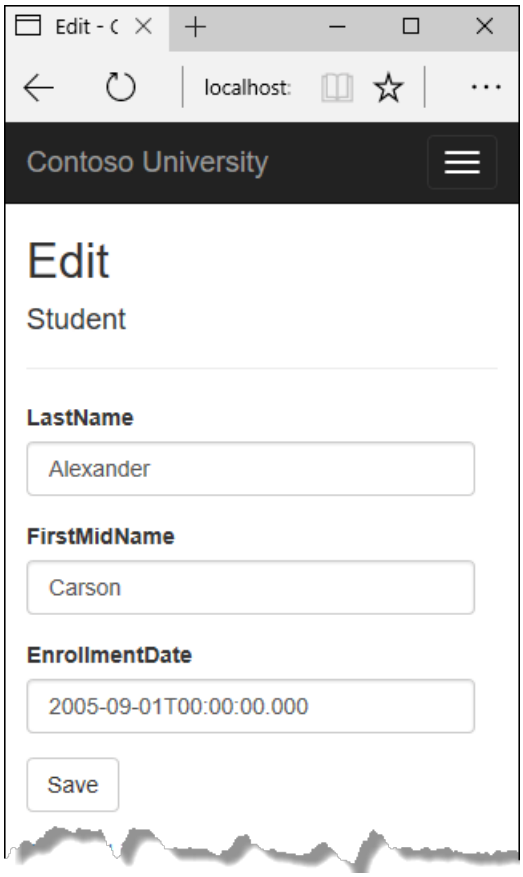
In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the HttpPost `Edit` action method is called, a new web request is made and you have a new instance of the `DbContext`. If you re-read the entity in that new context, you simulate desktop processing.

But if you don't want to do the extra read operation, you have to use the entity object created by the model binder. The simplest way to do this is to set the entity state to Modified as is done in the alternative HttpPost Edit code shown earlier. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want to avoid the read-first approach, but you also want the SQL UPDATE statement to update only the fields that the user actually changed, the code is more complex. You have to save the original values in some way (such as by using hidden fields) so that they're available when the HttpPost `Edit` method is called. Then you can create a Student entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`.

Test the Edit page

Run the app, select the **Students** tab, then click an **Edit** hyperlink.



Change some of the data and click **Save**. The **Index** page opens and you see the changed data.

Update the Delete page

In *StudentController.cs*, the template code for the HttpGet `Delete` method uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that's called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the HttpPost `Delete` method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the HttpPost `Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the HttpPost Delete method calls the HttpGet Delete method, passing it a parameter that indicates that an error has occurred. The HttpGet Delete method then redisplays the confirmation

page along with the error message, giving the user an opportunity to cancel or try again.

Replace the HttpGet `Delete` action method with the following code, which manages error reporting.

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the HttpGet `Delete` method is called without a previous failure. When it's called by the HttpPost `Delete` method in response to a database update error, the parameter is true and an error message is passed to the view.

The read-first approach to HttpPost Delete

Replace the HttpPost `Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students.FindAsync(id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

This code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When

`SaveChanges` is called, a SQL DELETE command is generated.

The create-and-attach approach to HttpPost Delete

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query by instantiating a Student entity using only the primary key value and then setting the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity. (Don't put this code in your project; it's here just to illustrate an alternative.)

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

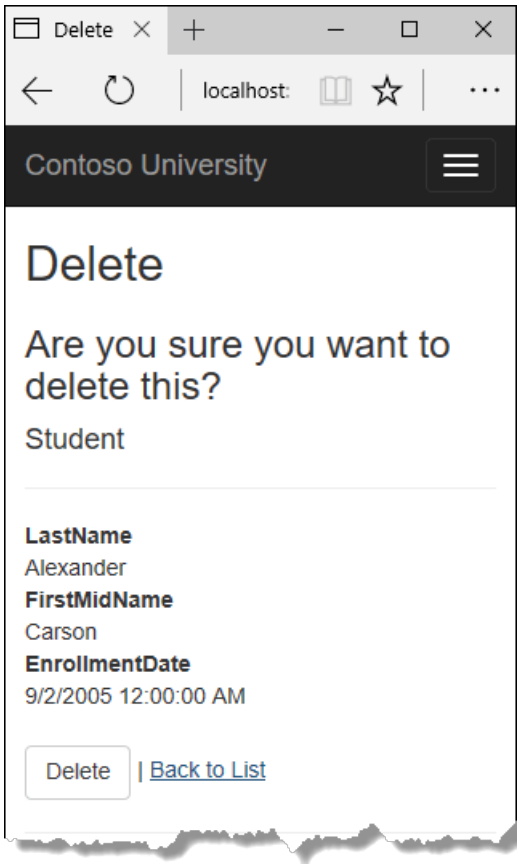
If the entity has related data that should also be deleted, make sure that cascade delete is configured in the database. With this approach to entity deletion, EF might not realize there are related entities to be deleted.

Update the Delete view

In *Views/Student/Delete.cshtml*, add an error message between the h2 heading and the h3 heading, as shown in the following example:

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the app, select the **Students** tab, and click a **Delete** hyperlink:



Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Close database connections

To free up the resources that a database connection holds, the context instance must be disposed as soon as possible when you are done with it. The ASP.NET Core built-in dependency injection takes care of that task for you.

In *Startup.cs*, you call the AddDbContext extension method to provision the `DbContext` class in the ASP.NET Core DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handle transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or they all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by calling the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

- During the context lifetime you don't need to update any entities, and you don't need EF to automatically load navigation properties with entities retrieved by separate queries. Frequently these conditions are met in a controller's HttpGet action methods.
- You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.
- You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see Tracking vs. No-Tracking.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Customized the Details page
- Updated the Create page
- Updated the Edit page
- Updated the Delete page
- Closed database connections

Advance to the next tutorial to learn how to expand the functionality of the **Index** page by adding sorting, filtering, and paging.

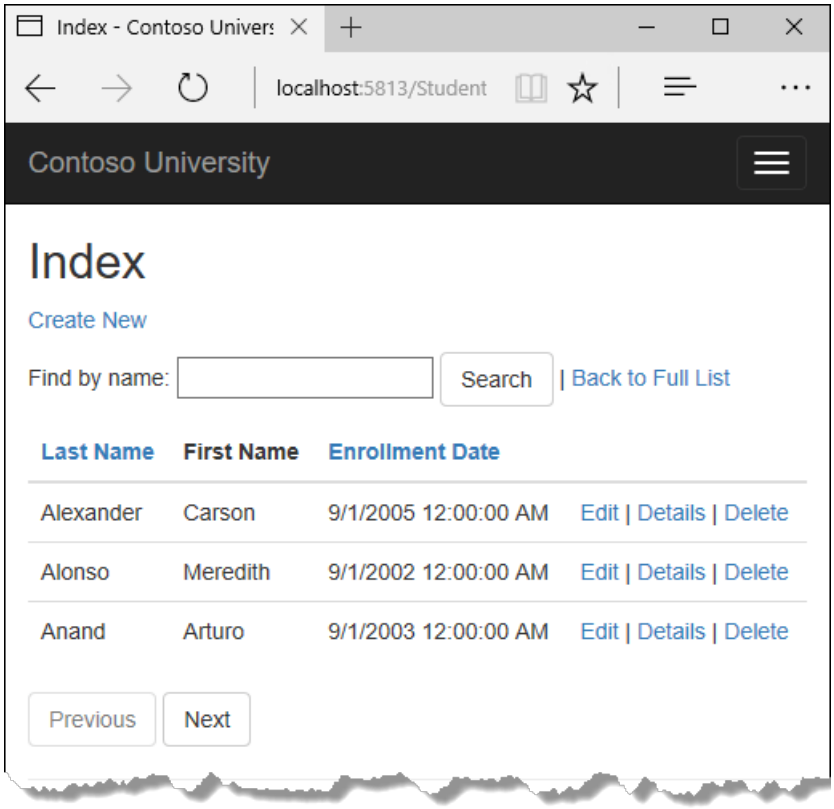
Next: Sorting, filtering, and paging

Tutorial: Add sorting, filtering, and paging - ASP.NET MVC with EF Core

9/22/2020 • 14 minutes to read • Edit Online

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for Student entities. In this tutorial you'll add sorting, filtering, and paging functionality to the Students Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



In this tutorial, you:

- Add column sort links
- Add a Search box
- Add paging to Students Index
- Add paging to Index method
- Add paging links
- Create an About page

Prerequisites

- Implement CRUD Functionality

Add column sort links

To add sorting to the Student Index page, you'll change the `Index` method of the Students controller and add code

to the Student Index view.

Add sorting Functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code:

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewData` elements (NameSortParm and DateSortParm) are used by the view to configure the column heading hyperlinks with the appropriate query string values.

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, NameSortParm should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an `IQueryable` variable before the switch statement, modifies it in the switch statement, and calls the `ToListAsync` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query isn't executed until you convert the `IQueryable` object into a collection by calling a method such as `ToListAsync`. Therefore, this code results in a single query that's not executed until the `return View` statement.

This code could get verbose with a large number of columns. The last tutorial in this series shows how to write code that lets you pass the name of the `OrderBy` column in a string variable.

Add column heading hyperlinks to the Student Index view

Replace the code in *Views/Students/Index.cshtml*, with the following code to add column heading hyperlinks. The changed lines are highlighted.

```
@model IEnumerable<ContosoUniversity.Models.Student>

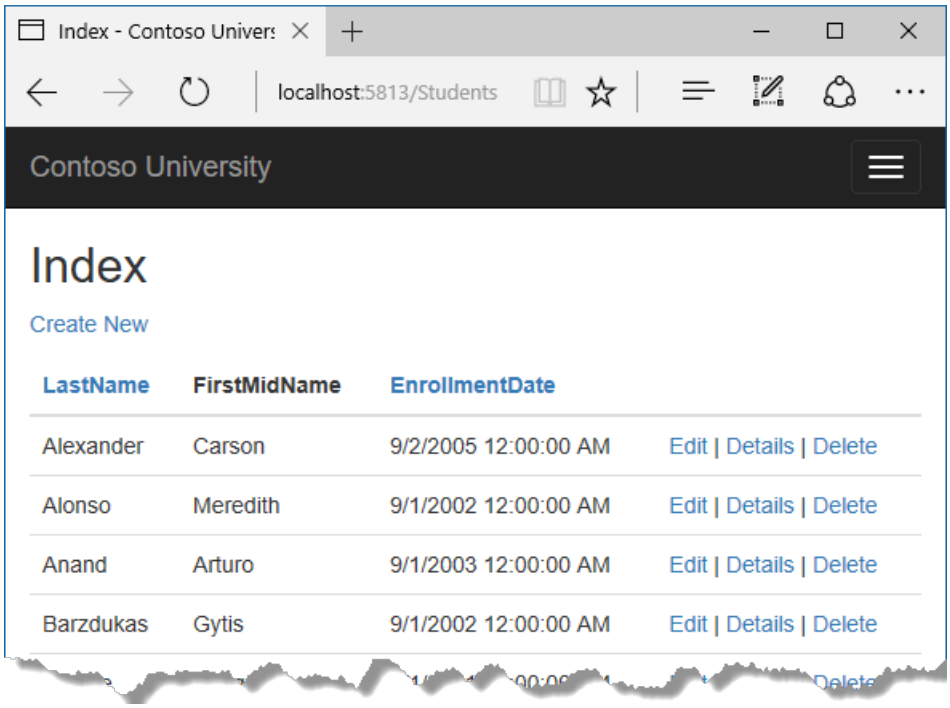
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

This code uses the information in `ViewData` properties to set up hyperlinks with the appropriate query string values.

Run the app, select the **Students** tab, and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Add a Search box

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code (the changes are highlighted).

```
public async Task<IActionResult> Index(string sortOrder, string searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

You've added a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

> **NOTE**
>
> Here you are calling the `Where` method on an `IQueryable` object, and the filter will be processed on the server. In some scenarios you might be calling the `Where` method as an extension method on an in-memory collection. (For example, suppose you change the reference to `_context.Students` so that instead of an EF `DbSet` it references a repository method that returns an `IEnumerable` collection.) The result would normally be the same but in some cases may be different.
>
> For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there's a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

In *Views/Student/Index.cshtml*, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

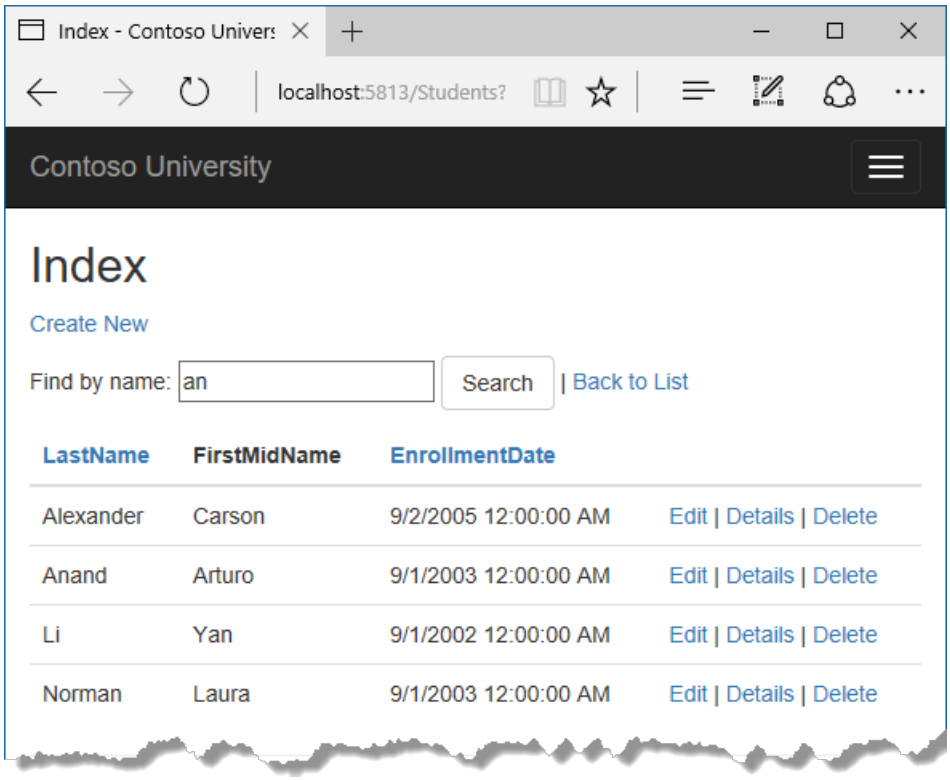
```
<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
```

This code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action doesn't result in an update.

Run the app, select the **Students** tab, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

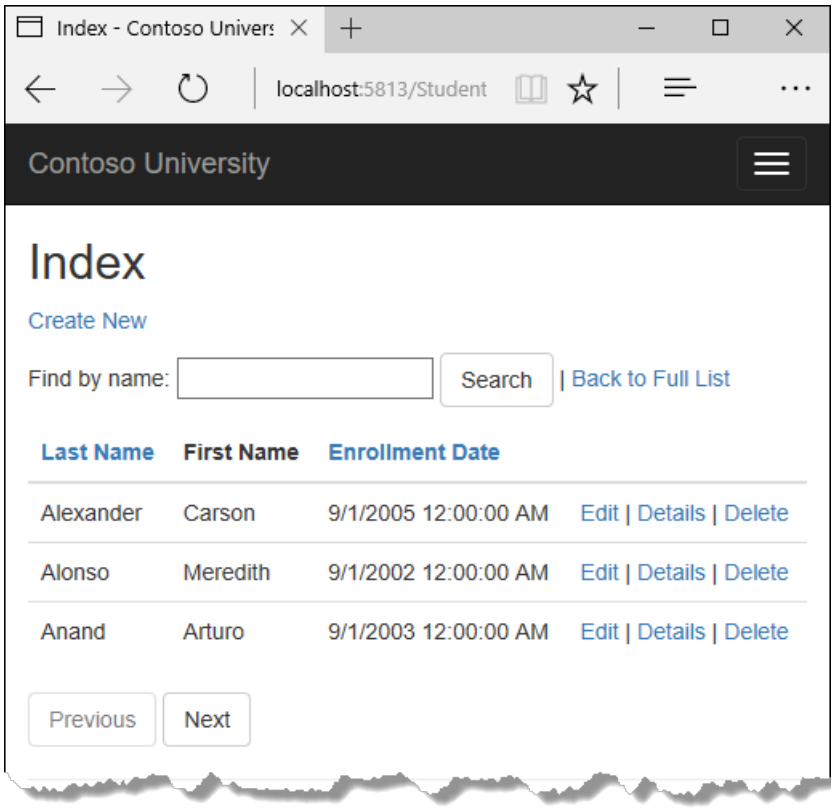
```
http://localhost:5813/Students?SearchString=an
```

If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding `method="get"` to the `form` tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the **Search** box. You'll fix that in the next section.

Add paging to Students Index

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs`, and then replace the template code with the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int
pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in this code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it will return a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` can be used to enable or disable **Previous** and **Next** paging buttons.

A `CreateAsync` method is used instead of a constructor to create the `PaginatedList<T>` object because constructors can't run asynchronous code.

Add paging to Index method

In *StudentsController.cs*, replace the `Index` method with the following code.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
        pageNumber = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1, pageSize));
}
```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The `ViewData` element named CurrentSort provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The `ViewData` element named CurrentFilter provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the `searchString` parameter isn't null.

```
if (searchString != null)
{
    pageNumber = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the `Index` method, the `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1, pageSize));
```

The `PaginatedList.CreateAsync` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(pageNumber ?? 1)` means return the value of `pageNumber` if it has a value, or return 1 if `pageNumber` is null.

Add paging links

In *Views/Students/Index.cshtml*, replace the existing code with the following code. The changes are highlighted.

```
@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
```

```
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex + 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

The `@model` statement at the top of the page specifies that the view now gets a `PaginatedList<T>` object instead of a `List<T>` object.

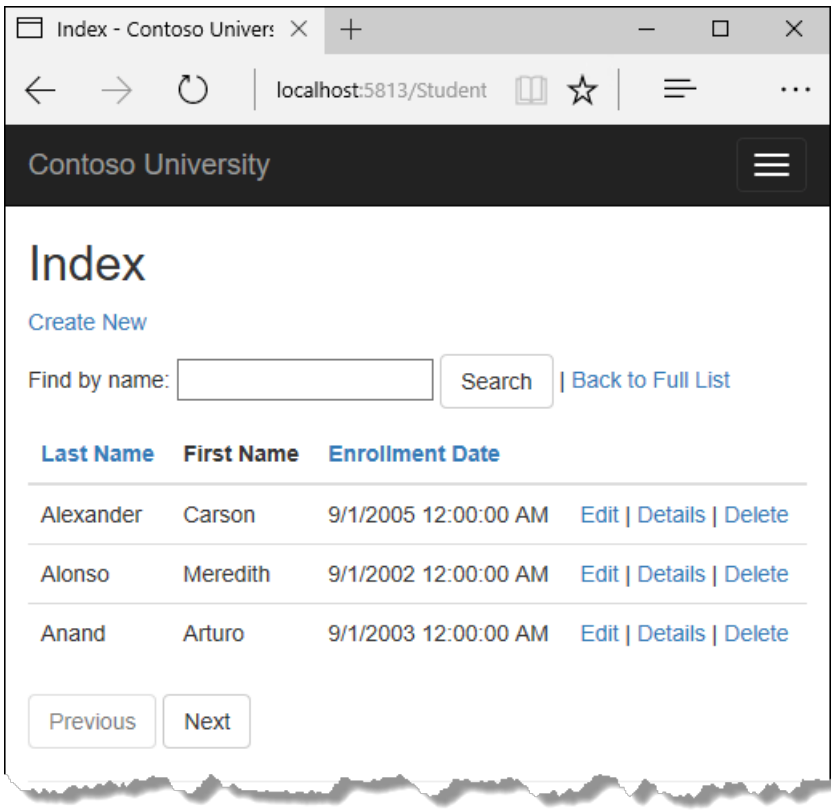
The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter
="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
   Previous
</a>
```

Run the app and go to the Students page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

Create an About page

For the Contoso University website's **About** page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Create the About method in the Home controller.
- Create the About view.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the new folder, add a class file *EnrollmentDateGroup.cs* and replace the template code with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Modify the Home Controller

In *HomeController.cs*, add the following using statements at the top of the file:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;

    public HomeController(SchoolContext context)
    {
        _context = context;
    }
```

Add an `About` method with the following code:

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Create the About View

Add a *Views/Home/About.cshtml* file with the following code:

```
@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and go to the About page. The count of students for each enrollment date is displayed in a table.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Added column sort links
- Added a Search box
- Added paging to Students Index
- Added paging to Index method
- Added paging links
- Created an About page

Advance to the next tutorial to learn how to handle data model changes by using migrations.

Next: Handle data model changes

Tutorial: Using the migrations feature - ASP.NET MVC with EF Core

9/22/2020 • 7 minutes to read • Edit Online

In this tutorial, you start using the EF Core migrations feature for managing data model changes. In later tutorials, you'll add more migrations as you change the data model.

In this tutorial, you:

- Learn about migrations
- Change the connection string
- Create an initial migration
- Examine Up and Down methods
- Learn about the data model snapshot
- Apply the migration

Prerequisites

- Sorting, filtering, and paging

About migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You started these tutorials by configuring the Entity Framework to create the database if it doesn't exist. Then each time you change the data model -- add, remove, or change entity classes or change your DbContext class -- you can delete the database and EF creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it's usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The EF Core Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.

To work with migrations, you can use the **Package Manager Console** (PMC) or the CLI. These tutorials show how to use CLI commands. Information about the PMC is at the end of this tutorial.

Change the connection string

In the *appsettings.json* file, change the name of the database in the connection string to ContosoUniversity2 or some other name that you haven't used on the computer you're using.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

This change sets up the project so that the first migration will create a new database. This isn't required to get started with migrations, but you'll see later why it's a good idea.

NOTE

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

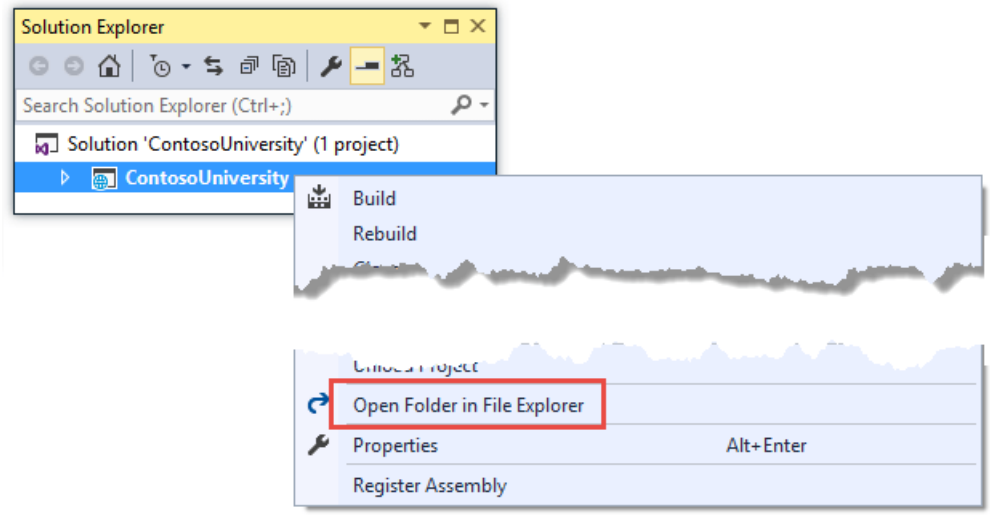
```
dotnet ef database drop
```

The following section explains how to run CLI commands.

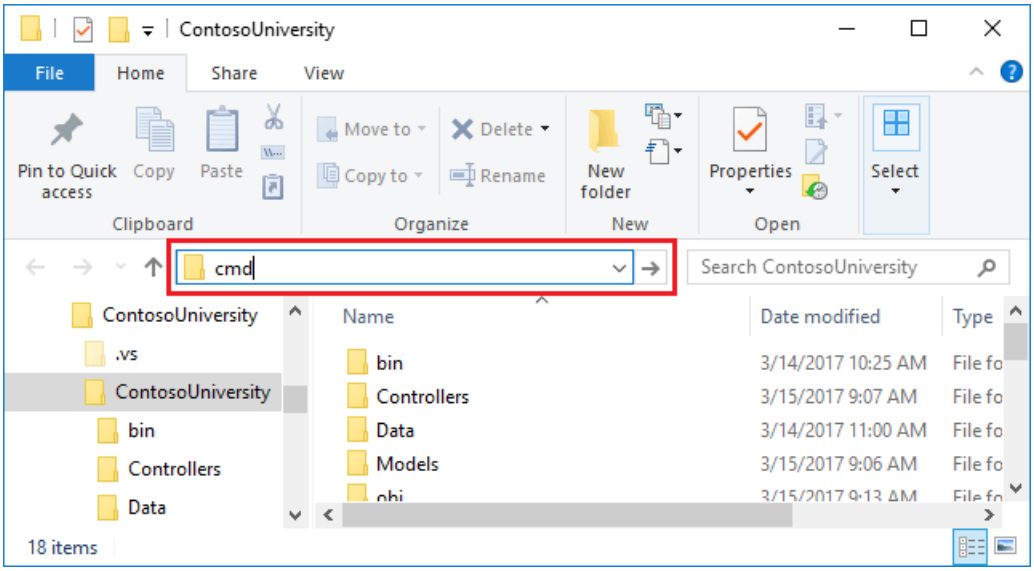
Create an initial migration

Save your changes and build the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

- In **Solution Explorer**, right-click the project and choose **Open Folder in File Explorer** from the context menu.



- Enter "cmd" in the address bar and press Enter.



Enter the following command in the command window:

```
dotnet tool install --global dotnet-ef
dotnet ef migrations add InitialCreate
```

`dotnet tool install --global dotnet-ef` installs `dotnet ef` as a global tool.

In the preceding commands, output similar to the following is displayed:

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.2.0-rtm-35687 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

If you see an error message "*cannot access the file ... ContosoUniversity.dll because it is being used by another process.*", find the IIS Express icon in the Windows System Tray, and right-click it, then click **ContosoUniversity > Stop Site**.

Examine Up and Down methods

When you executed the `migrations add` command, EF generated the code that will create the database from scratch. This code is in the *Migrations* folder, in the file named *<timestamp>_InitialCreate.cs*. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");
        // Additional code not shown
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the `migrations add InitialCreate` command. The migration name parameter ("InitialCreate" in the example) is used for the file name and can be whatever you want. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to

test it first. That's why you changed the name of the database in the connection string earlier -- so that migrations can create a new one from scratch.

The data model snapshot

Migrations creates a *snapshot* of the current database schema in *Migrations/SchoolContextModelSnapshot.cs*. When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

Use the dotnet ef migrations remove command to remove a migration. `dotnet ef migrations remove` deletes the migration and ensures the snapshot is correctly reset. If `dotnet ef migrations remove` fails, use `dotnet ef migrations remove -v` to get more information on the failure.

See EF Core Migrations in Team Environments for more information about how the snapshot file is used.

Apply the migration

In the command window, enter the following command to create the database and tables in it.

```
dotnet ef database update
```

The output from the command is similar to the `migrations add` command, except that you see logs for the SQL commands that set up the database. Most of the logs are omitted in the following sample output. If you prefer not to see this level of detail in log messages, you can change the log level in the *appsettings.Development.json* file. For more information, see Logging in .NET Core and ASP.NET Core.

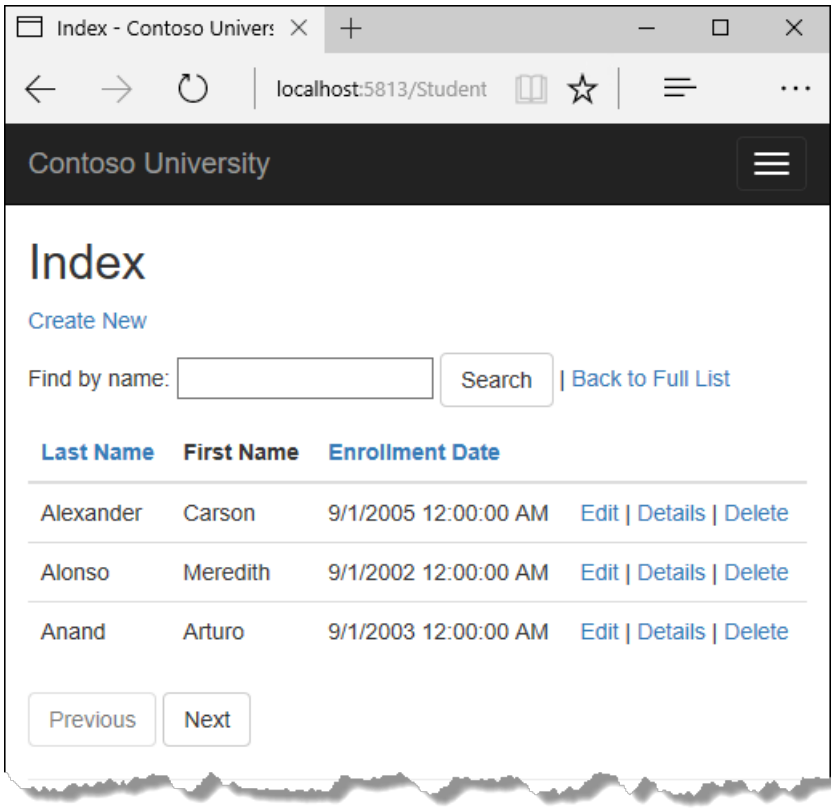
```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.2.0-rtm-35687 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (274ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (60ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [ContosoUniversity2] SET READ_COMMITTED_SNAPSHOT ON;
      END;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );

<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20190327172701_InitialCreate', N'2.2.0-rtm-35687');
Done.
```

Use **SQL Server Object Explorer** to inspect the database as you did in the first tutorial. You'll notice the addition of an __EFMigrationsHistory table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one row for the first migration. (The last log in the preceding CLI output example shows the INSERT statement that creates this row.)

Run the application to verify that everything still works the same as before.



Compare CLI and PMC

The EF tooling for managing migrations is available from .NET Core CLI commands or from PowerShell cmdlets in the Visual Studio **Package Manager Console** (PMC) window. This tutorial shows how to use the CLI, but you can use the PMC if you prefer.

The EF commands for the PMC commands are in the Microsoft.EntityFrameworkCore.Tools package. This package is included in the Microsoft.AspNetCore.App metapackage, so you don't need to add a package reference if your app has a package reference for `Microsoft.AspNetCore.App`.

Important: This isn't the same package as the one you install for the CLI by editing the *.csproj* file. The name of this one ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see .NET Core CLI.

For more information about the PMC commands, see Package Manager Console (Visual Studio).

Get the code

Download or view the completed application.

Next step

In this tutorial, you:

- Learned about migrations
- Learned about NuGet migration packages
- Changed the connection string
- Created an initial migration
- Examined Up and Down methods
- Learned about the data model snapshot
- Applied the migration

Advance to the next tutorial to begin looking at more advanced topics about expanding the data model. Along the way you'll create and apply additional migrations.

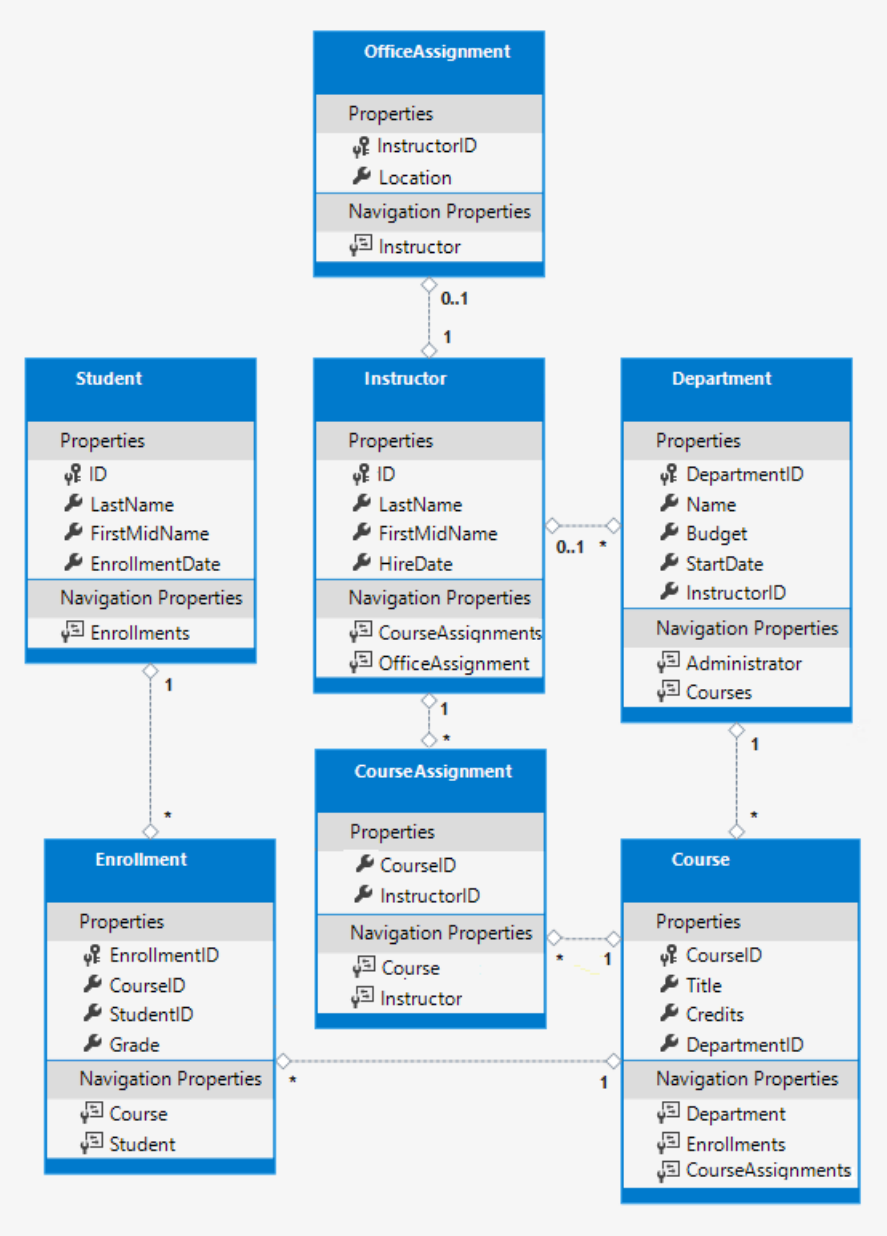
Create and apply additional migrations

Tutorial: Create a complex data model - ASP.NET MVC with EF Core

9/22/2020 • 30 minutes to read • Edit Online

In the previous tutorials, you worked with a simple data model that was composed of three entities. In this tutorial, you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



In this tutorial, you:

- Customize the Data model
- Make changes to Student entity
- Create Instructor entity
- Create OfficeAssignment entity

- Modify Course entity
- Create Department entity
- Modify Enrollment entity
- Update the database context
- Seed database with test data
- Add a migration
- Change the connection string
- Update the database

Prerequisites

- Using EF Core migrations

Customize the Data model

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete School data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In *Models/Student.cs*, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes don't provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

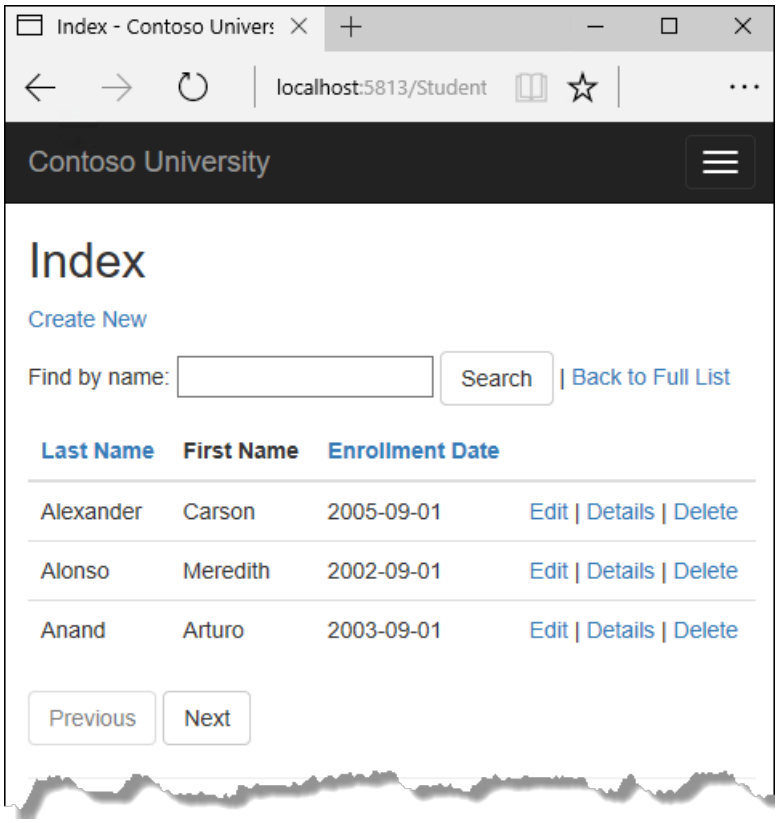
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).
- By default, the browser will render data using the correct format based on your locale.

For more information, see the <input> tag helper documentation.

Run the app, go to the Students Index page and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the Student model.



The StringLength attribute

You can also specify data validation rules and validation error messages using attributes. The `StringLength` attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET Core MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `StringLength` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `StringLength` attribute won't prevent a user from entering white space for a name. You can use the `RegularExpression` attribute to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

The `MaxLength` attribute provides functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

```
dotnet ef migrations add MaxLengthOnNames
```

```
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations creates a file named *<timeStamp>_MaxLengthOnNames.cs*. This file contains code in the `Up` method that will update the database to match the current data model. The `database update` command ran that code.

The timestamp prefixed to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the update-database command, and then all of the migrations are applied in the order in which they were created.

Run the app, select the **Students** tab, click **Create New**, and try to enter either name longer than 50 characters. The application should prevent you from doing this.

The Column attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against

the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they're given the same name as the property name.

In the *Student.cs* file, add a `using` statement for `System.ComponentModel.DataAnnotations.Schema` and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

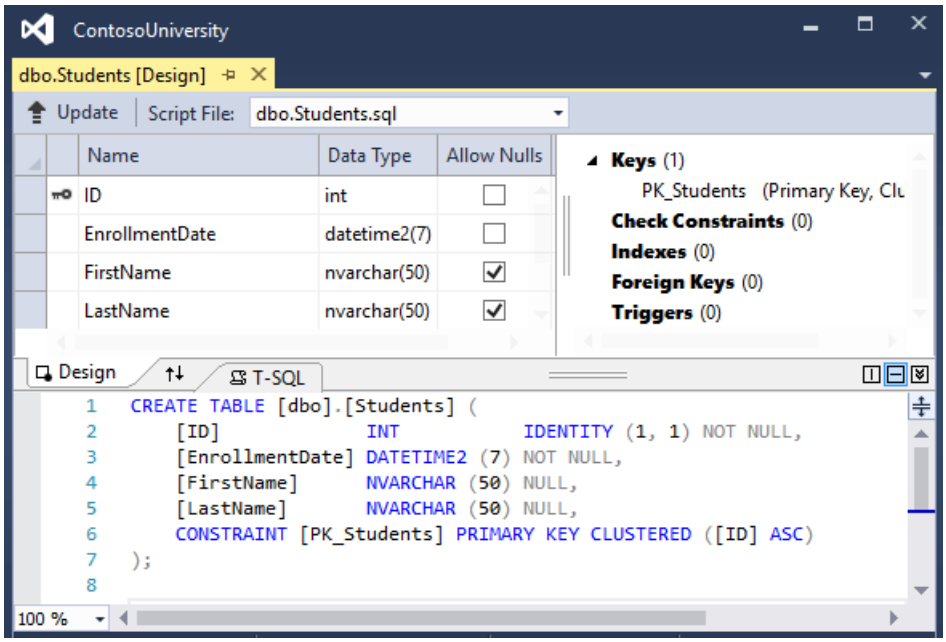
The addition of the `Column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

```
dotnet ef migrations add ColumnFirstName
```

```
dotnet ef database update
```

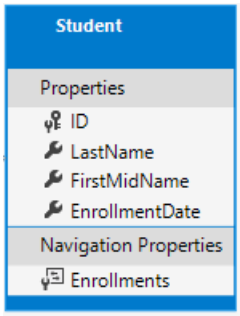
In **SQL Server Object Explorer**, open the Student table designer by double-clicking the **Student** table.



Before you applied the first two migrations, the name columns were of type nvarchar(MAX). They're now nvarchar(50) and the column name has changed from FirstMidName to FirstName.

> **NOTE**
> If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Changes to Student entity



In *Models/Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50)]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

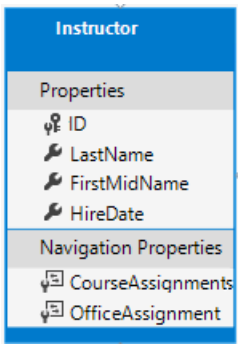
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a get accessor, and no `FullName` column will be generated in the database.

Create Instructor entity



Create *Models/Instructor.cs*, replacing the template code with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the Student and Instructor entities. In the Implementing Inheritance tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the `HireDate` attributes as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

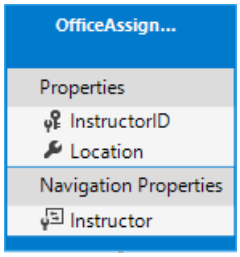
If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single OfficeAssignment entity (which may be null if no office is assigned).

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create OfficeAssignment entity



Create *Models/OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

There's a one-to-zero-or-one relationship between the Instructor and the OfficeAssignment entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the Instructor entity. But the Entity Framework can't automatically recognize InstructorID as the primary key of this entity because its name doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something other than classnameID or ID.

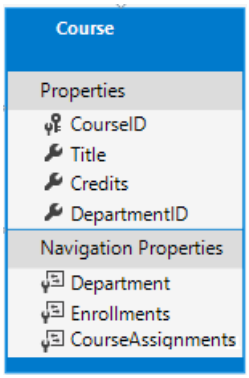
By default, EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The Instructor entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the OfficeAssignment entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an Instructor entity has a related OfficeAssignment entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the `InstructorID` foreign key (which is also the key to this table) is non-nullable.

Modify Course entity



In *Models/Course.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related Department entity and it has a `Department` navigation property.

The Entity Framework doesn't require you to add a foreign key property to your data model when you have a

navigation property for a related entity. EF automatically creates foreign keys in the database wherever they're needed and creates shadow properties for them. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the Department entity is null if you don't load it, so when you update the course entity, you would have to first fetch the Department entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the Department entity before you update.

The DatabaseGenerated attribute

The `DatabaseGenerated` attribute with the `None` parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for Course entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key properties and navigation properties in the Course entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

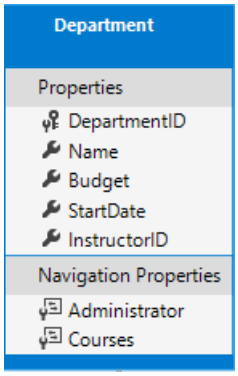
A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained later):

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create Department entity



Create *Models/Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Earlier you used the `Column` attribute to change column name mapping. In the code for the Department entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the Instructor entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation property is named `Administrator` but holds an Instructor entity:

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

A department may have many courses, so there's a Courses navigation property:

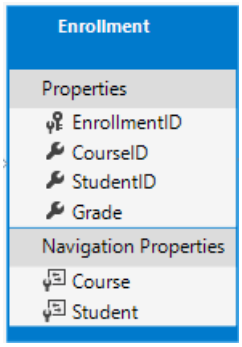
```
public ICollection<Course> Courses { get; set; }
```

NOTE

By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the Department.InstructorID property as nullable, EF would configure a cascade delete rule to delete the department when you delete the instructor, which isn't what you want to have happen. If your business rules required the `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

Modify Enrollment entity



In *Models/Enrollment.cs*, replace the code you added earlier with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The foreign key properties and navigation properties reflect the following relationships:

An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation

property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

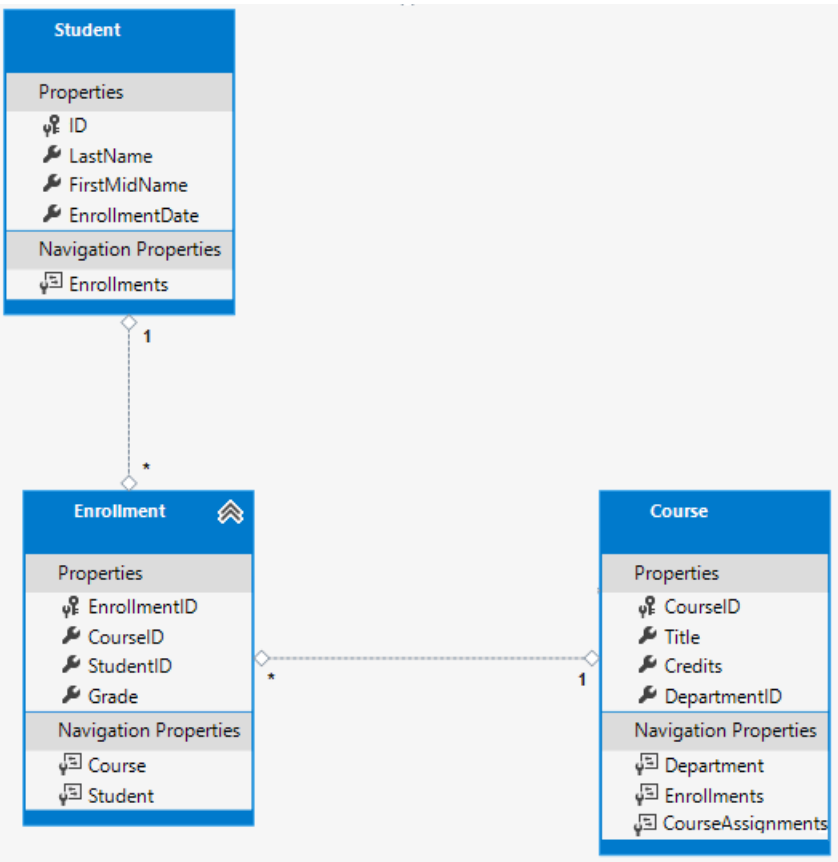
An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many relationships

There's a many-to-many relationship between the Student and Course entities, and the Enrollment entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the Enrollment table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a Grade property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

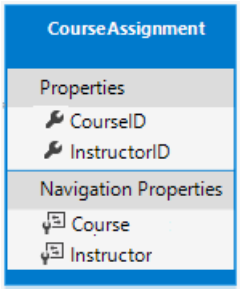


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the Enrollment table didn't include grade information, it would only need to contain the two foreign keys CourseID and StudentID. In that case, it would be a many-to-many join table without payload (or a pure join table) in the database. The Instructor and Course entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

(EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see the discussion in the EF Core GitHub repository.)

The CourseAssignment entity



Create *Models/CourseAssignment.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there's no need for a separate primary key. The *InstructorID* and *CourseID* properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Add the following highlighted code to the *Data/SchoolContext.cs* file:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

About a fluent API alternative

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the EF Core documentation:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

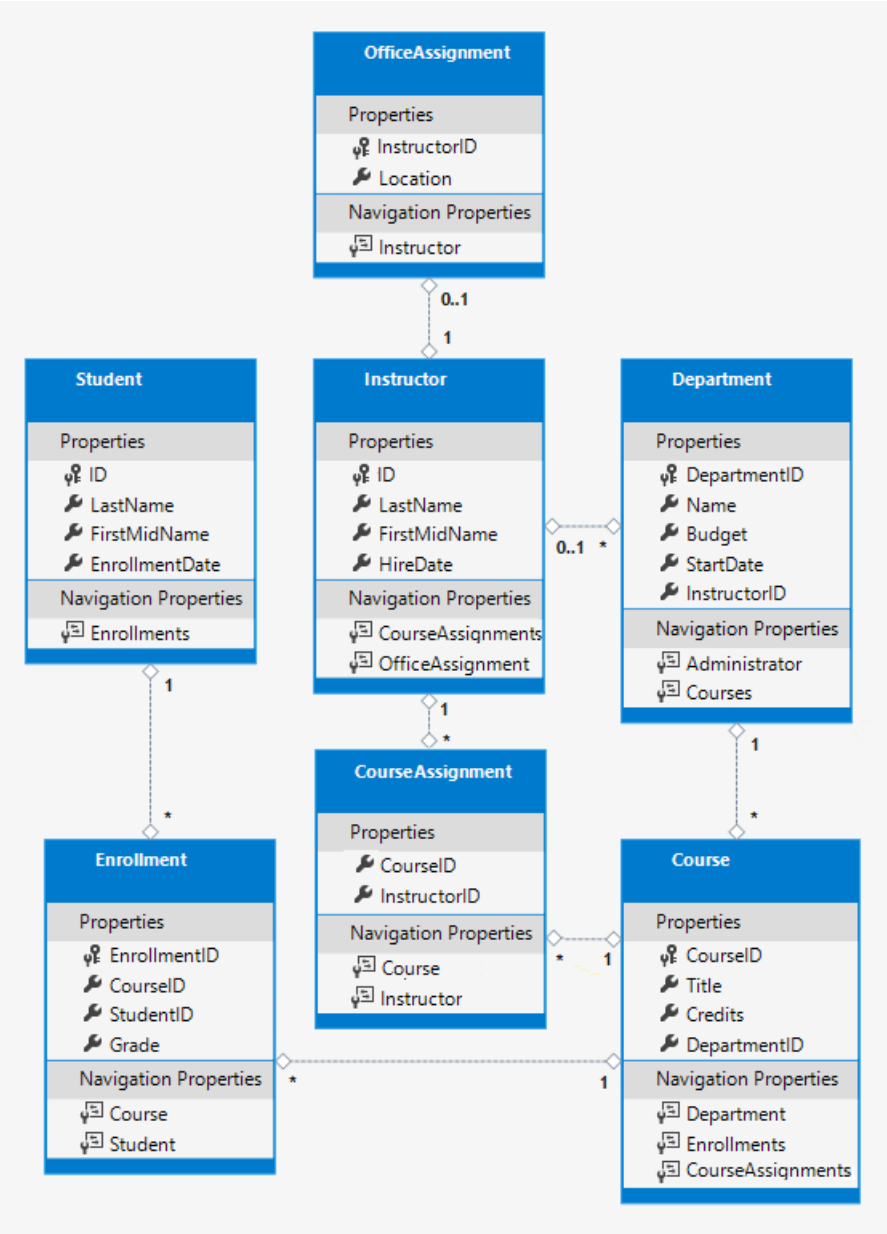
In this tutorial, you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible. If you do use both, note that wherever there's a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the Instructor and OfficeAssignment entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Seed database with test data

Replace the code in the *Data/DbInitializer.cs* file with the following code in order to provide seed data for the new entities you've created.

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
```

```
{
    public static void Initialize(SchoolContext context)
    {
        //context.Database.EnsureCreated();

        // Look for any students.
        if (context.Students.Any())
        {
            return;   // DB has been seeded
        }

        var students = new Student[]
        {
            new Student { FirstMidName = "Carson",   LastName = "Alexander",
                EnrollmentDate = DateTime.Parse("2010-09-01") },
            new Student { FirstMidName = "Meredith", LastName = "Alonso",
                EnrollmentDate = DateTime.Parse("2012-09-01") },
            new Student { FirstMidName = "Arturo",   LastName = "Anand",
                EnrollmentDate = DateTime.Parse("2013-09-01") },
            new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                EnrollmentDate = DateTime.Parse("2012-09-01") },
            new Student { FirstMidName = "Yan",      LastName = "Li",
                EnrollmentDate = DateTime.Parse("2012-09-01") },
            new Student { FirstMidName = "Peggy",    LastName = "Justice",
                EnrollmentDate = DateTime.Parse("2011-09-01") },
            new Student { FirstMidName = "Laura",    LastName = "Norman",
                EnrollmentDate = DateTime.Parse("2013-09-01") },
            new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                EnrollmentDate = DateTime.Parse("2005-09-01") }
        };

        foreach (Student s in students)
        {
            context.Students.Add(s);
        }
        context.SaveChanges();

        var instructors = new Instructor[]
        {
            new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                HireDate = DateTime.Parse("1995-03-11") },
            new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                HireDate = DateTime.Parse("2002-07-06") },
            new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                HireDate = DateTime.Parse("1998-07-01") },
            new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                HireDate = DateTime.Parse("2001-01-15") },
            new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                HireDate = DateTime.Parse("2004-02-12") }
        };

        foreach (Instructor i in instructors)
        {
            context.Instructors.Add(i);
        }
        context.SaveChanges();

        var departments = new Department[]
        {
            new Department { Name = "English",     Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
            new Department { Name = "Mathematics", Budget = 100000,
                StartDate = DateTime.Parse("2007-09-01"),
                InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
            new Department { Name = "Engineering", Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
            new Department { Name = "Economics",   Budget = 100000,
```

```
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
};

foreach (Department d in departments)
{
    context.Departments.Add(d);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
    },
    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
    new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
};

foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var officeAssignments = new OfficeAssignment[]
{
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
        Location = "Smith 17" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
        Location = "Gowan 27" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
        Location = "Thompson 304" },
};

foreach (OfficeAssignment o in officeAssignments)
{
    context.OfficeAssignments.Add(o);
}
context.SaveChanges();

var courseInstructors = new CourseAssignment[]
{
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
```

```
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
    };

    foreach (CourseAssignment ci in courseInstructors)
    {
        context.CourseAssignments.Add(ci);
    }
    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            Grade = Grade.A
        },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            Grade = Grade.C
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
            },
```

```
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Anand").ID,
                    CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
                    Grade = Grade.B
                    },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").ID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If you tried to run the `database update` command at this point (don't do it yet), you would get the following error:

> The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints. The generated code in the `Up` method adds a non-nullable DepartmentID foreign key to the Course table. If there are already rows in the Course table when the code runs, the `AddColumn` operation fails because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing Course rows will all be related to the "Temp" department after the `Up` method runs.

- Open the *{timestamp}_ComplexDataModel.cs* file.
- Comment out the line of code that adds the DepartmentID column to the Course table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

- Add the following highlighted code after the code that creates the Department table:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the Course.DepartmentID column.

Save your changes and build the project.

Change the connection string

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in *appsettings.json* to ContosoUniversity3 or some other name that you haven't used on the computer you're using.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

Save your change to *appsettings.json*.

NOTE

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

```
dotnet ef database drop
```

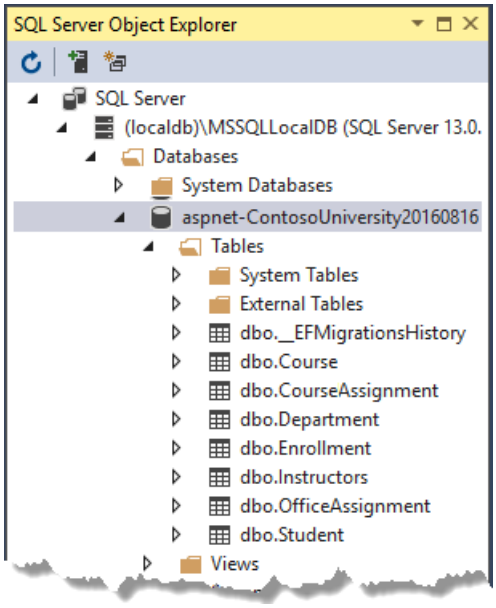
Update the database

After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

```
dotnet ef database update
```

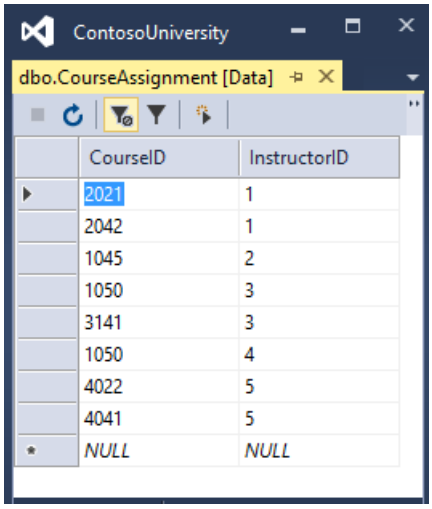
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

Open the database in SSOX as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have SSOX open from the earlier time, click the **Refresh** button.)



Run the app to trigger the initializer code that seeds the database.

Right-click the **CourseAssignment** table and select **View Data** to verify that it has data in it.



Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Customized the Data model
- Made changes to Student entity
- Created Instructor entity
- Created OfficeAssignment entity
- Modified Course entity
- Created Department entity
- Modified Enrollment entity
- Updated the database context
- Seeded database with test data
- Added a migration
- Changed the connection string
- Updated the database

Advance to the next tutorial to learn more about how to access related data.

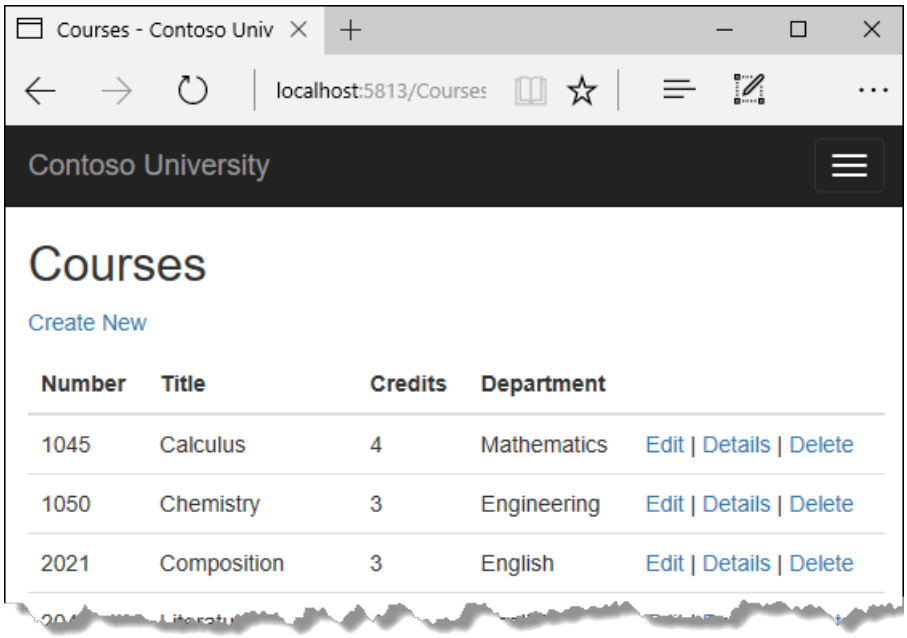
Next: Access related data

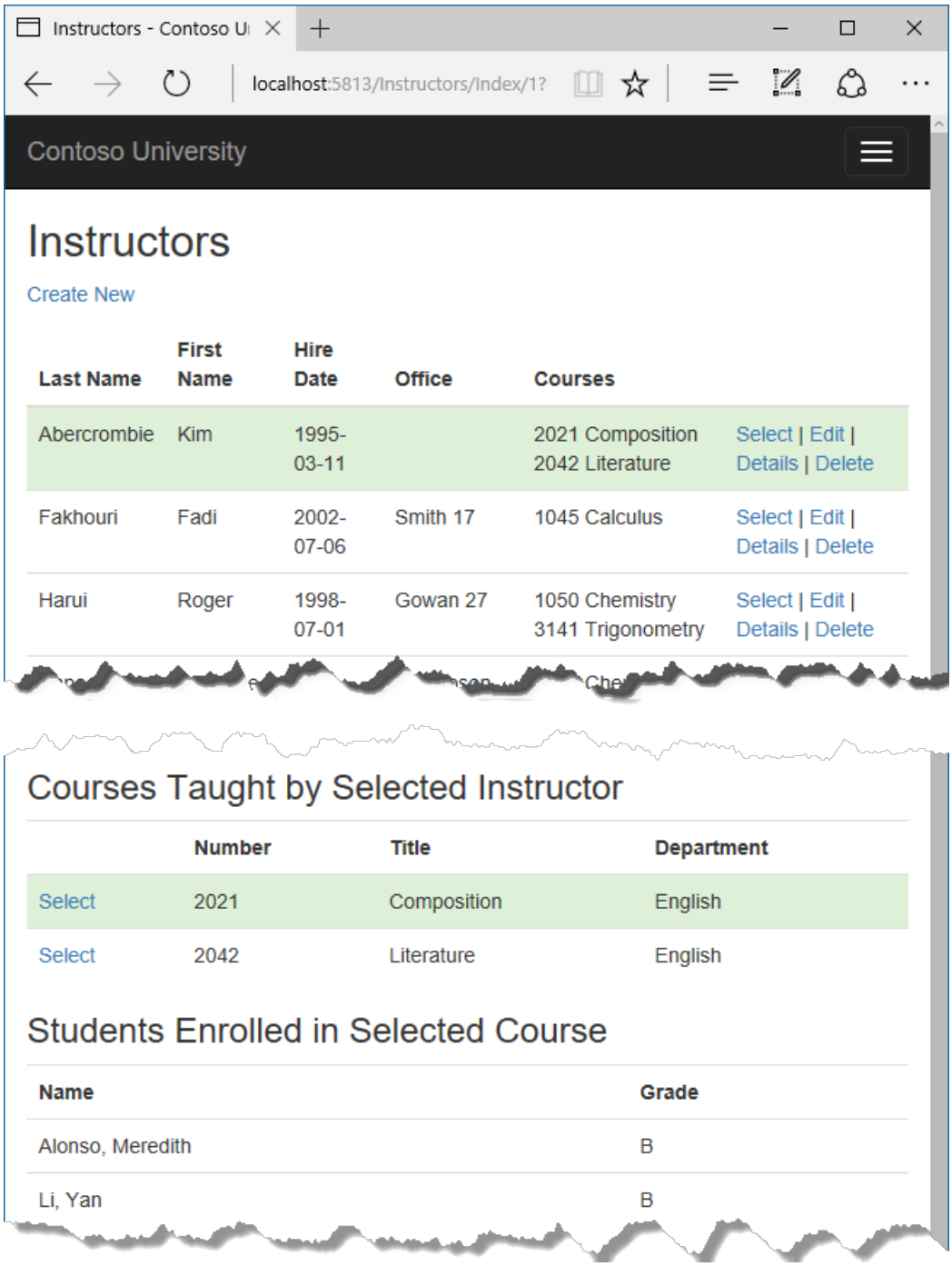
Tutorial: Read related data - ASP.NET MVC with EF Core

9/22/2020 • 14 minutes to read • Edit Online

In the previous tutorial, you completed the School data model. In this tutorial, you'll read and display related data -- that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.





In this tutorial, you:

- Learn how to load related data
- Create a Courses page
- Create an Instructors page
- Learn about explicit loading

Prerequisites

- Create a complex data model

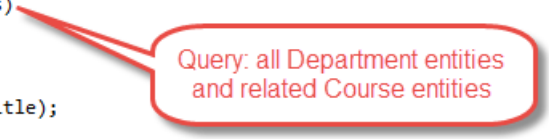
Learn how to load related data

There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

- Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single

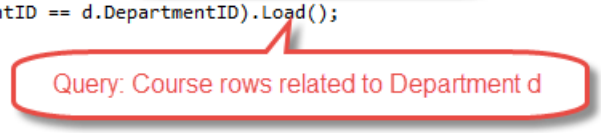
join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```



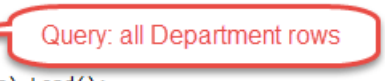
You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of previously retrieved entities. For the query that retrieves related data, you can use the `Load` method instead of a method that returns a list or object, such as `ToList` or `Single`.

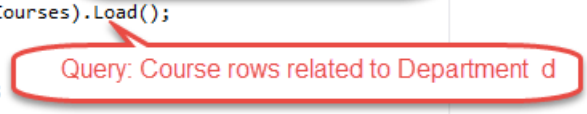
```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```



- Explicit loading. When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed. As in the case of eager loading with separate queries, explicit loading results in multiple queries sent to the database. The difference is that with explicit loading, the code specifies the navigation properties to be loaded. In Entity Framework Core 1.1 you can use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```





- Lazy loading. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. A query is sent to the database each time you try to get data from a navigation property for the first time. Entity Framework Core 1.0 doesn't support lazy loading.

Performance considerations

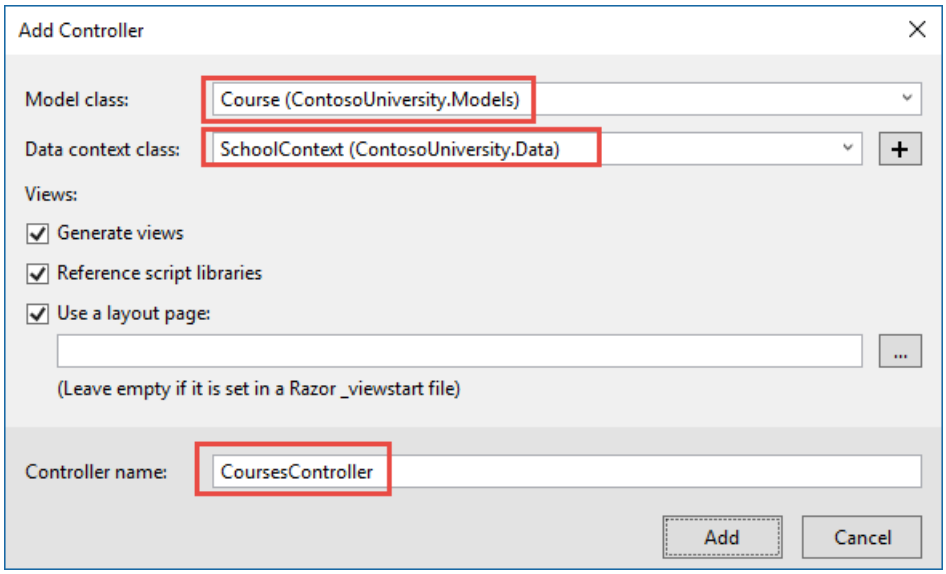
If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager loading of all related data would result in just a single (join) query and a single round trip to the database. A separate query for courses for each department would result in eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, separate queries might perform better because eager loading of everything up front would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Create a Courses page

The Course entity includes a navigation property that contains the Department entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the Name property from the Department entity that's in the `Course.Department` navigation property.

Create a controller named CoursesController for the Course entity type, using the same options for the **MVC Controller with views, using Entity Framework** scaffolder that you did earlier for the Students controller, as shown in the following illustration:



Open *CoursesController.cs* and examine the `Index` method. The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Replace the `Index` method with the following code that uses a more appropriate name for the `IQueryable` that returns Course entities (`courses` instead of `schoolContext`):

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Open *Views/Courses/Index.cshtml* and replace the template code with the following code. The changes are highlighted:

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

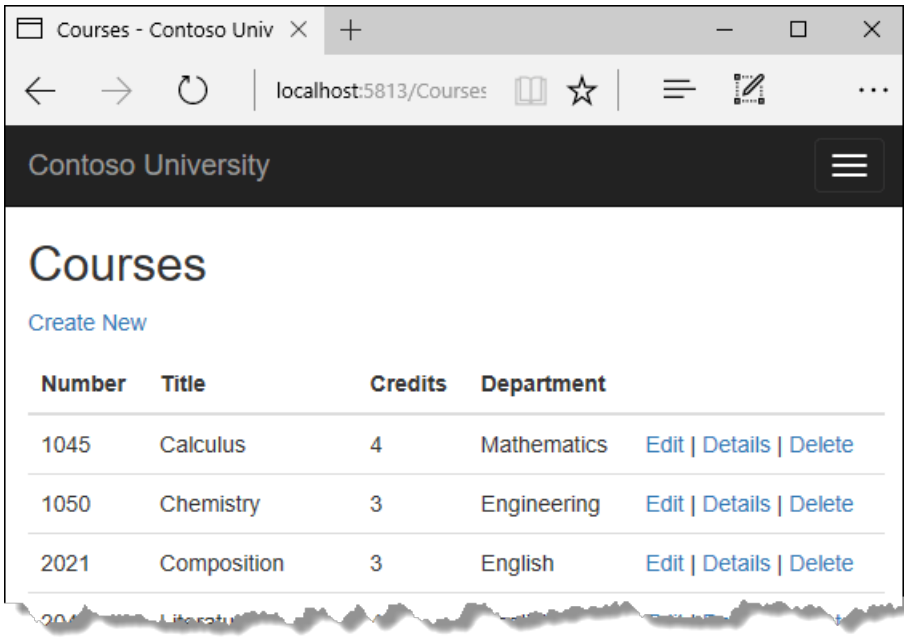
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

You've made the following changes to the scaffolded code:

- Changed the heading from Index to Courses.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the Department entity that's loaded into the `Department` navigation property:

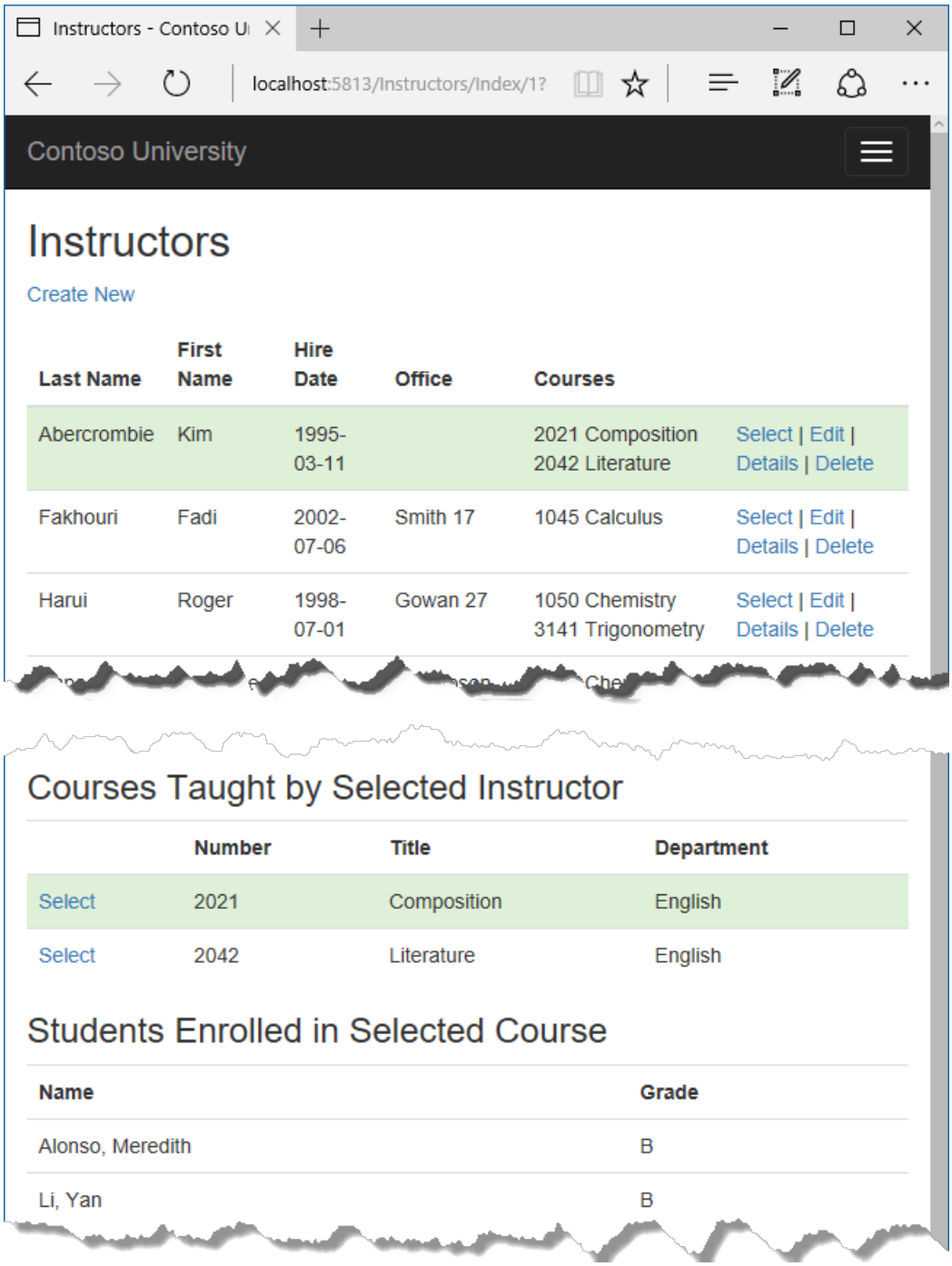
```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Create an Instructors page

In this section, you'll create a controller and view for the Instructor entity in order to display the Instructors page:



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the OfficeAssignment entity. The Instructor and OfficeAssignment entities are in a one-to-zero-or-one relationship. You'll use eager loading for the OfficeAssignment entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.
- When the user selects an instructor, related Course entities are displayed. The Instructor and Course entities are in a many-to-many relationship. You'll use eager loading for the Course entities and their related Department entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the Enrollments entity set is displayed. The Course and Enrollment entities are in a one-to-many relationship. You'll use separate queries for Enrollment entities and their related Student entities.

Create a view model for the Instructor Index view

The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

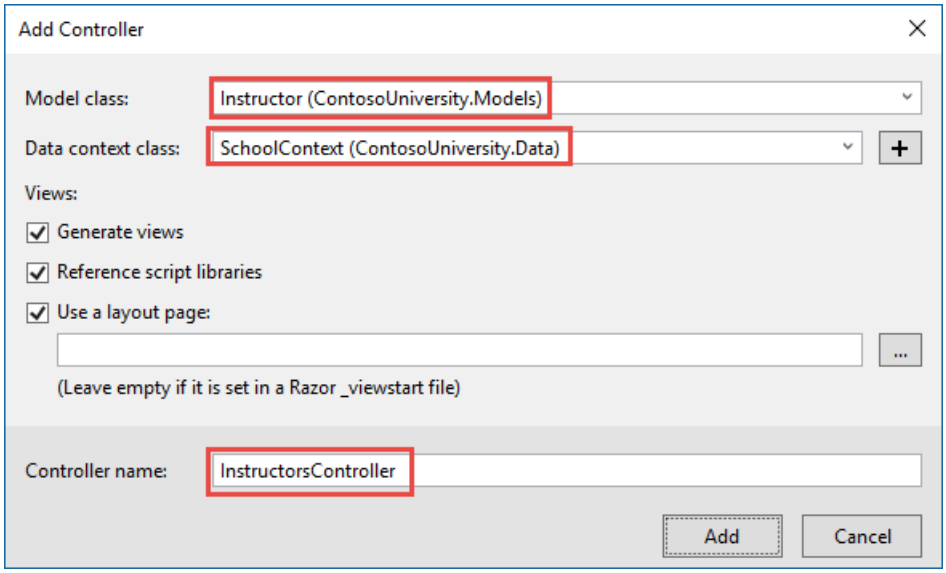
In the *SchoolViewModels* folder, create *InstructorIndexData.cs* and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:



Open *InstructorsController.cs* and add a using statement for the ViewModels namespace:

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Since the view always requires the OfficeAssignment entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `Enrollment.Student`.

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `Course.Enrollments`.

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the Course entities from that instructor's `CourseAssignments` navigation property.

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single Instructor entity being returned. The `Single` method converts the collection into a single Instructor entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The Single method throws an exception if the collection passed to it's empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the Where condition instead of calling the `Where` method separately:

```
.Single(i => i.ID == id.Value)
```

Instead of:

```
.Where(i => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Modify the Instructor Index view

In *Views/Instructors/Index.cshtml*, replace the template code with the following code. The changes are highlighted.

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
           }
    </tbody>
</table>
```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

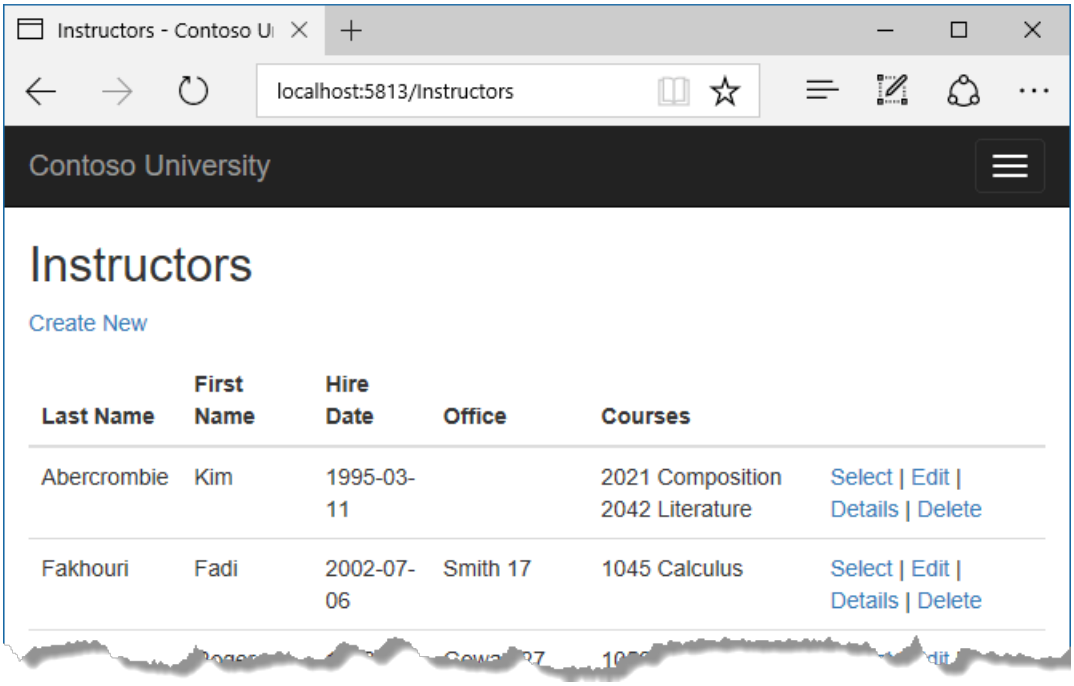
- Added a **Courses** column that displays courses taught by each instructor. For more information, see the Explicit line transition section of the Razor syntax article.
- Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Added a new hyperlink labeled **Select** immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.



In the *Views/Instructors/Index.cshtml* file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

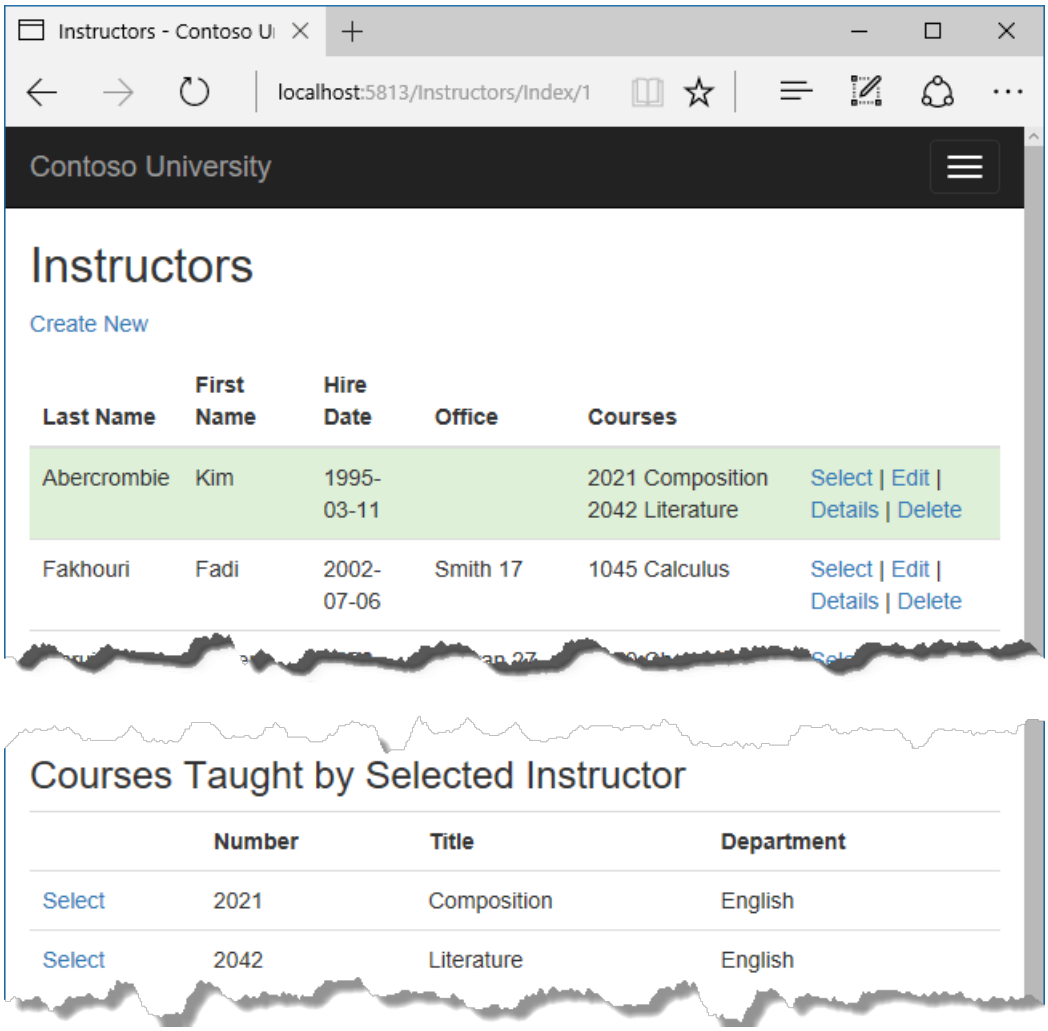
```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a **Select** hyperlink that sends the ID of the selected course to the `Index` action method.

Refresh the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



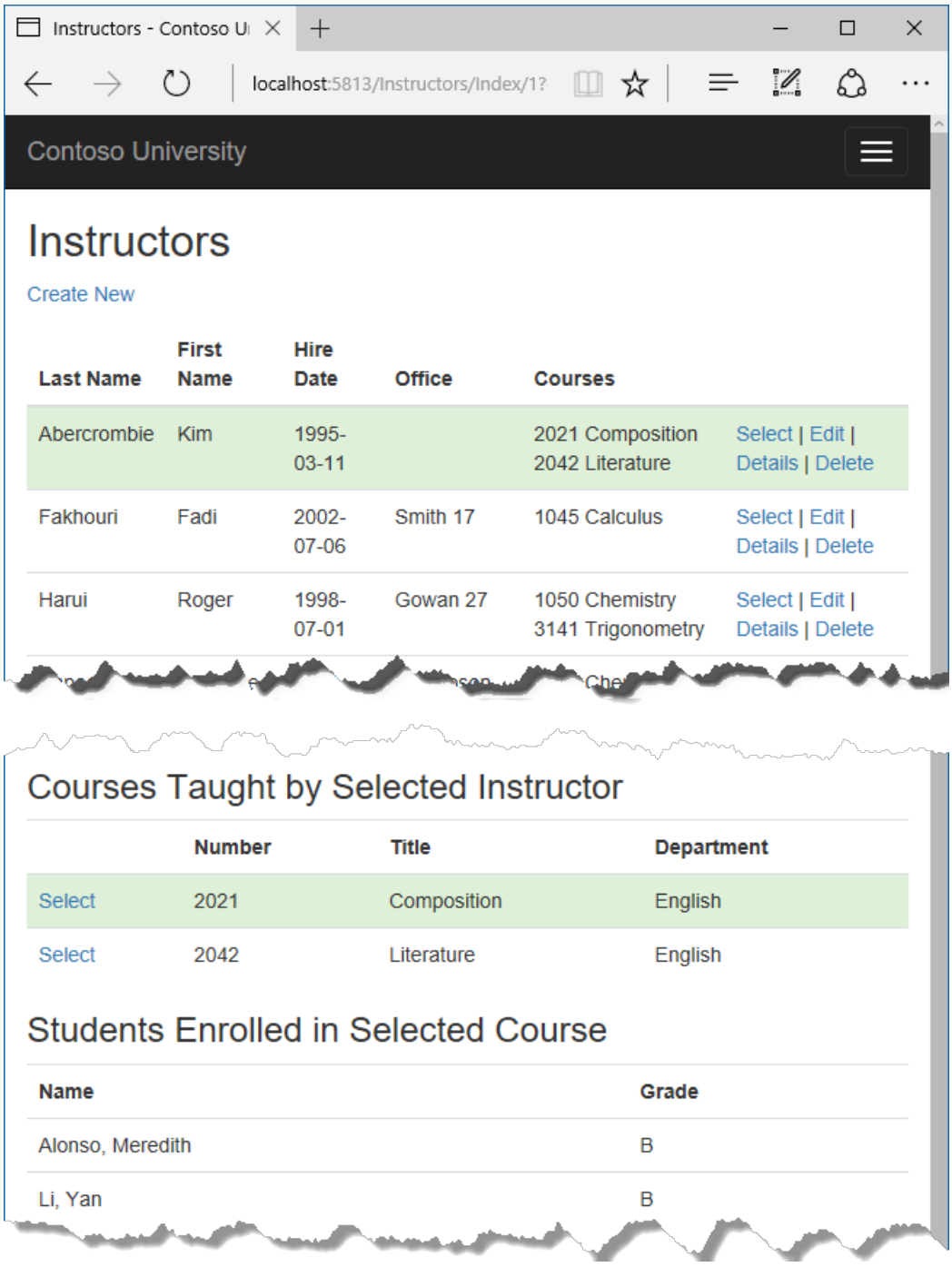
After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the Enrollments property of the view model in order to display a list of students enrolled in the course.

Refresh the page again and select an instructor. Then select a course to see the list of enrolled students and their

grades.



About explicit loading

When you retrieved the list of instructors in *InstructorsController.cs*, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for Enrollments and loads that property explicitly. The code changes are highlighted.

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

The new code drops the *ThenInclude* method calls for enrollment data from the code that retrieves instructor entities. It also drops `AsNoTracking`. If an instructor and course are selected, the highlighted code retrieves Enrollment entities for the selected course, and Student entities for each Enrollment.

Run the app, go to the Instructors Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Learned how to load related data
- Created a Courses page
- Created an Instructors page
- Learned about explicit loading

Advance to the next tutorial to learn how to update related data.

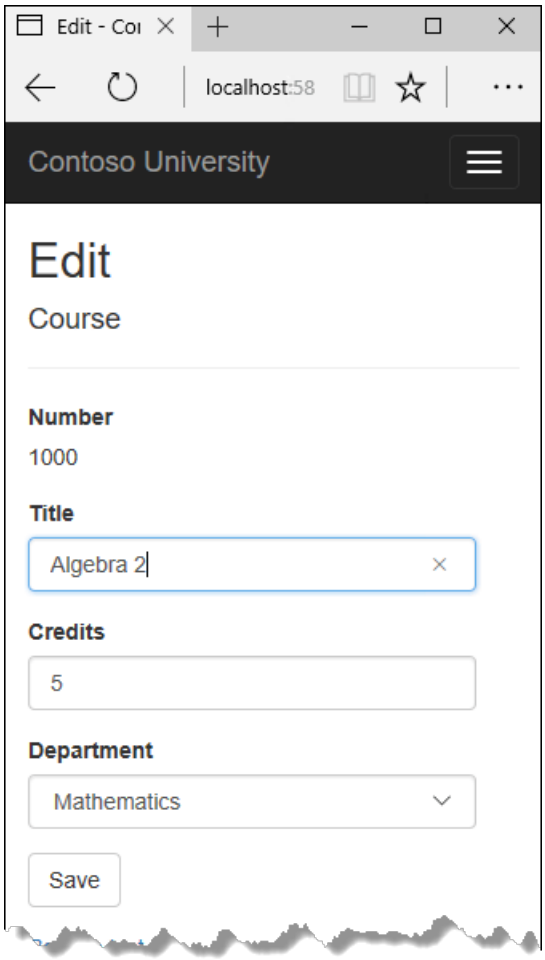
Update related data

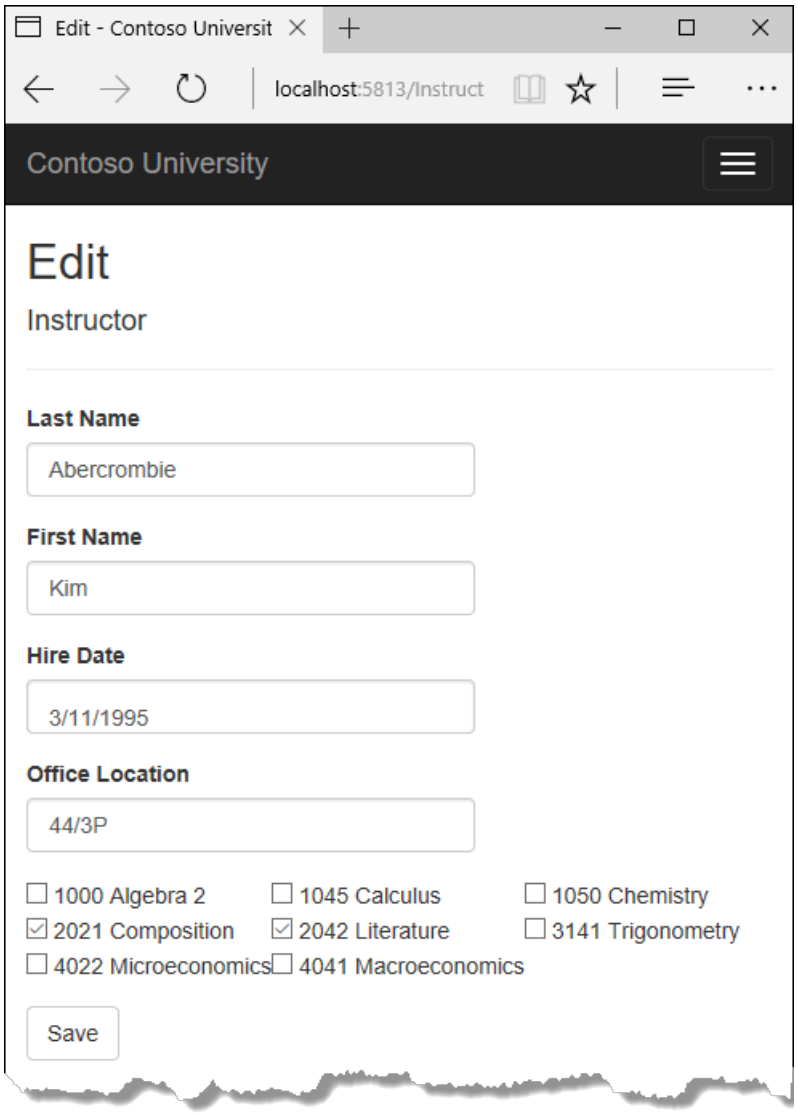
Tutorial: Update related data - ASP.NET MVC with EF Core

9/22/2020 • 18 minutes to read • Edit Online

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.





In this tutorial, you:

- Customize Courses pages
- Add Instructors Edit page
- Add courses to Edit page
- Update Delete page
- Add office location and courses to Create page

Prerequisites

- Read related data

Customize Courses pages

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate Department entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In *CoursesController.cs*, delete the four Create and Edit methods and replace them with the following code:

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .FirstOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}
```

After the `Edit` HttpPost method, create a new method that loads department info for the drop-down list.

```
private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(), "DepartmentID", "Name",
selectedDepartment);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The HttpGet `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department isn't established yet:

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The HttpGet `Edit` method sets the selected item, based on the ID of the department that's already assigned to the

course being edited:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The HttpPost methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add .AsNoTracking to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and HttpGet `Delete` methods.

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

Modify the Course views

In *Views/Courses/Create.cshtml*, add a "Select Department" option to the **Department** drop-down list, change the caption from **DepartmentID** to **Department**, and add a validation message.

```
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />
```

In *Views/Courses/Edit.cshtml*, make the same change for the Department field that you just did in *Create.cshtml*.

Also in *Views/Courses/Edit.cshtml*, add a course number field before the **Title** field. Because the course number is the primary key, it's displayed, but it can't be changed.

```
<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>
```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the **Edit** page.

In *Views/Courses/Delete.cshtml*, add a course number field at the top and change department ID to department name.

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

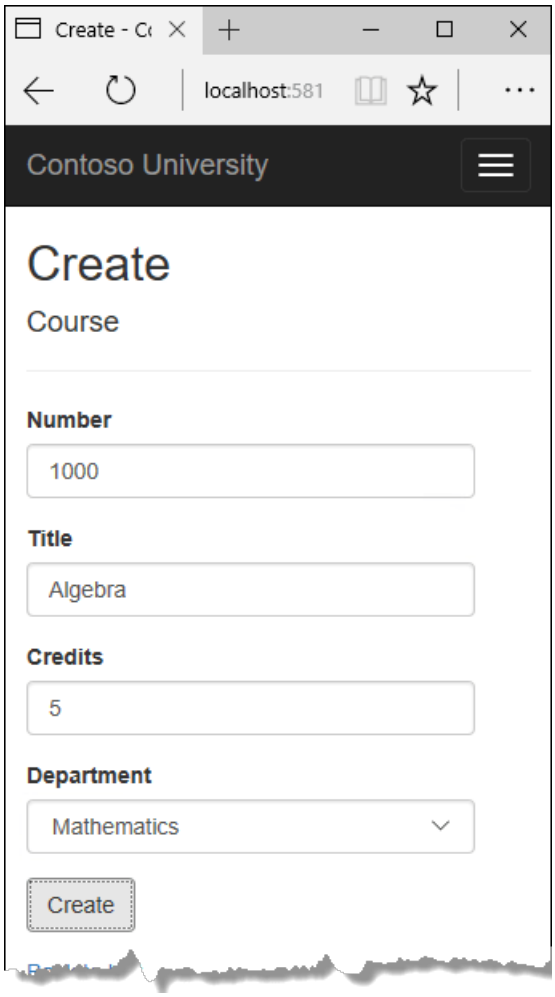
<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

In *Views/Courses/Details.cshtml*, make the same change that you just did for *Delete.cshtml*.

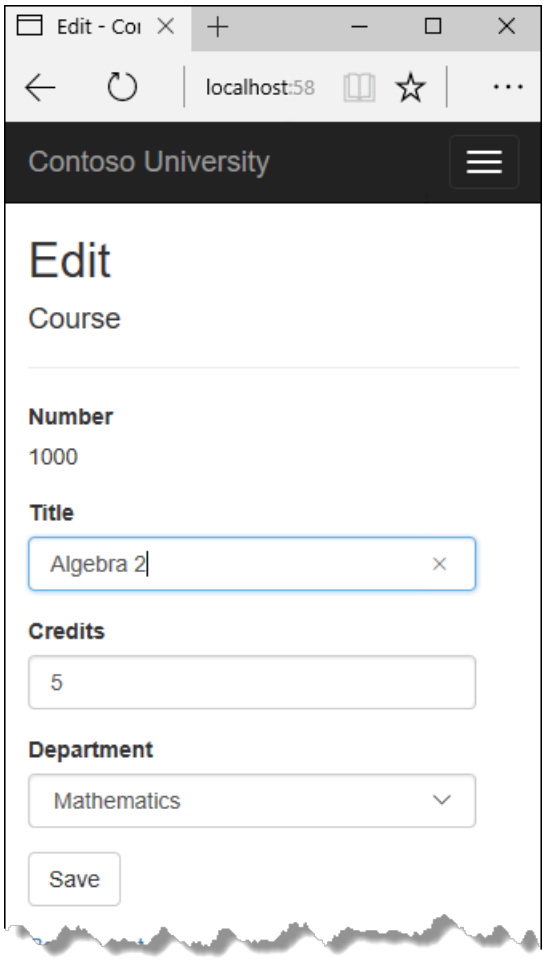
Test the Course pages

Run the app, select the **Courses** tab, click **Create New**, and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Click **Edit** on a course in the Courses Index page.



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add Instructors Edit page

When you edit an instructor record, you want to be able to update the instructor's office assignment. The Instructor entity has a one-to-zero-or-one relationship with the OfficeAssignment entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the OfficeAssignment entity.
- If the user enters an office assignment value and it originally was empty, create a new OfficeAssignment entity.
- If the user changes the value of an office assignment, change the value in an existing OfficeAssignment entity.

Update the Instructors controller

In *InstructorsController.cs*, change the code in the HttpGet `Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Replace the HttpPost `Edit` method with the following code to handle office assignment updates:

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .FirstOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}
```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the HttpGet `Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current Instructor entity from the database using eager loading for the `OfficeAssignment` navigation

property. This is the same as what you did in the HttpGet `Edit` method.

- Updates the retrieved Instructor entity with values from the model binder. The `TryUpdateModel` overload enables you to declare the properties you want to include. This prevents over-posting, as explained in the second tutorial.

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

- If the office location is blank, sets the Instructor.OfficeAssignment property to null so that the related row in the OfficeAssignment table will be deleted.

```
if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

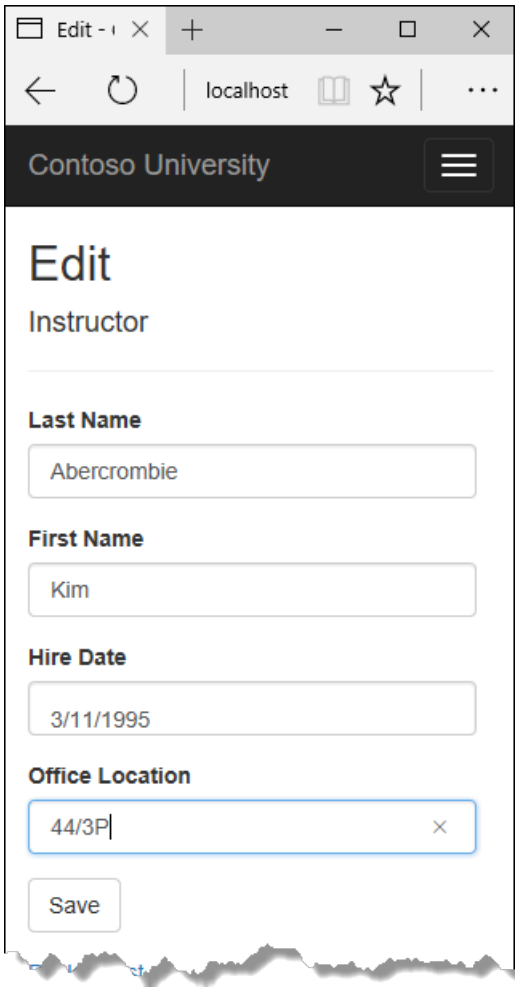
- Saves the changes to the database.

Update the Instructor Edit view

In *Views/Instructors/Edit.cshtml*, add a new field for editing the office location, at the end before the **Save** button:

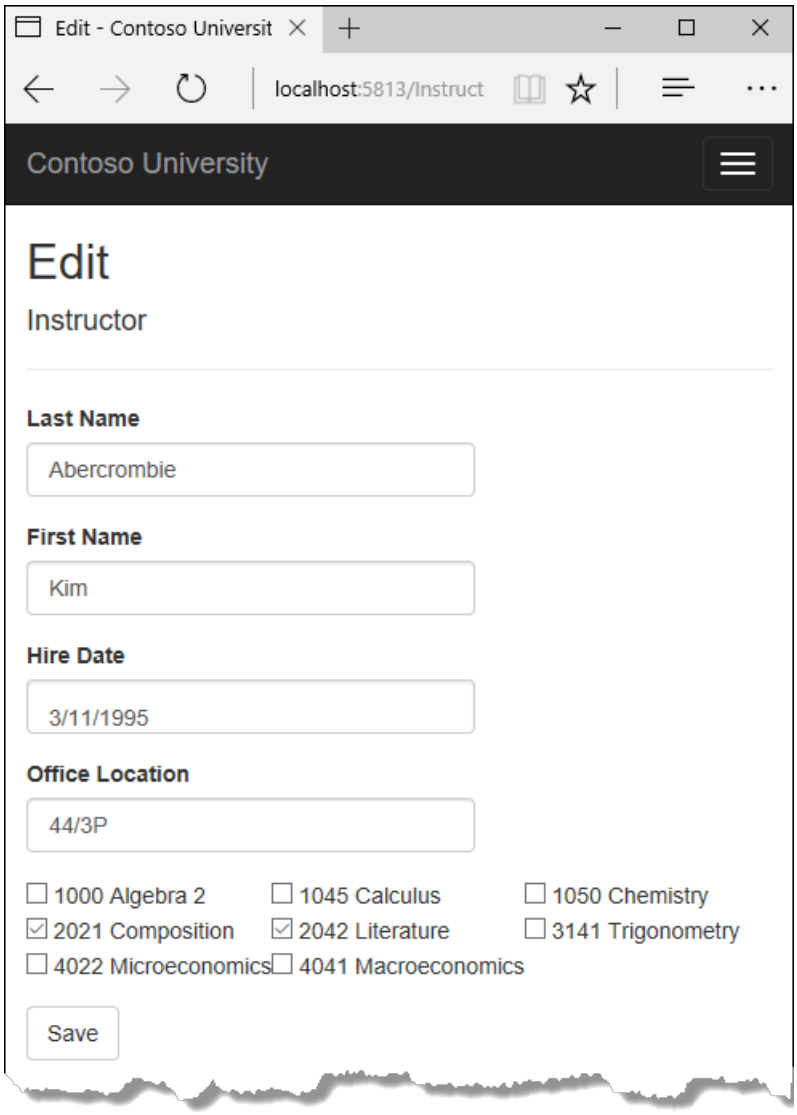
```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>
```

Run the app, select the **Instructors** tab, and then click **Edit** on an instructor. Change the **Office Location** and click **Save**.



Add courses to Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:



The relationship between the Course and Instructor entities is many-to-many. To add and remove relationships, you add and remove entities to and from the CourseAssignments join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of check boxes, you'll use a view model class.

Create *AssignedCourseData.cs* in the *SchoolViewModels* folder and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In *InstructorsController.cs*, replace the HttpGet `Edit` method with the following code. The changes are highlighted.

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all Course entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set to true for courses the instructor is assigned to. The view will use this property to determine which check boxes must

be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
        .FirstOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    UpdateInstructorCourses(selectedCourses, instructorToUpdate);
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}
```

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The method signature is now different from the HttpGet `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore, you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload that requires explicit approval and `CourseAssignments` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

Update the Instructor views

In *Views/Instructors/Edit.cshtml*, add a **Courses** field with an array of check boxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

> **NOTE**
>
> When you paste the code in Visual Studio, line breaks might be changed in a way that breaks the code. If the code looks different after pasting, press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab three times to line up the new code with the existing code. This problem is fixed in Visual Studio 2019.

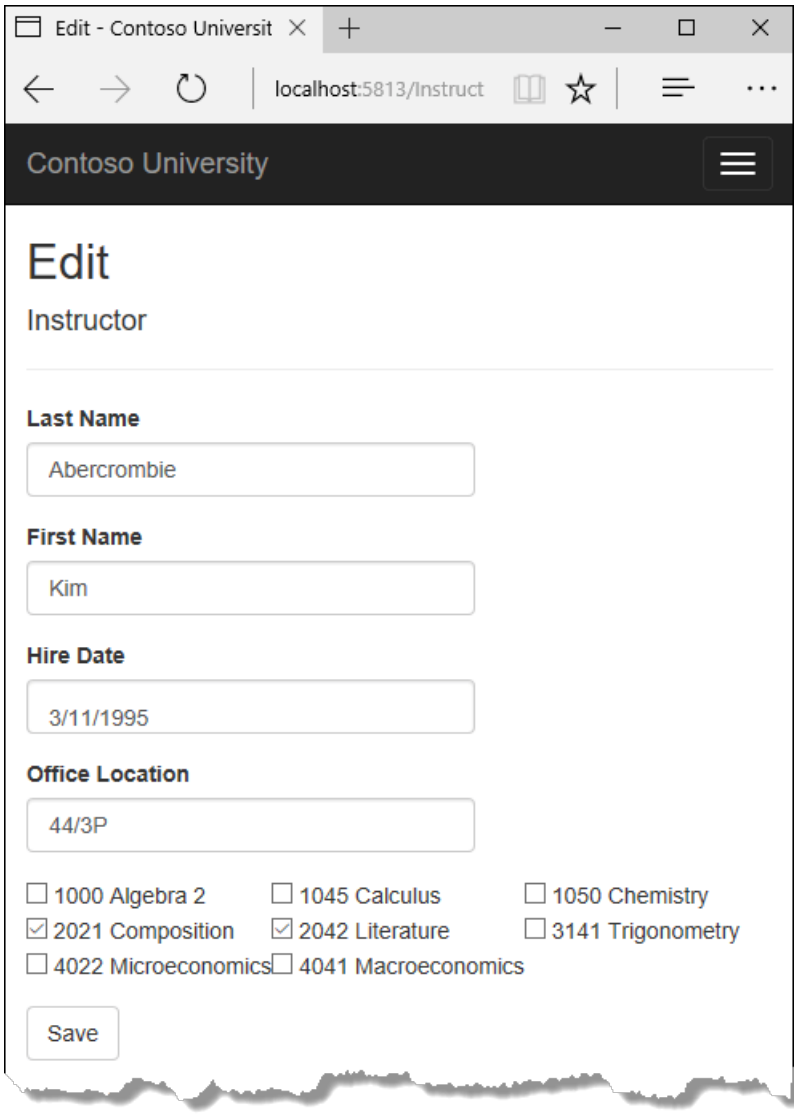
```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they're to be treated as a group. The value attribute of each check box is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the app, select the **Instructors** tab, and click **Edit** on an instructor to see the **Edit** page.



Change some course assignments and click Save. The changes you make are reflected on the Index page.

> **NOTE**
> The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update Delete page

In *InstructorsController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to Create page

In *InstructorsController.cs*, delete the HttpGet and HttpPost `Create` methods, and then add the following code in their place:

```
public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}

// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID = int.Parse(course)
};
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The HttpGet `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost `Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the Instructor model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In *Views/Instructor/Create.cshtml*, add an office location text box and check boxes for courses before the Submit button. As in the case of the Edit page, fix the formatting if Visual Studio reformats the code when you paste it.

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                            @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

Test by running the app and creating an instructor.

Handling Transactions

As explained in the CRUD tutorial, the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Customized Courses pages
- Added Instructors Edit page
- Added courses to Edit page
- Updated Delete page
- Added office location and courses to Create page

Advance to the next tutorial to learn how to handle concurrency conflicts.

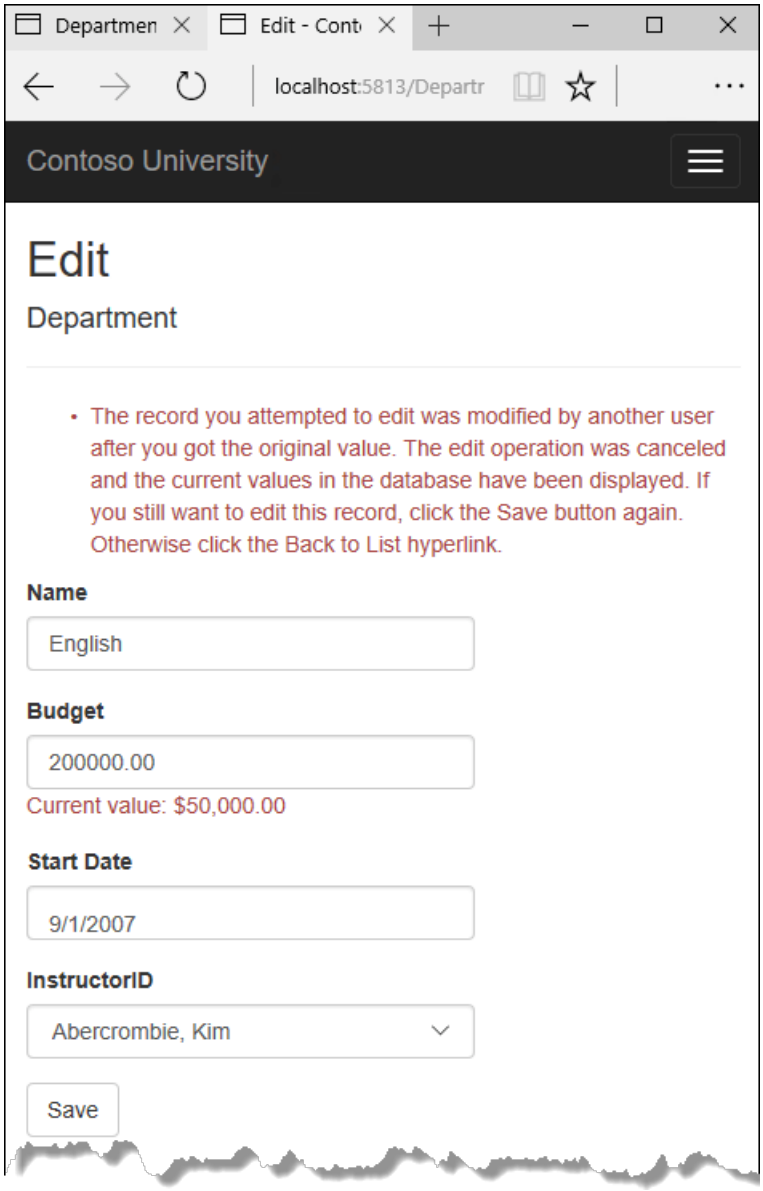
Handle concurrency conflicts

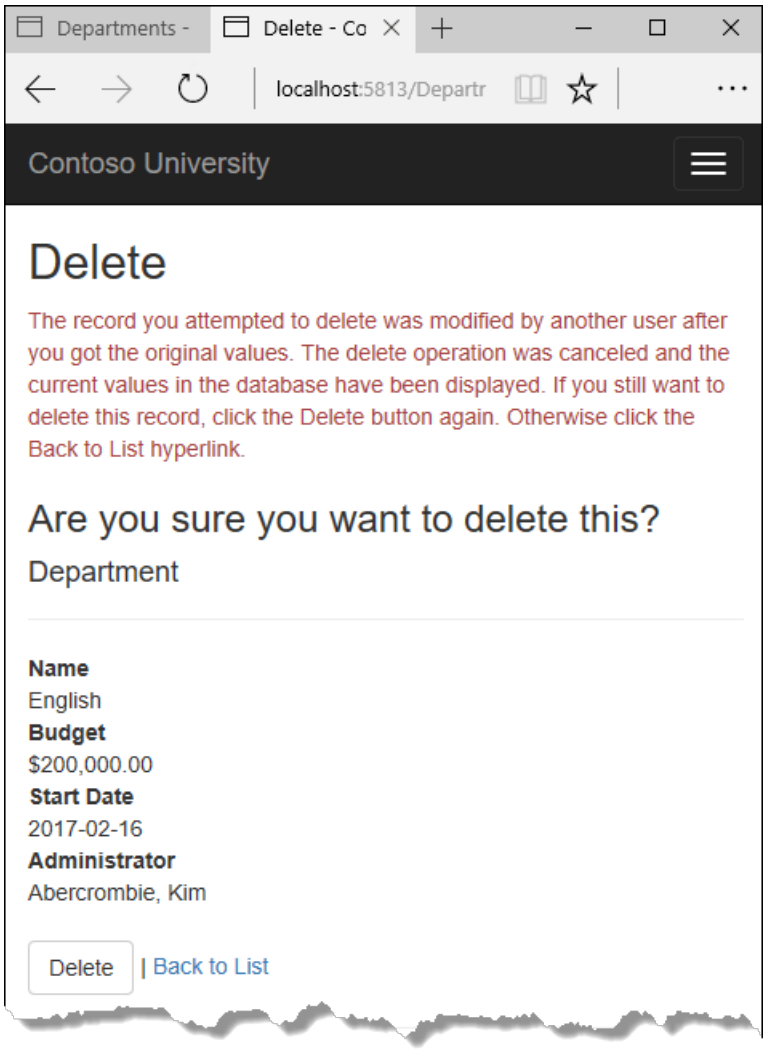
Tutorial: Handle concurrency - ASP.NET MVC with EF Core

9/22/2020 • 18 minutes to read • Edit Online

In earlier tutorials, you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the Department entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.





In this tutorial, you:

- Learn about concurrency conflicts
- Add a tracking property
- Create Departments controller and views
- Update Index view
- Update Edit methods
- Update Edit view
- Test concurrency conflicts
- Update the Delete page
- Update Details and Create views

Prerequisites

- Update related data

Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

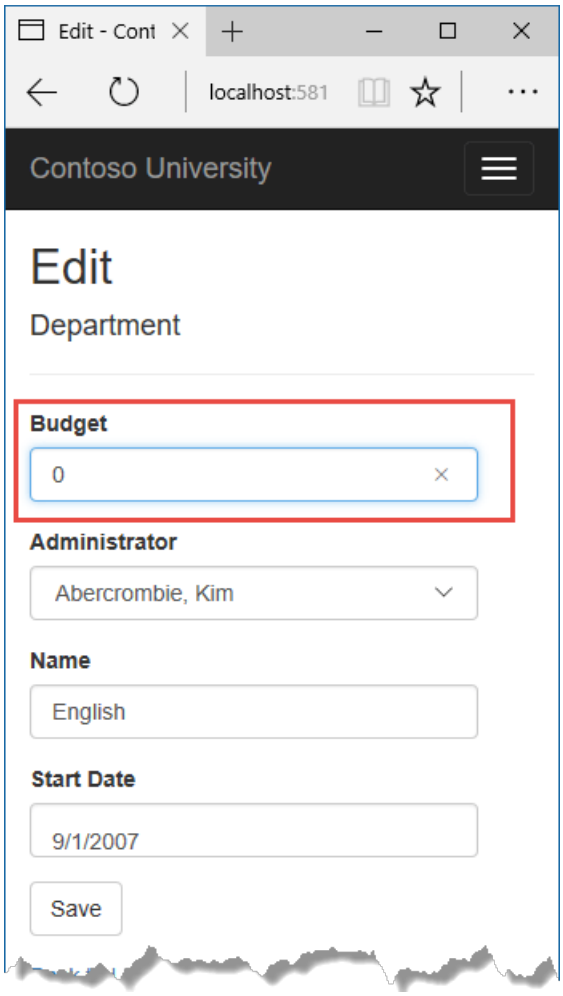
Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

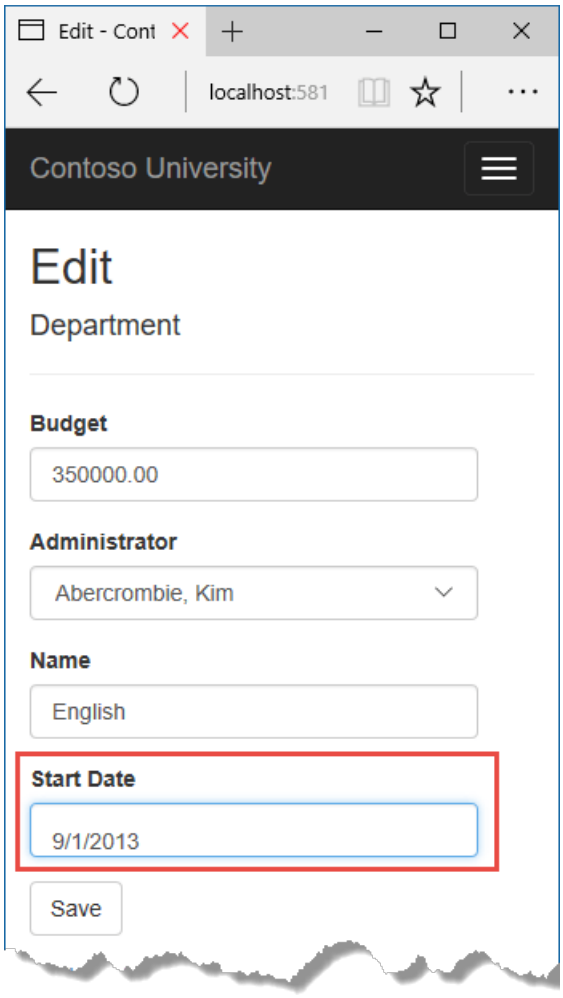
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

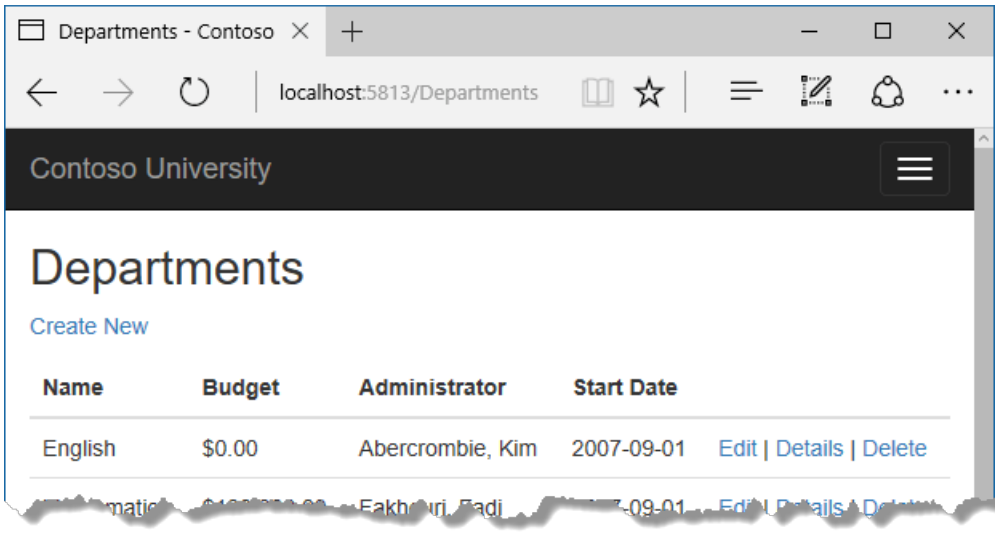
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



Then John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

  In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same

property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent John's change from being updated in the database.

  Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

  The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version) . If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

  As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

  If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

Add a tracking property

In *Models/Department.cs*, add a tracking property named RowVersion:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsConcurrencyToken` method (in *Data/SchoolContext.cs*) to specify the tracking property, as shown in the following example:

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

By adding a property you changed the database model, so you need to do another migration.

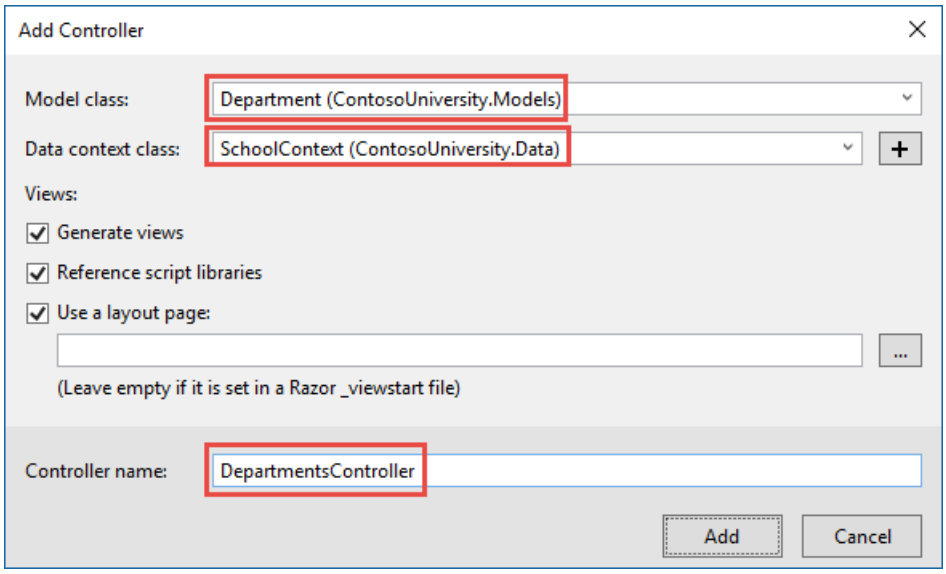
Save your changes and build the project, and then enter the following commands in the command window:

```
dotnet ef migrations add RowVersion
```

```
dotnet ef database update
```

Create Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



In the *DepartmentsController.cs* file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", department.InstructorID);
```

Update Index view

The scaffolding engine created a RowVersion column in the Index view, but that field shouldn't be displayed.

Replace the code in *Views/Departments/Index.cshtml* with the following code.

```
@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

This changes the heading to "Departments", deletes the RowVersion column, and shows full name instead of first name for the administrator.

Update Edit methods

In both the HttpGet `Edit` method and the `Details` method, add `AsNoTracking`. In the HttpGet `Edit` method, add eager loading for the Administrator.

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .FirstOrDefaultAsync(m => m.DepartmentID == id);
```

Replace the existing code for the HttpPost `Edit` method with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i => i.Administrator).FirstOrDefaultAsync(m =>
m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value: {databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {
```

```
                    ModelState.AddModelError("StartDate", $"Current value: {databaseValues.StartDate:d}");
                }
                if (databaseValues.InstructorID != clientValues.InstructorID)
                {
                    Instructor databaseInstructor = await _context.Instructors.FirstOrDefaultAsync(i => i.ID ==
databaseValues.InstructorID);
                    ModelState.AddModelError("InstructorID", $"Current value: {databaseInstructor?.FullName}");
                }

                ModelState.AddModelError(string.Empty, "The record you attempted to edit "
                        + "was modified by another user after you got the original value. The "
                        + "edit operation was canceled and the current values in the database "
                        + "have been displayed. If you still want to edit this record, click "
                        + "the Save button again. Otherwise click the Back to List hyperlink.");
                departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
                ModelState.Remove("RowVersion");
            }
        }
    }
    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}
```

The code begins by trying to read the department to be updated. If the `FirstOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a department entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the department entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;
```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

```
var exceptionEntry = ex.Entries.Single();
```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update Edit view

In *Views/Departments/Edit.cshtml*, make the following changes:

- Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.
- Add a "Select Administrator" option to the drop-down list.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

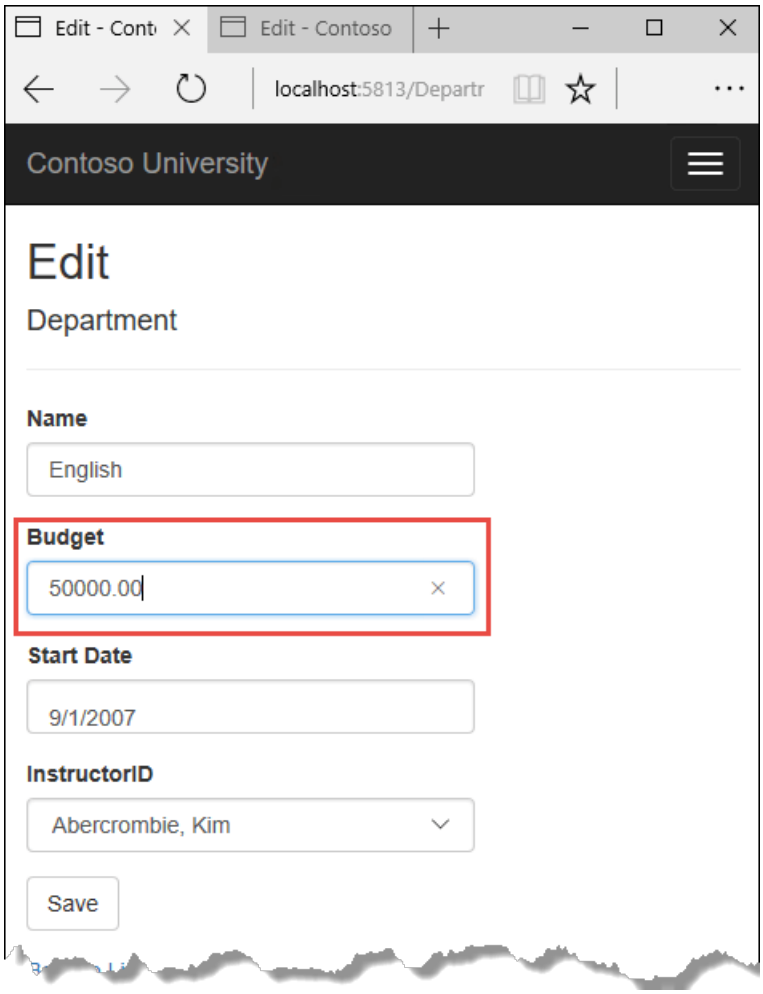
<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test concurrency conflicts

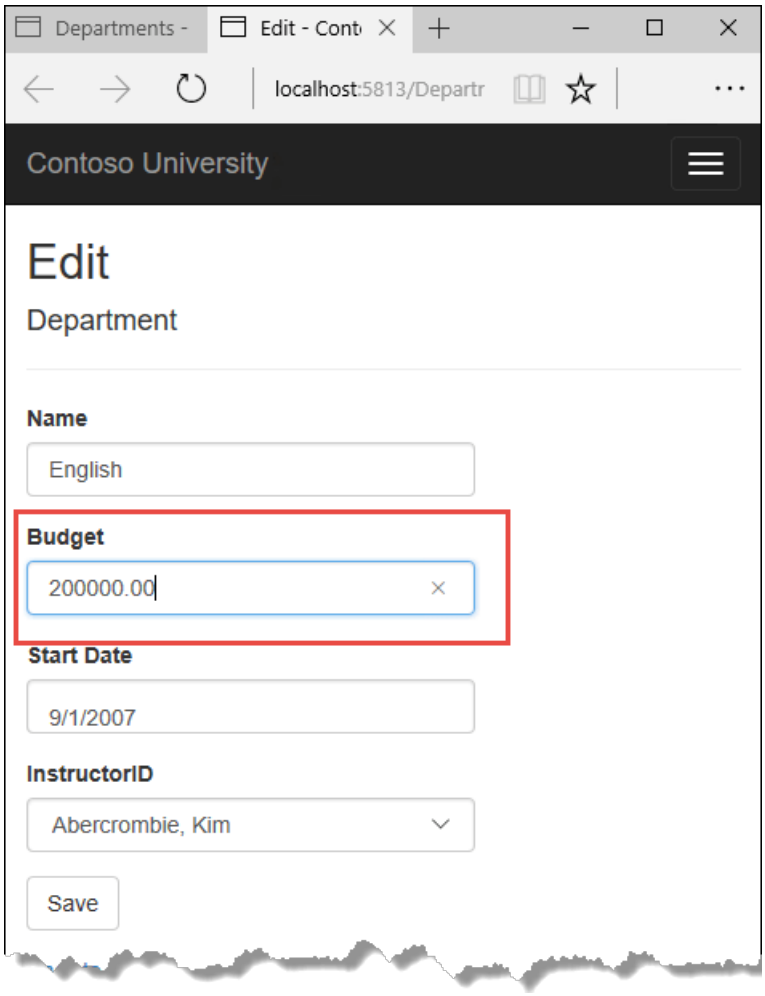
Run the app and go to the Departments Index page. Right-click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.

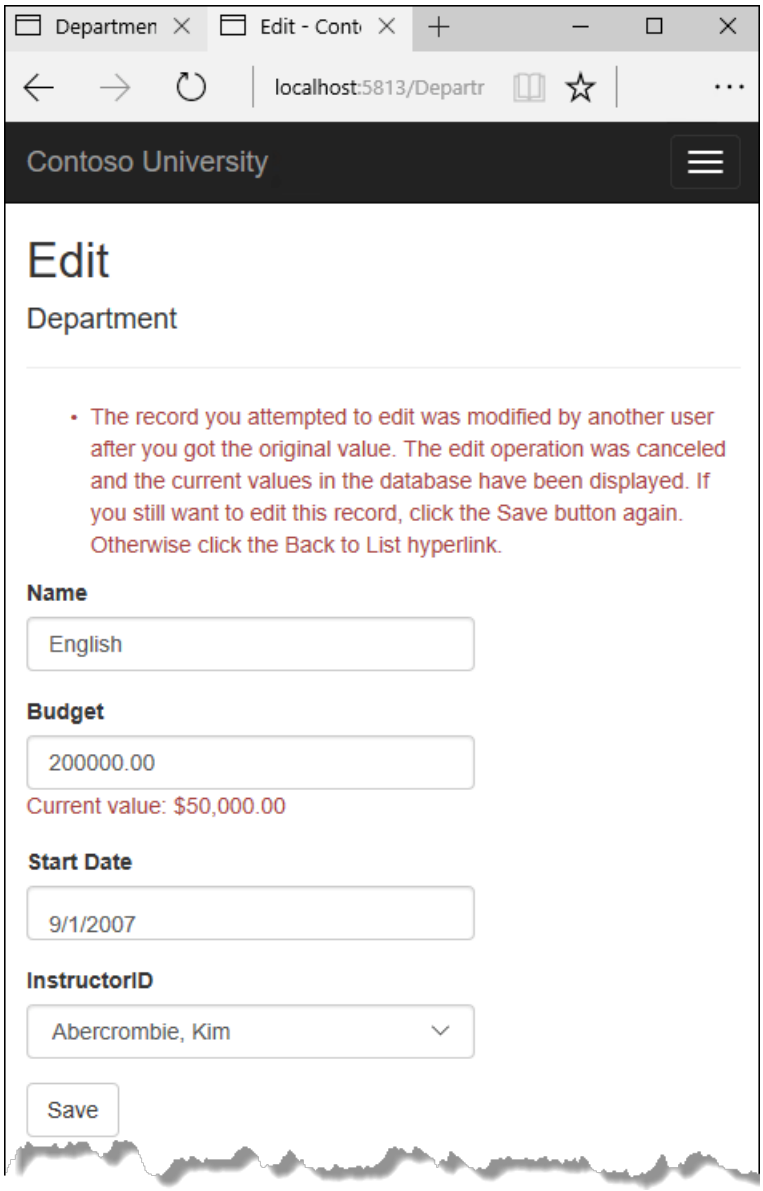


The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the HttpGet `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the HttpPost `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the HttpGet `Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In *DepartmentsController.cs*, replace the HttpGet `Delete` method with the following code:

```
public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

    return View(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the Department does exist, it was changed by another user. In that case, the code sends an error message to the view using `ViewData`.

Replace the code in the HttpPost `Delete` method (named `DeleteConfirmed`) with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID == department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError = true, id = department.DepartmentID });
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a Department entity instance created by the model binder. This gives EF access to the RowVersion property value in addition to the record key.

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In *Views/Departments/Delete.cshtml*, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

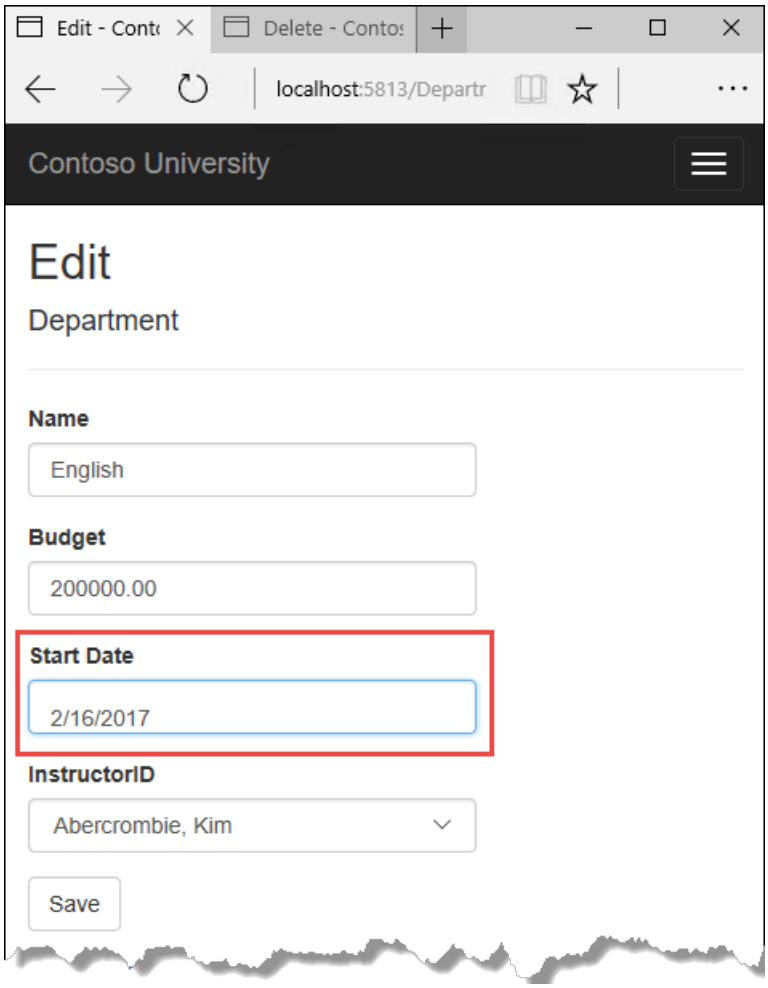
    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

This makes the following changes:

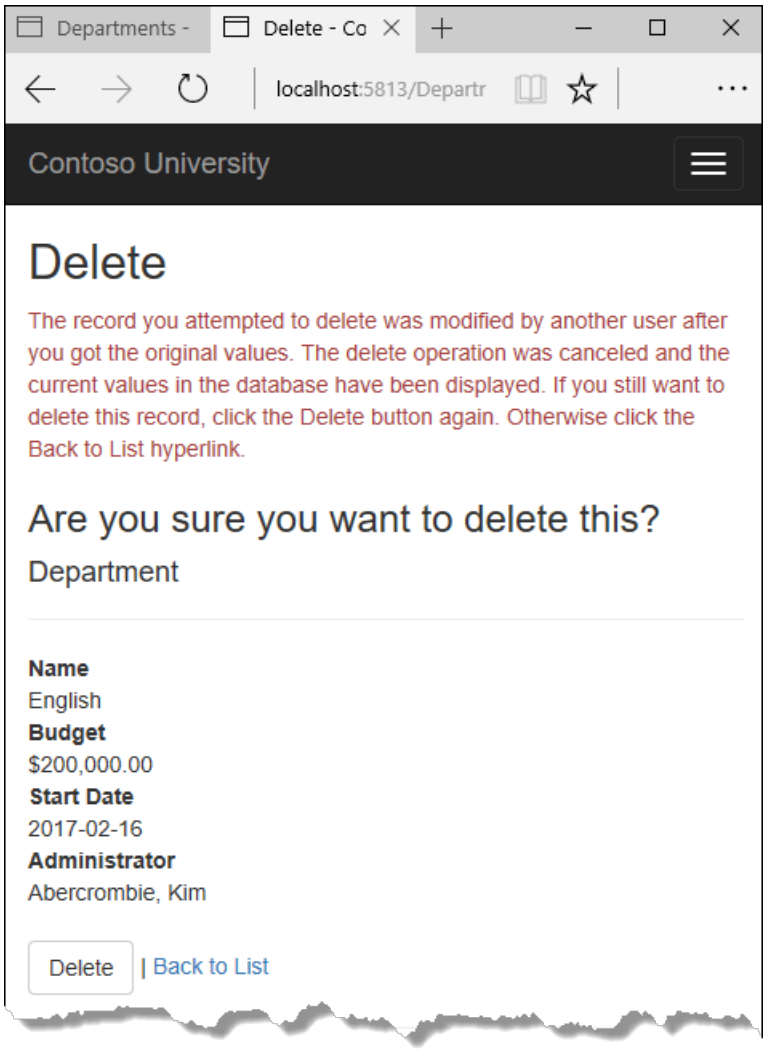
- Adds an error message between the `h2` and `h3` headings.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Removes the RowVersion field.
- Adds a hidden field for the `RowVersion` property.

Run the app and go to the Departments Index page. Right-click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in *Views/Departments/Details.cshtml* to delete the RowVersion column and show the full name of the Administrator.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Replace the code in *Views/Departments/Create.cshtml* to add a Select option to the drop-down list.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Get the code

Download or view the completed application.

Additional resources

For more information about how to handle concurrency in EF Core, see Concurrency conflicts.

Next steps

In this tutorial, you:

- Learned about concurrency conflicts

- Added a tracking property
- Created Departments controller and views
- Updated Index view
- Updated Edit methods
- Updated Edit view
- Tested concurrency conflicts
- Updated the Delete page
- Updated Details and Create views

Advance to the next tutorial to learn how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

Next: Implement table-per-hierarchy inheritance

Tutorial: Implement inheritance - ASP.NET MVC with EF Core

9/22/2020 • 8 minutes to read • Edit Online

In the previous tutorial, you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

In this tutorial, you:

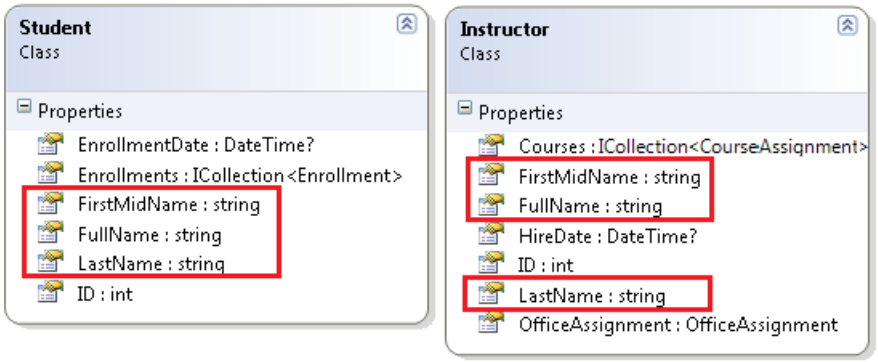
- Map inheritance to database
- Create the Person class
- Update Instructor and Student
- Add Person to the model
- Create and update migrations
- Test the implementation

Prerequisites

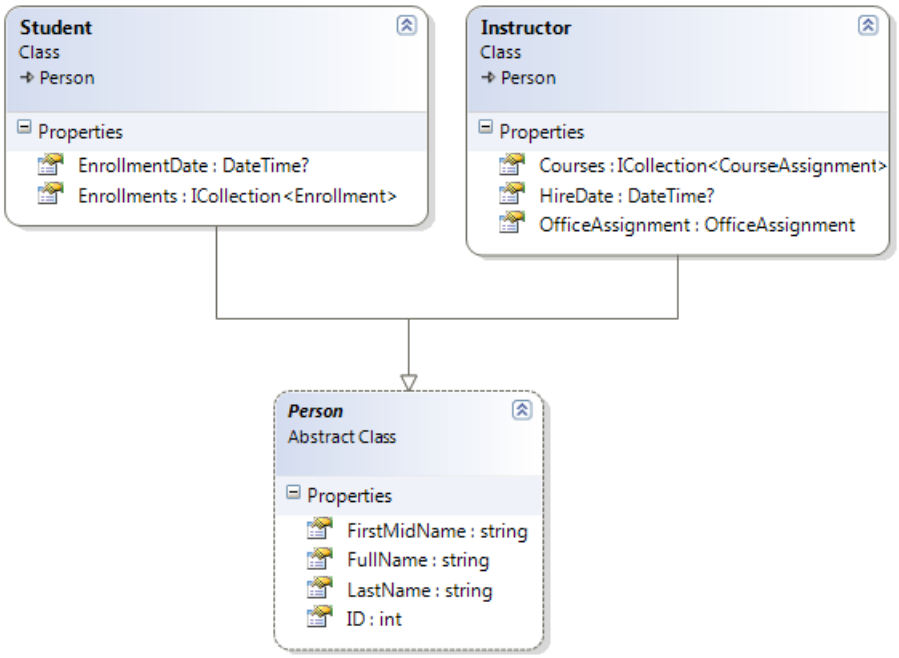
- Handle Concurrency

Map inheritance to database

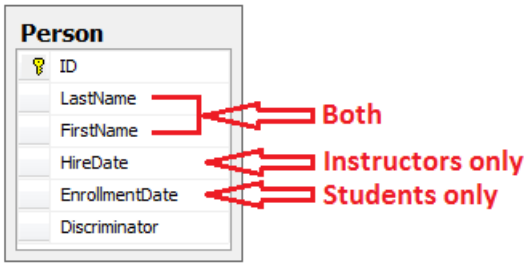
The `Instructor` and `Student` classes in the School data model have several properties that are identical:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:



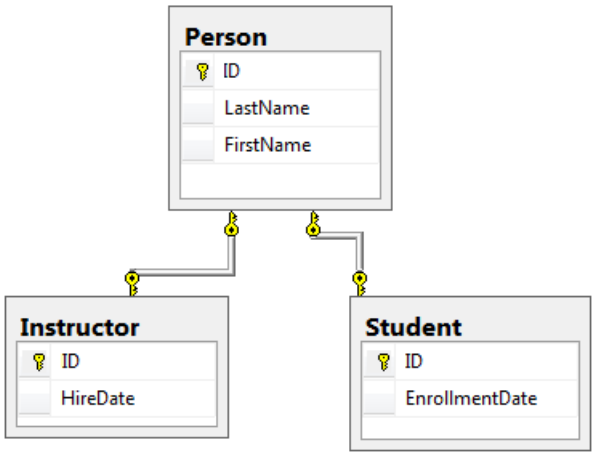
There are several ways this inheritance structure could be represented in the database. You could have a Person table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called table-per-hierarchy (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the Person table and have separate Instructor and Student tables with the date fields.

> **WARNING**
> Table Per Type (TPT) is not supported by EF Core 3.x, however it is has been implemented in EF Core 5.0.



This pattern of making a database table for each entity class is called table per type (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the Person, Student, and Instructor classes as shown earlier, the Student and Instructor tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

> **TIP**
> Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

Create the Person class

In the Models folder, create Person.cs and replace the template code with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

Update Instructor and Student

In *Instructor.cs*, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Make the same changes in *Student.cs*.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }


        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Add Person to the model

Add the Person entity type to *SchoolContext.cs*. The new lines are highlighted.

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and update migrations

Save your changes and build the project. Then open the command window in the project folder and enter the following command:

```
dotnet ef migrations add Inheritance
```

Don't run the `database update` command yet. That command will result in lost data because it will drop the Instructor table and rename the Student table to Person. You need to provide custom code to preserve existing data.

Open *Migrations/<timestamp>_Inheritance.cs* and replace the `Up` method with the following code:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table: "Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person", nullable: false, maxLength: 128,
defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate, Discriminator,
OldId) SELECT LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, ID AS OldId
FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId =
Enrollment.StudentId AND Discriminator = 'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
         name: "IX_Enrollment_StudentID",
         table: "Enrollment",
         column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
- Adds nullable EnrollmentDate for students.

- Adds Discriminator column to indicate whether a row is for a student or an instructor.
- Makes HireDate nullable since student rows won't have hire dates.
- Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they will get new primary key values.
- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command:

```
dotnet ef database update
```

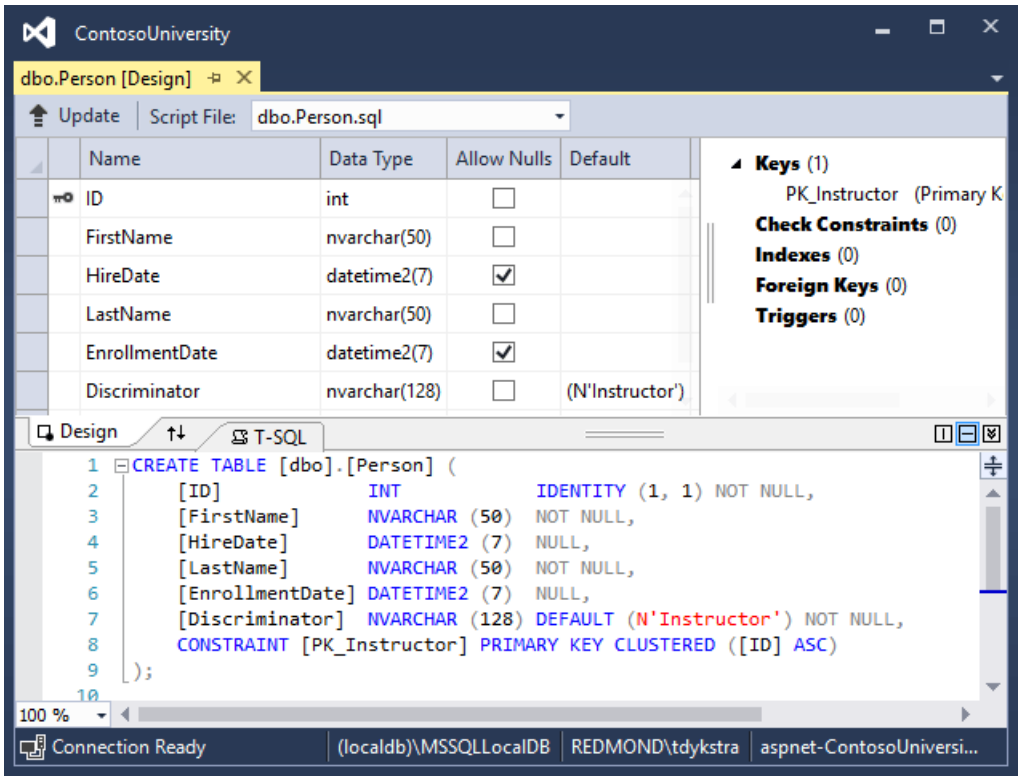
(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

> **NOTE**
> It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `database drop` CLI command.

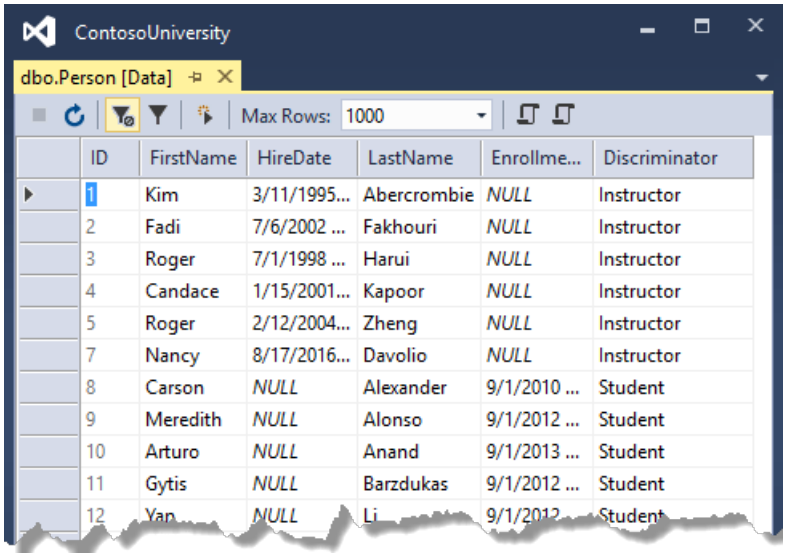
Test the implementation

Run the app and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.



Get the code

Download or view the completed application.

Additional resources

For more information about inheritance in Entity Framework Core, see Inheritance.

Next steps

In this tutorial, you:

- Mapped inheritance to database
- Created the Person class
- Updated Instructor and Student
- Added Person to the model

- Created and update migrations
- Tested the implementation

Advance to the next tutorial to learn how to handle a variety of relatively advanced Entity Framework scenarios.

Next: Advanced topics

Tutorial: Learn about advanced scenarios - ASP.NET MVC with EF Core

9/22/2020 • 13 minutes to read • Edit Online

In the previous tutorial, you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET Core web applications that use Entity Framework Core.

In this tutorial, you:

- Perform raw SQL queries
- Call a query to return entities
- Call a query to return other types
- Call an update query
- Examine SQL queries
- Create an abstraction layer
- Learn about Automatic change detection
- Learn about EF Core source code and development plans
- Learn how to use dynamic LINQ to simplify code

Prerequisites

- Implement Inheritance

Perform raw SQL queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

- Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they're automatically tracked by the database context unless you turn tracking off.
- Use the `Database.ExecuteSqlCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Call a query to return entities

The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type

`TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In *DepartmentsController.cs*, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

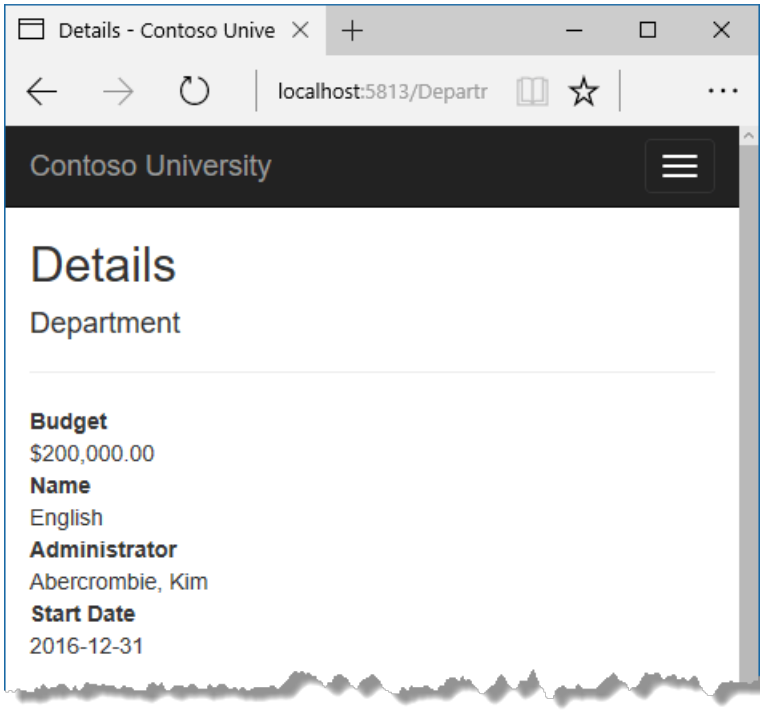
```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Call a query to return other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is write ADO.NET code and get the database connection from EF.

In *HomeController.cs*, replace the `About` method with the following code:

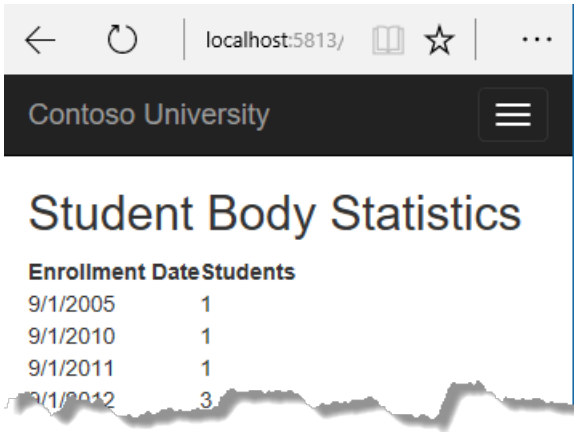
```
public async Task<ActionResult> About()
{
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate = reader.GetDateTime(0), StudentCount =
reader.GetInt32(1) };
                    groups.Add(row);
                }
            }
            reader.Dispose();
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}
```

Add a using statement:

```
using System.Data.Common;
```

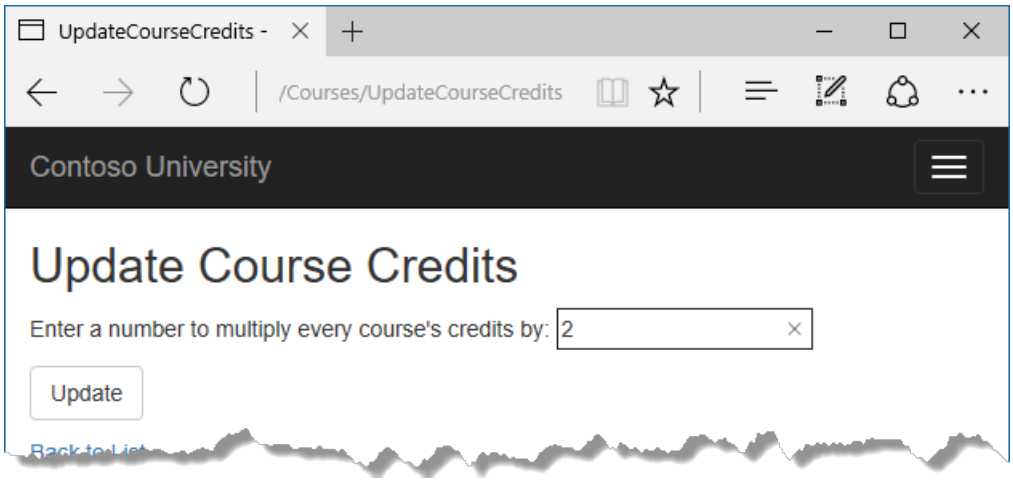
Run the app and go to the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by

executing a SQL UPDATE statement. The web page will look like the following illustration:



In *CoursesController.cs*, add UpdateCourseCredits methods for HttpGet and HttpPost:

```
public IActionResult UpdateCourseCredits()
{
    return View();
}
```

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

When the controller processes an HttpGet request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the HttpPost method is called, and multiplier has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

In **Solution Explorer**, right-click the *Views/Courses* folder, and then click **Add > New Item**.

In the **Add New Item** dialog, click **ASP.NET Core** under **Installed** in the left pane, click **Razor View**, and name the new view *UpdateCourseCredits.cshtml*.

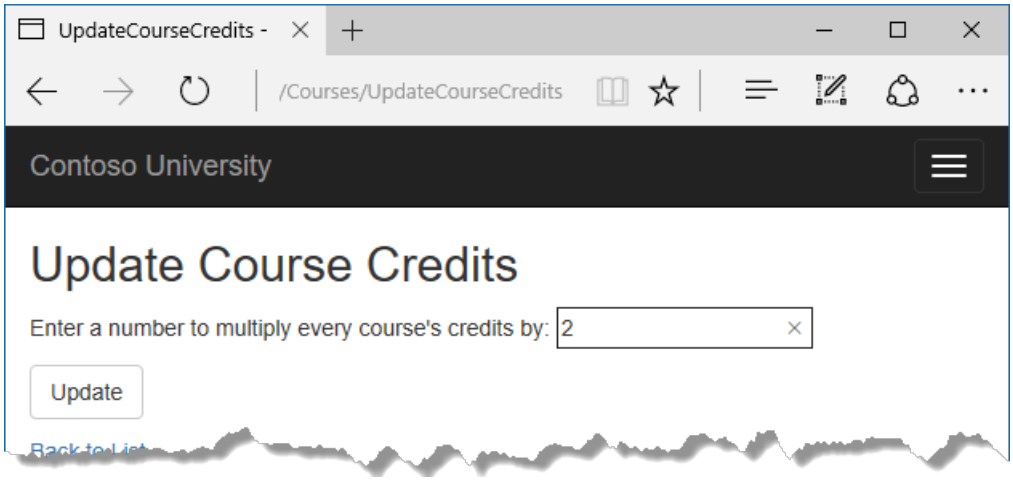
In *Views/Courses/UpdateCourseCredits.cshtml*, replace the template code with the following code:

```
@{
    ViewBag.Title = "UpdateCourseCredits";
}

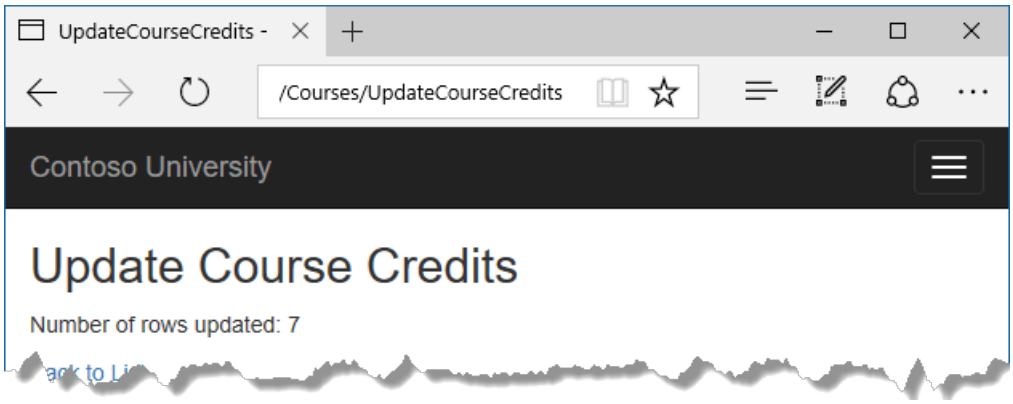
<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
            <p>
                Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default" />
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Courses/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see Raw SQL Queries.

Examine SQL queries

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open *StudentsController.cs* and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the app in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (56ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (122ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID], [s.Enrollments].[Grade], [s.Enrollments].
[StudentID], [e.Course].[CourseID], [e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] = [e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`) from the Person table. The `SingleOrDefaultAsync` method doesn't resolve to 1 row on the server. Here's why:

- If the query would return multiple rows, the method returns null.
- To determine whether the query would return multiple rows, EF has to check if it returns at least 2.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Create an abstraction layer

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns isn't always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series.

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see Test with InMemory.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- DbContext.SaveChanges
- DbContext.Entry
- ChangeTracker.Entries

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

EF Core source code and development plans

The Entity Framework Core source is at https://github.com/dotnet/efcore. The EF Core repository contains nightly builds, issue tracking, feature specs, design meeting notes, and the roadmap for future development. You can file or find bugs, and contribute.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the scaffold-dbcontext command. See the getting-started tutorial.

Use dynamic LINQ to simplify code

The third tutorial in this series shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] =
        String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
    ViewData["DateSortParm"] =
        sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" : "EnrollmentDate";

    if (searchString != null)
    {
        pageNumber = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }

    if (string.IsNullOrEmpty(sortOrder))
    {
        sortOrder = "LastName";
    }

    bool descending = false;
    if (sortOrder.EndsWith("_desc"))
    {
        sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
        descending = true;
    }

    if (descending)
    {
        students = students.OrderByDescending(e => EF.Property<object>(e, sortOrder));
    }
    else
    {
        students = students.OrderBy(e => EF.Property<object>(e, sortOrder));
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
        pageNumber ?? 1, pageSize));
}
```

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials. John Parente and Paul Goldman worked on updating the tutorial for ASP.NET Core 2.2.

Troubleshoot common errors

ContosoUniversity.dll used by another process

Error message:

> Cannot open '...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in *appsettings.json*. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the `database drop` CLI command:

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

> A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

Get the code

Download or view the completed application.

Additional resources

For more information about EF Core, see the Entity Framework Core documentation. A book is also available: Entity Framework Core in Action.

For information on how to deploy a web app, see Host and deploy ASP.NET Core.

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see Introduction to ASP.NET Core.

Next steps

In this tutorial, you:

- Performed raw SQL queries
- Called a query to return entities
- Called a query to return other types
- Called an update query
- Examined SQL queries
- Created an abstraction layer
- Learned about Automatic change detection
- Learned about EF Core source code and development plans
- Learned how to use dynamic LINQ to simplify code

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET Core MVC application. This series worked with a new database; an alternative is to reverse engineer a model from an existing database.

Tutorial: EF Core with MVC, existing database

ASP.NET Core fundamentals

9/22/2020 • 17 minutes to read • Edit Online

This article provides an overview of key topics for understanding how to develop ASP.NET Core apps.

The Startup class

The `Startup` class is where:

- Services required by the app are configured.
- The app's request handling pipeline is defined, as a series of middleware components.

Here's a sample `Startup` class:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<RazorPagesMovieContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));

        services.AddControllersWithViews();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute();
            endpoints.MapRazorPages();
        });
    }
}
```

For more information, see App startup in ASP.NET Core.

Dependency injection (services)

ASP.NET Core includes a built-in dependency injection (DI) framework that makes configured services available throughout an app. For example, a logging component is a service.

Code to configure (or *register*) services is added to the `Startup.ConfigureServices` method. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<RazorPagesMovieContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

Services are typically resolved from DI using constructor injection. With constructor injection, a class declares a constructor parameter of either the required type or an interface. The DI framework provides an instance of this service at runtime.

The following example uses constructor injection to resolve a `RazorPagesMovieContext` from DI:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovieContext context)
    {
        _context = context;
    }

    // ...

    public async Task OnGetAsync()
    {
        Movies = await _context.Movies.ToListAsync();
    }
}
```

If the built-in Inversion of Control (IoC) container doesn't meet all of an app's needs, a third-party IoC container can be used instead.

For more information, see Dependency injection in ASP.NET Core.

Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an `HttpContext` and either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline by invoking a `Use...` extension method in the `Startup.Configure` method. For example, to enable rendering of static files, call `UseStaticFiles`.

The following example configures a request handling pipeline:

```
public void Configure(IApplicationBuilder app)
{
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });
}
```

ASP.NET Core includes a rich set of built-in middleware. Custom middleware components can also be written.

For more information, see ASP.NET Core Middleware.

Host

On startup, an ASP.NET Core app builds a *host*. The host encapsulates all of the app's resources, such as:

- An HTTP server implementation
- Middleware components
- Logging
- Dependency injection (DI) services
- Configuration

There are two different hosts:

- .NET Generic Host
- ASP.NET Core Web Host

The .NET Generic Host is recommended. The ASP.NET Core Web Host is available only for backwards compatibility.

The following example creates a .NET Generic Host:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The `CreateDefaultBuilder` and `ConfigureWebHostDefaults` methods configure a host with a set of default options, such as:

- Use Kestrel as the web server and enable IIS integration.
- Load configuration from *appsettings.json*, *appsettings.{Environment Name}.json*, environment variables, command line arguments, and other configuration sources.
- Send logging output to the console and debug providers.

For more information, see .NET Generic Host.

Non-web scenarios

The Generic Host allows other types of apps to use cross-cutting framework extensions, such as logging, dependency injection (DI), configuration, and app lifetime management. For more information, see .NET Generic Host and Background tasks with hosted services in ASP.NET Core.

Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of request features composed into an `HttpContext`.

- Windows
- macOS
- Linux

ASP.NET Core provides the following server implementations:

- *Kestrel* is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using IIS. In ASP.NET Core 2.0 or later, Kestrel can be run as a public-facing edge server exposed directly to the Internet.
- *IIS HTTP Server* is a server for Windows that uses IIS. With this server, the ASP.NET Core app and IIS run in the same process.
- *HTTP.sys* is a server for Windows that isn't used with IIS.

For more information, see Web server implementations in ASP.NET Core.

Configuration

ASP.NET Core provides a configuration framework that gets settings as name-value pairs from an ordered set of configuration providers. Built-in configuration providers are available for a variety of sources, such as *.json* files, *.xml* files, environment variables, and command-line arguments. Write custom configuration providers to support other sources.

By default, ASP.NET Core apps are configured to read from *appsettings.json*, environment variables, the command line, and more. When the app's configuration is loaded, values from environment variables override values from *appsettings.json*.

The preferred way to read related configuration values is using the options pattern. For more information, see Bind hierarchical configuration data using the options pattern.

For managing confidential configuration data such as passwords, ASP.NET Core provides the Secret Manager. For production secrets, we recommend Azure Key Vault.

For more information, see Configuration in ASP.NET Core.

Environments

Execution environments, such as `Development`, `Staging`, and `Production`, are a first-class notion in ASP.NET Core. Specify the environment an app is running in by setting the `ASPNETCORE_ENVIRONMENT` environment variable. ASP.NET Core reads that environment variable at app startup and stores the value in an `IWebHostEnvironment` implementation. This implementation is available anywhere in an app via dependency injection (DI).

The following example configures the app to provide detailed error information when running in the `Development` environment:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });
}
```

For more information, see Use multiple environments in ASP.NET Core.

Logging

ASP.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. Available providers include:

- Console
- Debug
- Event Tracing on Windows
- Windows Event Log
- TraceSource
- Azure App Service
- Azure Application Insights

To create logs, resolve an ILogger<TCategoryName> service from dependency injection (DI) and call logging methods such as LogInformation. For example:

```
public class TodoController : ControllerBase
{
    private readonly ILogger _logger;

    public TodoController(ILogger<TodoController> logger)
    {
        _logger = logger;
    }

    [HttpGet("{id}", Name = "GetTodo")]
    public ActionResult<TodoItem> GetById(string id)
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);

        // Item lookup code removed.

        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
            return NotFound();
        }

        return item;
    }
}
```

Logging methods such as `LogInformation` support any number of fields. These fields are commonly used to construct a message `string`, but some logging providers send these to a data store as separate fields. This feature makes it possible for logging providers to implement semantic logging, also known as structured logging.

For more information, see Logging in .NET Core and ASP.NET Core.

Routing

A *route* is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

For more information, see Routing in ASP.NET Core.

Error handling

ASP.NET Core has built-in features for handling errors, such as:

- A developer exception page
- Custom error pages
- Static status code pages
- Startup exception handling

For more information, see Handle errors in ASP.NET Core.

Make HTTP requests

An implementation of `IHttpClientFactory` is available for creating `HttpClient` instances. The factory:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, register and configure a *github* client for accessing GitHub. Register and configure a default client for other purposes.
- Supports registration and chaining of multiple delegating handlers to build an outgoing request middleware pipeline. This pattern is similar to ASP.NET Core's inbound middleware pipeline. The pattern provides a mechanism to manage cross-cutting concerns for HTTP requests, including caching, error handling,

serialization, and logging.

- Integrates with *Polly*, a popular third-party library for transient fault handling.
- Manages the pooling and lifetime of underlying `HttpClientHandler` instances to avoid common DNS problems that occur when managing `HttpClient` lifetimes manually.
- Adds a configurable logging experience via ILogger for all requests sent through clients created by the factory.

For more information, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

Content root

The content root is the base path for:

- The executable hosting the app (*.exe*).
- Compiled assemblies that make up the app (*.dll*).
- Content files used by the app, such as:
  - Razor files (*.cshtml*, *.razor*)
  - Configuration files (*.json*, *.xml*)
  - Data files (*.db*)
- The Web root, typically the *wwwroot* folder.

During development, the content root defaults to the project's root directory. This directory is also the base path for both the app's content files and the Web root. Specify a different content root by setting its path when building the host. For more information, see Content root.

Web root

The web root is the base path for public, static resource files, such as:

- Stylesheets (*.css*)
- JavaScript (*.js*)
- Images (*.png*, *.jpg*)

By default, static files are served only from the web root directory and its sub-directories. The web root path defaults to *{content root}/wwwroot*. Specify a different web root by setting its path when building the host. For more information, see Web root.

Prevent publishing files in *wwwroot* with the <Content> project item in the project file. The following example prevents publishing content in *wwwroot/local* and its sub-directories:

```
<ItemGroup>
  <Content Update="wwwroot\local\**\*.*" CopyToPublishDirectory="Never" />
</ItemGroup>
```

In Razor *.cshtml* files, tilde-slash (`~/`) points to the web root. A path beginning with `~/` is referred to as a *virtual path*.

For more information, see Static files in ASP.NET Core.

This article is an overview of key topics for understanding how to develop ASP.NET Core apps.

The Startup class

The `Startup` class is where:

- Services required by the app are configured.

- The request handling pipeline is defined.

Services are components that are used by the app. For example, a logging component is a service. Code to configure (or *register*) services is added to the `Startup.ConfigureServices` method.

The request handling pipeline is composed as a series of *middleware* components. For example, a middleware might handle requests for static files or redirect HTTP requests to HTTPS. Each middleware performs asynchronous operations on an `HttpContext` and then either invokes the next middleware in the pipeline or terminates the request. Code to configure the request handling pipeline is added to the `Startup.Configure` method.

Here's a sample `Startup` class:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

        services.AddDbContext<MovieContext>(options =>
                options.UseSqlServer(Configuration.GetConnectionString("MovieDb")));
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseMvc();
    }
}
```

For more information, see App startup in ASP.NET Core.

Dependency injection (services)

ASP.NET Core has a built-in dependency injection (DI) framework that makes configured services available to an app's classes. One way to get an instance of a service in a class is to create a constructor with a parameter of the required type. The parameter can be the service type or an interface. The DI system provides the service at runtime.

Here's a class that uses DI to get an Entity Framework Core context object. The highlighted line is an example of constructor injection:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovieContext context)
    {
        _context = context;
    }

    // ...

    public async Task OnGetAsync()
    {
        Movies = await _context.Movies.ToListAsync();
    }
}
```

While DI is built in, it's designed to let you plug in a third-party Inversion of Control (IoC) container if you prefer.

For more information, see Dependency injection in ASP.NET Core.

Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs asynchronous operations on an `HttpContext` and then either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline by invoking its `Use...` extension method in the `Startup.Configure` method. For example, to enable rendering of static files, call `UseStaticFiles`.

The highlighted code in the following example configures the request handling pipeline:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

        services.AddDbContext<MovieContext>(options =>
                options.UseSqlServer(Configuration.GetConnectionString("MovieDb")));
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseMvc();
    }
}
```

ASP.NET Core includes a rich set of built-in middleware, and you can write custom middleware.

For more information, see ASP.NET Core Middleware.

Host

An ASP.NET Core app builds a *host* on startup. The host is an object that encapsulates all of the app's resources, such as:

- An HTTP server implementation
- Middleware components
- Logging
- DI
- Configuration

The main reason for including all of the app's interdependent resources in one object is lifetime management: control over app startup and graceful shutdown.

Two hosts are available: the Web Host and the Generic Host. In ASP.NET Core 2.x, the Generic Host is only for non-web scenarios.

The code to create a host is in `Program.Main`:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

The `CreateDefaultBuilder` method configures a host with commonly used options, such as the following:

- Use Kestrel as the web server and enable IIS integration.
- Load configuration from *appsettings.json*, *appsettings.{Environment Name}.json*, environment variables, command line arguments, and other configuration sources.
- Send logging output to the console and debug providers.

For more information, see ASP.NET Core Web Host.

Non-web scenarios

The Generic Host allows other types of apps to use cross-cutting framework extensions, such as logging, dependency injection (DI), configuration, and app lifetime management. For more information, see .NET Generic Host and Background tasks with hosted services in ASP.NET Core.

Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of request features composed into an `HttpContext`.

- Windows
- macOS
- Linux

ASP.NET Core provides the following server implementations:

- *Kestrel* is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using IIS. Kestrel can be run as a public-facing edge server exposed directly to the Internet.
- *IIS HTTP Server* is a server for windows that uses IIS. With this server, the ASP.NET Core app and IIS run in the same process.
- *HTTP.sys* is a server for Windows that isn't used with IIS.

- Windows
- macOS
- Linux

ASP.NET Core provides the following server implementations:

- *Kestrel* is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using IIS. Kestrel can be run as a public-facing edge server exposed directly to the Internet.
- *HTTP.sys* is a server for Windows that isn't used with IIS.

For more information, see Web server implementations in ASP.NET Core.

Configuration

ASP.NET Core provides a configuration framework that gets settings as name-value pairs from an ordered set of configuration providers. There are built-in configuration providers for a variety of sources, such as *.json* files, *.xml* files, environment variables, and command-line arguments. You can also write custom configuration providers.

For example, you could specify that configuration comes from *appsettings.json* and environment variables. Then when the value of *ConnectionString* is requested, the framework looks first in the *appsettings.json* file. If the value is found there but also in an environment variable, the value from the environment variable would take precedence.

For managing confidential configuration data such as passwords, ASP.NET Core provides a Secret Manager tool. For production secrets, we recommend Azure Key Vault.

For more information, see Configuration in ASP.NET Core.

Options

Where possible, ASP.NET Core follows the *options pattern* for storing and retrieving configuration values. The options pattern uses classes to represent groups of related settings.

For example, the following code sets WebSockets options:

```
var options = new WebSocketOptions
{
   KeepAliveInterval = TimeSpan.FromSeconds(120),
   ReceiveBufferSize = 4096
};

app.UseWebSockets(options);
```

For more information, see Options pattern in ASP.NET Core.

Environments

Execution environments, such as *Development*, *Staging*, and *Production*, are a first-class notion in ASP.NET Core. You can specify the environment an app is running in by setting the `ASPNETCORE_ENVIRONMENT` environment variable. ASP.NET Core reads that environment variable at app startup and stores the value in an `IHostingEnvironment` implementation. The environment object is available anywhere in the app via DI.

The following sample code from the `Startup` class configures the app to provide detailed error information only when it runs in development:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseMvc();
}
```

For more information, see Use multiple environments in ASP.NET Core.

Logging

ASP.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. Available providers include the following:

- Console
- Debug
- Event Tracing on Windows
- Windows Event Log
- TraceSource
- Azure App Service
- Azure Application Insights

Write logs from anywhere in an app's code by getting an `ILogger` object from DI and calling log methods.

Here's sample code that uses an `ILogger` object, with constructor injection and the logging method calls highlighted.

```
public class TodoController : ControllerBase
{
    private readonly ILogger _logger;

    public TodoController(ILogger<TodoController> logger)
    {
        _logger = logger;
    }

    [HttpGet("{id}", Name = "GetTodo")]
    public ActionResult<TodoItem> GetById(string id)
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
        // Item lookup code removed.
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
            return NotFound();
        }
        return item;
    }
}
```

The `ILogger` interface lets you pass any number of fields to the logging provider. The fields are commonly used to construct a message string, but the provider can also send them as separate fields to a data store. This feature makes it possible for logging providers to implement semantic logging, also known as structured logging.

For more information, see Logging in .NET Core and ASP.NET Core.

Routing

A *route* is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

For more information, see Routing in ASP.NET Core.

Error handling

ASP.NET Core has built-in features for handling errors, such as:

- A developer exception page

- Custom error pages
- Static status code pages
- Startup exception handling

For more information, see Handle errors in ASP.NET Core.

Make HTTP requests

An implementation of `IHttpClientFactory` is available for creating `HttpClient` instances. The factory:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, a *github* client can be registered and configured to access GitHub. A default client can be registered for other purposes.
- Supports registration and chaining of multiple delegating handlers to build an outgoing request middleware pipeline. This pattern is similar to the inbound middleware pipeline in ASP.NET Core. The pattern provides a mechanism to manage cross-cutting concerns around HTTP requests, including caching, error handling, serialization, and logging.
- Integrates with *Polly*, a popular third-party library for transient fault handling.
- Manages the pooling and lifetime of underlying `HttpClientHandler` instances to avoid common DNS problems that occur when manually managing `HttpClient` lifetimes.
- Adds a configurable logging experience (via `ILogger`) for all requests sent through clients created by the factory.

For more information, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

Content root

The content root is the base path to the:

- Executable hosting the app (*.exe*).
- Compiled assemblies that make up the app (*.dll*).
- Non-code content files used by the app, such as:
  - Razor files (*.cshtml*, *.razor*)
  - Configuration files (*.json*, *.xml*)
  - Data files (*.db*)
- Web root, typically the published *wwwroot* folder.

During development:

- The content root defaults to the project's root directory.
- The project's root directory is used to create the:
  - Path to the app's non-code content files in the project's root directory.
  - Web root, typically the *wwwroot* folder in the project's root directory.

An alternative content root path can be specified when building the host. For more information, see ASP.NET Core Web Host.

Web root

The web root is the base path to public, non-code, static resource files, such as:

- Stylesheets (*.css*)
- JavaScript (*.js*)
- Images (*.png*, *.jpg*)

Static files are only served by default from the web root directory (and sub-directories).

The web root path defaults to *{content root}/wwwroot*, but a different web root can be specified when building the host. For more information, see Web root.

Prevent publishing files in *wwwroot* with the <Content> project item in the project file. The following example prevents publishing content in the *wwwroot/local* directory and sub-directories:

```
<ItemGroup>
  <Content Update="wwwroot\local\**\*.*" CopyToPublishDirectory="Never" />
</ItemGroup>
```

In Razor (*.cshtml*) files, the tilde-slash (`~/`) points to the web root. A path beginning with `~/` is referred to as a *virtual path*.

For more information, see Static files in ASP.NET Core.

App startup in ASP.NET Core

9/22/2020 • 13 minutes to read • Edit Online

By Rick Anderson, Tom Dykstra, and Steve Smith

The `Startup` class configures services and the app's request pipeline.

The Startup class

ASP.NET Core apps use a `Startup` class, which is named `Startup` by convention. The `Startup` class:

- Optionally includes a ConfigureServices method to configure the app's *services*. A service is a reusable component that provides app functionality. Services are *registered* in `ConfigureServices` and consumed across the app via dependency injection (DI) or ApplicationServices.
- Includes a Configure method to create the app's request processing pipeline.

`ConfigureServices` and `Configure` are called by the ASP.NET Core runtime when the app starts:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

The preceding sample is for Razor Pages; the MVC version is similar.

The `Startup` class is specified when the app's host is built. The `Startup` class is typically specified by calling the WebHostBuilderExtensions.UseStartup<TStartup> method on the host builder:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The host provides services that are available to the `Startup` class constructor. The app adds additional services via `ConfigureServices`. Both the host and app services are available in `Configure` and throughout the app.

Only the following service types can be injected into the `Startup` constructor when using the Generic Host (IHostBuilder):

- IWebHostEnvironment
- IHostEnvironment
- IConfiguration

```
public class Startup
{
    private readonly IWebHostEnvironment _env;

    public Startup(IConfiguration configuration, IWebHostEnvironment env)
    {
        Configuration = configuration;
        _env = env;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (_env.IsDevelopment())
        {
        }
        else
        {
        }
    }
}
```

Most services are not available until the `Configure` method is called.

Multiple Startup

When the app defines separate `Startup` classes for different environments (for example, `StartupDevelopment`), the appropriate `Startup` class is selected at runtime. The class whose name suffix matches the current environment is prioritized. If the app is run in the Development environment and includes both a `Startup`

class and a `StartupDevelopment` class, the `StartupDevelopment` class is used. For more information, see Use multiple environments.

See The host for more information on the host. For information on handling errors during startup, see Startup exception handling.

The ConfigureServices method

The ConfigureServices method is:

- Optional.
- Called by the host before the `Configure` method to configure the app's services.
- Where configuration options are set by convention.

The host may configure some services before `Startup` methods are called. For more information, see The host.

For features that require substantial setup, there are `Add{Service}` extension methods on IServiceCollection. For example, **Add**DbContext, **Add**DefaultIdentity, **Add**EntityFrameworkStores, and **Add**RazorPages:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {

        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(
            options => options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();

        services.AddRazorPages();
    }
```

Adding services to the service container makes them available within the app and in the `Configure` method. The services are resolved via dependency injection or from ApplicationServices.

The Configure method

The Configure method is used to specify how the app responds to HTTP requests. The request pipeline is configured by adding middleware components to an IApplicationBuilder instance. `IApplicationBuilder` is available to the `Configure` method, but it isn't registered in the service container. Hosting creates an `IApplicationBuilder` and passes it directly to `Configure`.

The ASP.NET Core templates configure the pipeline with support for:

- Developer Exception Page
- Exception handler
- HTTP Strict Transport Security (HSTS)
- HTTPS redirection

- Static files
- ASP.NET Core MVC and Razor Pages

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

The preceding sample is for Razor Pages; the MVC version is similar.

Each `Use` extension method adds one or more middleware components to the request pipeline. For instance, UseStaticFiles configures middleware to serve static files.

Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the chain, if appropriate.

Additional services, such as `IWebHostEnvironment`, `ILoggerFactory`, or anything defined in `ConfigureServices`, can be specified in the `Configure` method signature. These services are injected if they're available.

For more information on how to use `IApplicationBuilder` and the order of middleware processing, see ASP.NET Core Middleware.

Configure services without Startup

To configure services and the request processing pipeline without using a `Startup` class, call `ConfigureServices` and `Configure` convenience methods on the host builder. Multiple calls to `ConfigureServices` append to one another. If multiple `Configure` method calls exist, the last `Configure` call is

used.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.ConfigureServices(services =>
                {
                    services.AddControllersWithViews();
                })
                .Configure(app =>
                {
                    var loggerFactory = app.ApplicationServices
                        .GetRequiredService<ILoggerFactory>();
                    var logger = loggerFactory.CreateLogger<Program>();
                    var env = app.ApplicationServices.GetRequiredService<IWebHostEnvironment>();
                    var config = app.ApplicationServices.GetRequiredService<IConfiguration>();

                    logger.LogInformation("Logged in Configure");

                    if (env.IsDevelopment())
                    {
                        app.UseDeveloperExceptionPage();
                    }
                    else
                    {
                        app.UseExceptionHandler("/Home/Error");
                        app.UseHsts();
                    }

                    var configValue = config["MyConfigKey"];
                });
            });
        });
}
```

Extend Startup with startup filters

Use IStartupFilter:

- To configure middleware at the beginning or end of an app's Configure middleware pipeline without an explicit call to `Use{Middleware}`. `IStartupFilter` is used by ASP.NET Core to add defaults to the beginning of the pipeline without having to make the app author explicitly register the default middleware. `IStartupFilter` allows a different component call `Use{Middleware}` on behalf of the app author.
- To create a pipeline of `Configure` methods. IStartupFilter.Configure can set a middleware to run before or after middleware added by libraries.

`IStartupFilter` implements Configure, which receives and returns an `Action<IApplicationBuilder>`. An IApplicationBuilder defines a class to configure an app's request pipeline. For more information, see Create a middleware pipeline with IApplicationBuilder.

Each `IStartupFilter` can add one or more middlewares in the request pipeline. The filters are invoked in the

order they were added to the service container. Filters may add middleware before or after passing control to the next filter, thus they append to the beginning or end of the app pipeline.

The following example demonstrates how to register a middleware with `IStartupFilter`. The `RequestSetOptionsMiddleware` middleware sets an options value from a query string parameter:

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;

    public RequestSetOptionsMiddleware( RequestDelegate next )
    {
        _next = next;
    }

    // Test with https://localhost:5001/Privacy/?option=Hello
    public async Task Invoke(HttpContext httpContext)
    {
        var option = httpContext.Request.Query["option"];

        if (!string.IsNullOrWhiteSpace(option))
        {
            httpContext.Items["option"] = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

The `RequestSetOptionsMiddleware` is configured in the `RequestSetOptionsStartupFilter` class:

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

The `IStartupFilter` is registered in the service container in ConfigureServices.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
           .ConfigureAppConfiguration((hostingContext, config) =>
           {
           })
         .ConfigureWebHostDefaults(webBuilder =>
         {
             webBuilder.UseStartup<Startup>();
         })
        .ConfigureServices(services =>
        {
            services.AddTransient<IStartupFilter,
                      RequestSetOptionsStartupFilter>();
        });
}
```

When a query string parameter for `option` is provided, the middleware processes the value assignment before the ASP.NET Core middleware renders the response.

Middleware execution order is set by the order of `IStartupFilter` registrations:

- Multiple `IStartupFilter` implementations may interact with the same objects. If ordering is important, order their `IStartupFilter` service registrations to match the order that their middlewares should run.
- Libraries may add middleware with one or more `IStartupFilter` implementations that run before or after other app middleware registered with `IStartupFilter`. To invoke an `IStartupFilter` middleware before a middleware added by a library's `IStartupFilter`:
  - Position the service registration before the library is added to the service container.
  - To invoke afterward, position the service registration after the library is added.

Add configuration at startup from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Startup` class. For more information, see Use hosting startup assemblies in ASP.NET Core.

Additional resources

- The host
- Use multiple environments in ASP.NET Core
- ASP.NET Core Middleware
- Logging in .NET Core and ASP.NET Core
- Configuration in ASP.NET Core

The Startup class

ASP.NET Core apps use a `Startup` class, which is named `Startup` by convention. The `Startup` class:

- Optionally includes a ConfigureServices method to configure the app's *services*. A service is a reusable component that provides app functionality. Services are *registered* in `ConfigureServices` and consumed

across the app via dependency injection (DI) or ApplicationServices.

- Includes a Configure method to create the app's request processing pipeline.

`ConfigureServices` and `Configure` are called by the ASP.NET Core runtime when the app starts:

```
public class Startup
{
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
        ...
    }
}
```

The `Startup` class is specified when the app's host is built. The `Startup` class is typically specified by calling the WebHostBuilderExtensions.UseStartup<TStartup> method on the host builder:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

The host provides services that are available to the `Startup` class constructor. The app adds additional services via `ConfigureServices`. Both the host and app services are then available in `Configure` and throughout the app.

A common use of dependency injection into the `Startup` class is to inject:

- IHostingEnvironment to configure services by environment.
- IConfiguration to read configuration.
- ILoggerFactory to create a logger in `Startup.ConfigureServices`.

```
public class Startup
{
    private readonly IHostingEnvironment _env;
    private readonly IConfiguration _config;
    private readonly ILoggerFactory _loggerFactory;

    public Startup(IHostingEnvironment env, IConfiguration config,
        ILoggerFactory loggerFactory)
    {
        _env = env;
        _config = config;
        _loggerFactory = loggerFactory;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        var logger = _loggerFactory.CreateLogger<Startup>();

        if (_env.IsDevelopment())
        {
            // Development service configuration

            logger.LogInformation("Development environment");
        }
        else
        {
            // Non-development service configuration

            logger.LogInformation("Environment: {EnvironmentName}", _env.EnvironmentName);
        }

        // Configuration is available during startup.
        // Examples:
        //   _config["key"]
        //   _config["subsection:suboption1"]
    }
}
```

Most services are not available until the `Configure` method is called.

Multiple Startup

When the app defines separate `Startup` classes for different environments (for example, `StartupDevelopment`), the appropriate `Startup` class is selected at runtime. The class whose name suffix matches the current environment is prioritized. If the app is run in the Development environment and includes both a `Startup` class and a `StartupDevelopment` class, the `StartupDevelopment` class is used. For more information, see Use multiple environments.

See The host for more information on the host. For information on handling errors during startup, see Startup exception handling.

The ConfigureServices method

The ConfigureServices method is:

- Optional.
- Called by the host before the `Configure` method to configure the app's services.
- Where configuration options are set by convention.

The host may configure some services before `Startup` methods are called. For more information, see The host.

For features that require substantial setup, there are `Add{Service}` extension methods on IServiceCollection. For example, **Add**DbContext, **Add**DefaultIdentity, **Add**EntityFrameworkStores, and **Add**RazorPages:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

Adding services to the service container makes them available within the app and in the `Configure` method. The services are resolved via dependency injection or from ApplicationServices.

See SetCompatibilityVersion for more information on `SetCompatibilityVersion`.

The Configure method

The Configure method is used to specify how the app responds to HTTP requests. The request pipeline is configured by adding middleware components to an IApplicationBuilder instance. `IApplicationBuilder` is available to the `Configure` method, but it isn't registered in the service container. Hosting creates an `IApplicationBuilder` and passes it directly to `Configure`.

The ASP.NET Core templates configure the pipeline with support for:

- Developer Exception Page
- Exception handler
- HTTP Strict Transport Security (HSTS)
- HTTPS redirection
- Static files
- ASP.NET Core MVC and Razor Pages
- General Data Protection Regulation (GDPR)

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseMvc();
}
```

Each `Use` extension method adds one or more middleware components to the request pipeline. For instance, UseStaticFiles configures middleware to serve static files.

Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the chain, if appropriate.

Additional services, such as `IHostingEnvironment` and `ILoggerFactory`, or anything defined in `ConfigureServices`, can be specified in the `Configure` method signature. These services are injected if they're available.

For more information on how to use `IApplicationBuilder` and the order of middleware processing, see ASP.NET Core Middleware.

Configure services without Startup

To configure services and the request processing pipeline without using a `Startup` class, call `ConfigureServices` and `Configure` convenience methods on the host builder. Multiple calls to `ConfigureServices` append to one another. If multiple `Configure` method calls exist, the last `Configure` call is used.

```
public class Program
{
    public static IHostingEnvironment HostingEnvironment { get; set; }
    public static IConfiguration Configuration { get; set; }

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
            })
            .ConfigureServices(services =>
            {
                ...
            })
            .Configure(app =>
            {
                var loggerFactory = app.ApplicationServices
                    .GetRequiredService<ILoggerFactory>();
                var logger = loggerFactory.CreateLogger<Program>();
                var env = app.ApplicationServices.GetRequiredService<IHostingEnvironment>();
                var config = app.ApplicationServices.GetRequiredService<IConfiguration>();

                logger.LogInformation("Logged in Configure");

                if (env.IsDevelopment())
                {
                    ...
                }
                else
                {
                    ...
                }

                var configValue = config["subsection:suboption1"];

                ...
            });
}
```

Extend Startup with startup filters

Use IStartupFilter:

- To configure middleware at the beginning or end of an app's Configure middleware pipeline without an explicit call to `Use{Middleware}`. `IStartupFilter` is used by ASP.NET Core to add defaults to the beginning of the pipeline without having to make the app author explicitly register the default middleware. `IStartupFilter` allows a different component call `Use{Middleware}` on behalf of the app author.
- To create a pipeline of `Configure` methods. IStartupFilter.Configure can set a middleware to run before or after middleware added by libraries.

`IStartupFilter` implements Configure, which receives and returns an `Action<IApplicationBuilder>`. An IApplicationBuilder defines a class to configure an app's request pipeline. For more information, see Create a middleware pipeline with IApplicationBuilder.

Each `IStartupFilter` can add one or more middlewares in the request pipeline. The filters are invoked in the order they were added to the service container. Filters may add middleware before or after passing control to the next filter, thus they append to the beginning or end of the app pipeline.

The following example demonstrates how to register a middleware with `IStartupFilter`. The `RequestSetOptionsMiddleware` middleware sets an options value from a query string parameter:

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;
    private IOptions<AppOptions> _injectedOptions;

    public RequestSetOptionsMiddleware(
        RequestDelegate next, IOptions<AppOptions> injectedOptions)
    {
        _next = next;
        _injectedOptions = injectedOptions;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        Console.WriteLine("RequestSetOptionsMiddleware.Invoke");

        var option = httpContext.Request.Query["option"];

        if (!string.IsNullOrWhiteSpace(option))
        {
            _injectedOptions.Value.Option = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

The `RequestSetOptionsMiddleware` is configured in the `RequestSetOptionsStartupFilter` class:

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

The `IStartupFilter` is registered in the service container in ConfigureServices.

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddTransient<IStartupFilter,
            RequestSetOptionsStartupFilter>();
    })
    .UseStartup<Startup>()
    .Build();
```

When a query string parameter for `option` is provided, the middleware processes the value assignment before the ASP.NET Core middleware renders the response.

Middleware execution order is set by the order of `IStartupFilter` registrations:

- Multiple `IStartupFilter` implementations may interact with the same objects. If ordering is important,

order their `IStartupFilter` service registrations to match the order that their middlewares should run.

- Libraries may add middleware with one or more `IStartupFilter` implementations that run before or after other app middleware registered with `IStartupFilter`. To invoke an `IStartupFilter` middleware before a middleware added by a library's `IStartupFilter`:
  - Position the service registration before the library is added to the service container.
  - To invoke afterward, position the service registration after the library is added.

Add configuration at startup from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Startup` class. For more information, see Use hosting startup assemblies in ASP.NET Core.

Additional resources

- The host
- Use multiple environments in ASP.NET Core
- ASP.NET Core Middleware
- Logging in .NET Core and ASP.NET Core
- Configuration in ASP.NET Core

Dependency injection in ASP.NET Core

9/22/2020 • 40 minutes to read • Edit Online

By Kirk Larkin, Steve Smith, Scott Addie, and Brandon Dahler

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

For more information specific to dependency injection within MVC controllers, see Dependency injection into controllers in ASP.NET Core.

For more information on dependency injection of options, see Options pattern in ASP.NET Core.

View or download sample code (how to download)

Overview of dependency injection

A *dependency* is an object that another object depends on. Examine the following `MyDependency` class with a `WriteMessage` method that other classes depend on:

```
public class MyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage called. Message:
{message}");
    }
}
```

A class can create an instance of the `MyDependency` class to make use of its `WriteMessage` method. In the following example, the `MyDependency` class is a dependency of the `IndexModel` class:

```
public class IndexModel : PageModel
{
    private readonly MyDependency _dependency = new MyDependency();

    public void OnGet()
    {
        _dependency.WriteMessage("IndexModel.OnGet created this message.");
    }
}
```

The class creates and directly depends on the `MyDependency` class. Code dependencies, such as in the previous example, are problematic and should be avoided for the following reasons:

- To replace `MyDependency` with a different implementation, the `IndexModel` class must be modified.
- If `MyDependency` has dependencies, they must also be configured by the `IndexModel` class. In a large project with multiple classes depending on

`MyDependency`, the configuration code becomes scattered across the app.

- This implementation is difficult to unit test. The app should use a mock or stub `MyDependency` class, which isn't possible with this approach.

Dependency injection addresses these problems through:

- The use of an interface or base class to abstract the dependency implementation.
- Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, IServiceProvider. Services are typically registered in the app's `Startup.ConfigureServices` method.
- *Injection* of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

In the sample app, the `IMyDependency` interface defines the `WriteMessage` method:

```
public interface IMyDependency
{
    void WriteMessage(string message);
}
```

This interface is implemented by a concrete type, `MyDependency`:

```
public class MyDependency : IMyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage Message: {message}");
    }
}
```

The sample app registers the `IMyDependency` service with the concrete type `MyDependency`. The AddScoped method registers the service with a scoped lifetime, the lifetime of a single request. Service lifetimes are described later in this topic.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyDependency, MyDependency>();

    services.AddRazorPages();
}
```

In the sample app, the `IMyDependency` service is requested and used to call the `WriteMessage` method:

```
public class Index2Model : PageModel
{
    private readonly IMyDependency _myDependency;

    public Index2Model(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }

    public void OnGet()
    {
        _myDependency.WriteMessage("Index2Model.OnGet");
    }
}
```

By using the DI pattern, the controller:

- Doesn't use the concrete type `MyDependency`, only the `IMyDependency` interface it implements. That makes it easy to change the implementation that the controller uses without modifying the controller.
- Doesn't create an instance of `MyDependency`, it's created by the DI container.

The implementation of the `IMyDependency` interface can be improved by using the built-in logging API:

```
public class MyDependency2 : IMyDependency
{
    private readonly ILogger<MyDependency2> _logger;

    public MyDependency2(ILogger<MyDependency2> logger)
    {
        _logger = logger;
    }

    public void WriteMessage(string message)
    {
        _logger.LogInformation( $"MyDependency2.WriteMessage Message: {message}");
    }
}
```

The updated `ConfigureServices` method registers the new `IMyDependency` implementation:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyDependency, MyDependency2>();

    services.AddRazorPages();
}
```

`MyDependency2` depends on ILogger<TCategoryName>, which it requests in the constructor. `ILogger<TCategoryName>` is a framework-provided service.

It's not unusual to use dependency injection in a chained fashion. Each requested dependency in turn requests its own dependencies. The container resolves the dependencies in the graph and returns the fully resolved service. The collective set of dependencies that must be resolved is typically referred to as a *dependency tree*, *dependency graph*, or *object graph*.

The container resolves `ILogger<TCategoryName>` by taking advantage of (generic) open types, eliminating the need to register every (generic) constructed type.

In dependency injection terminology, a service:

- Is typically an object that provides a service to other objects, such as the `IMyDependency` service.
- Is not related to a web service, although the service may use a web service.

The framework provides a robust logging system. The `IMyDependency` implementations shown in the preceding examples were written to demonstrate basic DI, not to implement logging. Most apps shouldn't need to write loggers. The following code demonstrates using the default logging, which doesn't require any services to be registered in `ConfigureServices`:

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }

    public string Message { get; set; }

    public void OnGet()
    {
        Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
        _logger.LogInformation(Message);
    }
}
```

Using the preceding code, there is no need to update `ConfigureServices`, because logging is provided by the framework.

Services injected into Startup

Services can be injected into the `Startup` constructor and the `Startup.Configure` method.

Only the following services can be injected into the `Startup` constructor when using the Generic Host (IHostBuilder):

- IWebHostEnvironment
- IHostEnvironment
- IConfiguration

Any service registered with the DI container can be injected into the `Startup.Configure` method:

```
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
    ...
}
```

For more information, see App startup in ASP.NET Core and Access configuration in Startup.

Register groups of services with extension methods

The ASP.NET Core framework uses a convention for registering a group of related services. The convention is to use a single `Add{GROUP_NAME}` extension method to register all of the services required by a framework feature. For example, the <Microsoft.Extensions.DependencyInjection.MvcServiceCollectionExtensions.AddControllers> extension method registers the services required for MVC controllers.

The following code is generated by the Razor Pages template using individual user accounts and shows how to add additional services to the container using the extension methods AddDbContext and AddDefaultIdentity:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
}
```

Consider the following `ConfigureServices` method, which registers services and configures options:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<PositionOptions>(
        Configuration.GetSection(PositionOptions.Position));
    services.Configure<ColorOptions>(
        Configuration.GetSection(ColorOptions.Color));

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddScoped<IMyDependency2, MyDependency2>();

    services.AddRazorPages();
}
```

Related groups of registrations can be moved to an extension method to register services. For example, the configuration services are added to the following class:

```
using ConfigSample.Options;
using Microsoft.Extensions.Configuration;

namespace Microsoft.Extensions.DependencyInjection
{
    public static class MyConfigServiceCollectionExtensions
    {
        public static IServiceCollection AddConfig(
             this IServiceCollection services, IConfiguration config)
        {
            services.Configure<PositionOptions>(
                config.GetSection(PositionOptions.Position));
            services.Configure<ColorOptions>(
                config.GetSection(ColorOptions.Color));

            return services;
        }
    }
}
```

The remaining services are registered in a similar class. The following `ConfigureServices` method uses the new extension methods to register the services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddConfig(Configuration)
            .AddMyDependencyGroup();

    services.AddRazorPages();
}
```

Note: Each `services.Add{GROUP_NAME}` extension method adds and potentially configures services. For example, AddControllersWithViews adds the services MVC controllers with views require, and AddRazorPages adds the services Razor Pages requires. We recommended that apps follow this naming convention. Place extension methods in the Microsoft.Extensions.DependencyInjection namespace to encapsulate groups of service registrations.

Service lifetimes

Services can be registered with one of the following lifetimes:

- Transient
- Scoped
- Singleton

The following sections describe each of the preceding lifetimes. Choose an appropriate lifetime for each registered service.

Transient

Transient lifetime services are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services. Register transient services with AddTransient.

In apps that process requests, transient services are disposed at the end of the request.

Scoped

Scoped lifetime services are created once per client request (connection). Register scoped services with AddScoped.

In apps that process requests, scoped services are disposed at the end of the request.

When using Entity Framework Core, the AddDbContext extension method registers `DbContext` types with a scoped lifetime by default.

Do ***not*** resolve a scoped service from a singleton. It may cause the service to have incorrect state when processing subsequent requests. It's fine to:

- Resolve a singleton service from a scoped or transient service.
- Resolve a scoped service from another scoped or transient service.

By default, in the development environment, resolving a service from another service with a longer lifetime throws an exception. For more information, see Scope validation.

To use scoped services in middleware, use one of the following approaches:

- Inject the service into the middleware's `Invoke` or `InvokeAsync` method. Using constructor injection throws a runtime exception because it forces the scoped service to behave like a singleton. The sample in the Lifetime and registration options section demonstrates the `InvokeAsync` approach.
- Use Factory-based middleware. Middleware registered using this approach is activated per client request (connection), which allows scoped services to be injected into the middleware's `InvokeAsync` method.

For more information, see Write custom ASP.NET Core middleware.

Singleton

Singleton lifetime services are created either:

- The first time they're requested.
- By the developer, when providing an implementation instance directly to the container. This approach is rarely needed.

Every subsequent request uses the same instance. If the app requires singleton behavior, allow the service container to manage the service's lifetime. Don't implement the singleton design pattern and provide code to dispose of the singleton. Services should never be disposed by code that resolved the service from the container. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

Register singleton services with AddSingleton. Singleton services must be thread safe and are often used in stateless services.

In apps that process requests, singleton services are disposed when the ServiceProvider is disposed on application shutdown. Because memory is not released until the app is shut down, consider memory use with a singleton service.

> **WARNING**
> Do ***not*** resolve a scoped service from a singleton. It may cause the service to have incorrect state when processing subsequent requests. It's fine to resolve a singleton service from a scoped or transient service.

Service registration methods

The framework provides service registration extension methods that are useful in specific scenarios:

METHOD	AUTOMATIC OBJECT DISPOSAL	MULTIPLE IMPLEMENTATIONS	PASS ARGS
`Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()` Example: `services.AddSingleton<IMyDep, MyDep>();`	Yes	Yes	No
`Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION})` Examples: `services.AddSingleton<IMyDep>(sp => new MyDep());` `services.AddSingleton<IMyDep>(sp => new MyDep(99));`	Yes	Yes	Yes
`Add{LIFETIME}<{IMPLEMENTATION}>()` Example: `services.AddSingleton<MyDep>();`	Yes	No	No
`AddSingleton<{SERVICE}>(new {IMPLEMENTATION})` Examples: `services.AddSingleton<IMyDep>(new MyDep());` `services.AddSingleton<IMyDep>(new MyDep(99));`	No	Yes	Yes
`AddSingleton(new {IMPLEMENTATION})` Examples: `services.AddSingleton(new MyDep());` `services.AddSingleton(new MyDep(99));`	No	No	Yes

For more information on type disposal, see the Disposal of services section. It's common to use multiple implementations when mocking types for testing.

The framework also provides `TryAdd{LIFETIME}` extension methods, which register the service only if there isn't already an implementation registered.

In the following example, the call to `AddSingleton` registers `MyDependency` as an implementation for `IMyDependency`. The call to `TryAddSingleton` has no effect because `IMyDependency` already has a registered implementation:

```
services.AddSingleton<IMyDependency, MyDependency>();
// The following line has no effect:
services.TryAddSingleton<IMyDependency, DifferentDependency>();
```

For more information, see:

- TryAdd
- TryAddTransient
- TryAddScoped
- TryAddSingleton

The TryAddEnumerable(ServiceDescriptor) methods register the service only if there isn't already an implementation *of the same type*. Multiple services are resolved via `IEnumerable<{SERVICE}>`. When registering services, the developer should add an instance if one of the same type hasn't already been added. Generally, library authors use `TryAddEnumerable` to avoid registering multiple copies of an implementation in the container.

In the following example, the first call to `TryAddEnumerable` registers `MyDependency` as an implementation for `IMyDependency1`. The second call registers `MyDependency` for `IMyDependency2`. The third call has no effect because `IMyDependency1` already has a registered implementation of `MyDependency`:

```
public interface IMyDependency1 { }
public interface IMyDependency2 { }

public class MyDependency : IMyDependency1, IMyDependency2 { }

services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDependency1,
MyDependency>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDependency2,
MyDependency>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDependency1,
MyDependency>());
```

Service registration is generally order independent except when registering multiple implementations of the same type.

`IServiceCollection` is a collection of ServiceDescriptor objects. The following example shows how to register a service by creating and adding a `ServiceDescriptor`:

```
var myKey = Configuration["MyKey"];
var descriptor = new ServiceDescriptor(
    typeof(IMyDependency),
    sp => new MyDependency5(myKey),
    ServiceLifetime.Transient);

services.Add(descriptor);
```

The built-in `Add{LIFETIME}` methods use the same approach. For example, see the AddScoped source code.

Constructor injection behavior

Services can be resolved by using:

- IServiceProvider

- ActivatorUtilities:
  - Creates objects that aren't registered in the container.
  - Used with framework features, such as Tag Helpers, MVC controllers, and model binders.

Constructors can accept arguments that aren't provided by dependency injection, but the arguments must assign default values.

When services are resolved by `IServiceProvider` or `ActivatorUtilities`, constructor injection requires a *public* constructor.

When services are resolved by `ActivatorUtilities`, constructor injection requires that only one applicable constructor exists. Constructor overloads are supported, but only one overload can exist whose arguments can all be fulfilled by dependency injection.

Entity Framework contexts

By default, Entity Framework contexts are added to the service container using the scoped lifetime because web app database operations are normally scoped to the client request. To use a different lifetime, specify the lifetime by using an AddDbContext overload. Services of a given lifetime shouldn't use a database context with a lifetime that's shorter than the service's lifetime.

Lifetime and registration options

To demonstrate the difference between service lifetimes and their registration options, consider the following interfaces that represent a task as an operation with an identifier, `OperationId`. Depending on how the lifetime of an operation's service is configured for the following interfaces, the container provides either the same or different instances of the service when requested by a class:

```
public interface IOperation
{
    string OperationId { get; }
}

public interface IOperationTransient : IOperation { }
public interface IOperationScoped : IOperation { }
public interface IOperationSingleton : IOperation { }
```

The following `Operation` class implements all of the preceding interfaces. The `Operation` constructor generates a GUID and stores the last 4 characters in the `OperationId` property:

```
public class Operation : IOperationTransient, IOperationScoped,
IOperationSingleton
{
    public Operation()
    {
        OperationId = Guid.NewGuid().ToString()[^4..];
    }

    public string OperationId { get; }
}
```

The `Startup.ConfigureServices` method creates multiple registrations of the

`Operation` class according to the named lifetimes:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();

    services.AddRazorPages();
}
```

The sample app demonstrates object lifetimes both within and between requests. The `IndexModel` and the middleware request each kind of `IOperation` type and log the `OperationId` for each:

```
public class IndexModel : PageModel
{
    private readonly ILogger _logger;
    private readonly IOperationTransient _transientOperation;
    private readonly IOperationSingleton _singletonOperation;
    private readonly IOperationScoped _scopedOperation;

    public IndexModel(ILogger<IndexModel> logger,
                      IOperationTransient transientOperation,
                      IOperationScoped scopedOperation,
                      IOperationSingleton singletonOperation)
    {
        _logger = logger;
        _transientOperation = transientOperation;
        _scopedOperation    = scopedOperation;
        _singletonOperation = singletonOperation;
    }

    public void  OnGet()
    {
        _logger.LogInformation("Transient: " + _transientOperation.OperationId);
        _logger.LogInformation("Scoped: "    + _scopedOperation.OperationId);
        _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);
    }
}
```

Similar to the `IndexModel`, the middleware resolves the same services:

```
public class MyMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger _logger;

    private readonly IOperationTransient _transientOperation;
    private readonly IOperationSingleton _singletonOperation;

    public MyMiddleware(RequestDelegate next, ILogger<MyMiddleware> logger,
        IOperationTransient transientOperation,
        IOperationSingleton singletonOperation)
    {
        _logger = logger;
        _transientOperation = transientOperation;
        _singletonOperation = singletonOperation;
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context,
        IOperationScoped scopedOperation)
    {
        _logger.LogInformation("Transient: " + _transientOperation.OperationId);
        _logger.LogInformation("Scoped: "    + scopedOperation.OperationId);
        _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);

        await _next(context);
    }
}

public static class MyMiddlewareExtensions
{
    public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder
builder)
    {
        return builder.UseMiddleware<MyMiddleware>();
    }
}
```

Scoped services must be resolved in the `InvokeAsync` method:

```
public async Task InvokeAsync(HttpContext context,
    IOperationScoped scopedOperation)
{
    _logger.LogInformation("Transient: " + _transientOperation.OperationId);
    _logger.LogInformation("Scoped: "    + scopedOperation.OperationId);
    _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);

    await _next(context);
}
```

The logger output shows:

- *Transient* objects are always different. The transient `OperationId` value is different in the `IndexModel` and in the middleware.
- *Scoped* objects are the same for each request but different across each request.
- *Singleton* objects are the same for every request.

To reduce the logging output, set "Logging:LogLevel:Microsoft:Error" in the *appsettings.Development.json* file:

```
{
  "MyKey": "MyKey from appsettings.Developement.json",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "System": "Debug",
      "Microsoft": "Error"
    }
  }
}
```

Call services from main

Create an IServiceScope with IServiceScopeFactory.CreateScope to resolve a scoped service within the app's scope. This approach is useful to access a scoped service at startup to run initialization tasks.

The following example shows how to access the scoped `IMyDependency` service and call its `WriteMessage` method in `Program.Main`:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();

        using (var serviceScope = host.Services.CreateScope())
        {
            var services = serviceScope.ServiceProvider;

            try
            {
                var myDependency = services.GetRequiredService<IMyDependency>();
                myDependency.WriteMessage("Call services from main");
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred.");
            }
        }

        host.Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Scope validation

When the app runs in the Development environment and calls CreateDefaultBuilder to build the host, the default service provider performs checks to verify that:

- Scoped services aren't resolved from the root service provider.
- Scoped services aren't injected into singletons.

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton because it's only disposed by the root container when the app shuts down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

For more information, see Scope validation.

Request Services

The services available within an ASP.NET Core request are exposed through the HttpContext.RequestServices collection. When services are requested from inside of a request, the services and their dependencies are resolved from the `RequestServices` collection.

The framework creates a scope per request and `RequestServices` exposes the scoped service provider. All scoped services are valid for as long as the request is active.

> **NOTE**
>
> Prefer requesting dependencies as constructor parameters to resolving services from the `RequestServices` collection. This results in classes that are easier to test.

Design services for dependency injection

When designing services for dependency injection:

- Avoid stateful, static classes and members. Avoid creating global state by designing apps to use singleton services instead.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make services small, well-factored, and easily tested.

If a class has a lot of injected dependencies, it might be a sign that the class has too many responsibilities and violates the Single Responsibility Principle (SRP). Attempt to refactor the class by moving some of its responsibilities into new classes. Keep in mind that Razor Pages page model classes and MVC controller classes should focus on UI concerns.

Disposal of services

The container calls Dispose for the IDisposable types it creates. Services resolved from the container should never be disposed by the developer. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

In the following example, the services are created by the service container and disposed automatically:

```
public class Service1 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
```

```
        Console.WriteLine($"Service1: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service1.Dispose");
        _disposed = true;
    }
}

public class Service2 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
        Console.WriteLine($"Service2: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service2.Dispose");
        _disposed = true;
    }
}

public interface IService3
{
    public void Write(string message);
}

public class Service3 : IService3, IDisposable
{
    private bool _disposed;

    public Service3(string myKey)
    {
        MyKey = myKey;
    }

    public string MyKey { get; }

    public void Write(string message)
    {
        Console.WriteLine($"Service3: {message}, MyKey = {MyKey}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service3.Dispose");
        _disposed = true;
    }
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<Service1>();
    services.AddSingleton<Service2>();

    var myKey = Configuration["MyKey"];
    services.AddSingleton<IService3>(sp => new Service3(myKey));

    services.AddRazorPages();
}
```

```
public class IndexModel : PageModel
{
    private readonly Service1 _service1;
    private readonly Service2 _service2;
    private readonly IService3 _service3;

    public IndexModel(Service1 service1, Service2 service2, IService3 service3)
    {
        _service1 = service1;
        _service2 = service2;
        _service3 = service3;
    }

    public void OnGet()
    {
        _service1.Write("IndexModel.OnGet");
        _service2.Write("IndexModel.OnGet");
        _service3.Write("IndexModel.OnGet");
    }
}
```

The debug console shows the following output after each refresh of the Index page:

```
Service1: IndexModel.OnGet
Service2: IndexModel.OnGet
Service3: IndexModel.OnGet
Service1.Dispose
```

Services not created by the service container

Consider the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton(new Service1());
    services.AddSingleton(new Service2());

    services.AddRazorPages();
}
```

In the preceding code:

- The service instances aren't created by the service container.
- The framework doesn't dispose of the services automatically.
- The developer is responsible for disposing the services.

IDisposable guidance for Transient and shared instances

Transient, limited lifetime

Scenario

The app requires an IDisposable instance with a transient lifetime for either of the following scenarios:

- The instance is resolved in the root scope (root container).
- The instance should be disposed before the scope ends.

Solution

Use the factory pattern to create an instance outside of the parent scope. In this situation, the app would generally have a `Create` method that calls the final type's constructor directly. If the final type has other dependencies, the factory can:

- Receive an IServiceProvider in its constructor.
- Use ActivatorUtilities.CreateInstance to instantiate the instance outside of the container, while using the container for its dependencies.

Shared instance, limited lifetime

Scenario

The app requires a shared IDisposable instance across multiple services, but the IDisposable instance should have a limited lifetime.

Solution

Register the instance with a scoped lifetime. Use IServiceScopeFactory.CreateScope to create a new IServiceScope. Use the scope's IServiceProvider to get required services. Dispose the scope when it's no longer needed.

General IDisposable guidelines

- Don't register IDisposable instances with a transient lifetime. Use the factory pattern instead.
- Don't resolve IDisposable instances with a transient or scoped lifetime in the root scope. The only exception to this is if the app creates/recreates and disposes IServiceProvider, but this isn't an ideal pattern.
- Receiving an IDisposable dependency via DI doesn't require that the receiver implement IDisposable itself. The receiver of the IDisposable dependency shouldn't call Dispose on that dependency.
- Use scopes to control the lifetimes of services. Scopes aren't hierarchical, and there's no special connection among scopes.

Default service container replacement

The built-in service container is designed to serve the needs of the framework and most consumer apps. We recommend using the built-in container unless you need a specific feature that it doesn't support, such as:

- Property injection
- Injection based on name
- Child containers
- Custom lifetime management
- `Func<T>` support for lazy initialization
- Convention-based registration

The following third-party containers can be used with ASP.NET Core apps:

- Autofac

- Dryloc
- Grace
- LightInject
- Lamar
- Stashbox
- Unity

Thread safety

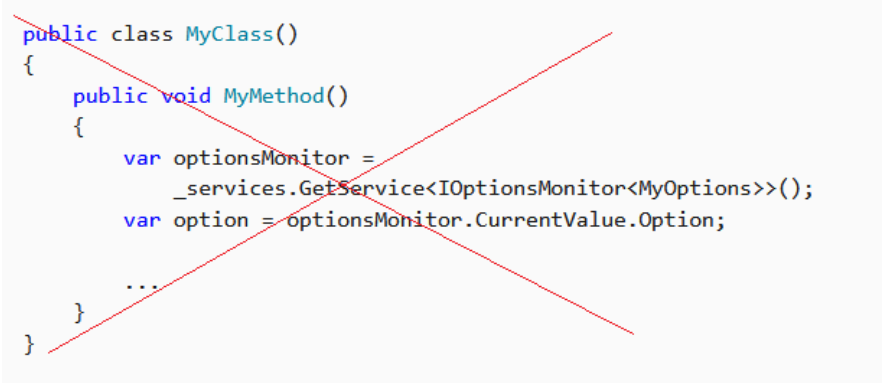
Create thread-safe singleton services. If a singleton service has a dependency on a transient service, the transient service may also require thread safety depending on how it's used by the singleton.

The factory method of single service, such as the second argument to AddSingleton<TService>(IServiceCollection, Func<IServiceProvider,TService>), doesn't need to be thread-safe. Like a type (`static`) constructor, it's guaranteed to be called only once by a single thread.

Recommendations

- `async/await` and `Task` based service resolution isn't supported. Because C# doesn't support asynchronous constructors, use asynchronous methods after synchronously resolving the service.
- Avoid storing data and configuration directly in the service container. For example, a user's shopping cart shouldn't typically be added to the service container. Configuration should use the options pattern. Similarly, avoid "data holder" objects that only exist to allow access to another object. It's better to request the actual item via DI.
- Avoid static access to services. For example, avoid capturing IApplicationBuilder.ApplicationServices as a static field or property for use elsewhere.
- Keep DI factories fast and synchronous.
- Avoid using the *service locator pattern*. For example, don't invoke GetService to obtain a service instance when you can use DI instead:

  Incorrect:



```
public class MyClass()
{
    public void MyMethod()
    {
        var optionsMonitor =
            _services.GetService<IOptionsMonitor<MyOptions>>();
        var option = optionsMonitor.CurrentValue.Option;

        ...
    }
}
```

  Correct:

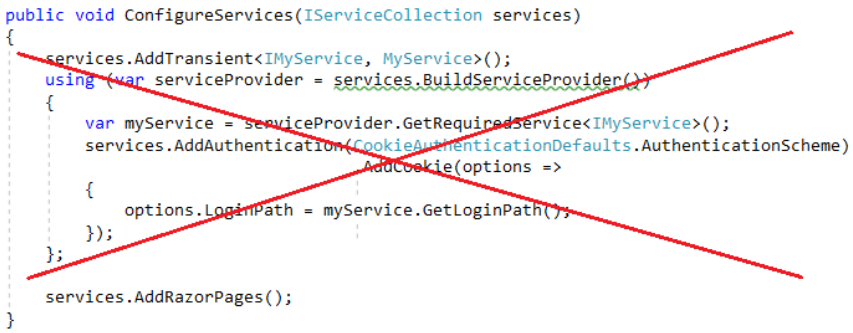
```
public class MyClass
{
    private readonly IOptionsMonitor<MyOptions> _optionsMonitor;

    public MyClass(IOptionsMonitor<MyOptions> optionsMonitor)
    {
        _optionsMonitor = optionsMonitor;
    }

    public void MyMethod()
    {
        var option = _optionsMonitor.CurrentValue.Option;

        ...
    }
}
```

- Another service locator variation to avoid is injecting a factory that resolves dependencies at runtime. Both of these practices mix Inversion of Control strategies.
- Avoid static access to `HttpContext` (for example, IHttpContextAccessor.HttpContext).
- Avoid calls to BuildServiceProvider in `ConfigureServices`. Calling `BuildServiceProvider` typically happens when the developer wants to resolve a service in `ConfigureServices`. For example, consider the case where the `LoginPath` is loaded from configuration. Avoid the following approach:



In the preceding image, selecting the green wavy line under `services.BuildServiceProvider` shows the following ASP0000 warning:

> ASP0000 Calling 'BuildServiceProvider' from application code results in an additional copy of singleton services being created. Consider alternatives such as dependency injecting services as parameters to 'Configure'.

Calling `BuildServiceProvider` creates a second container, which can create torn singletons and cause references to object graphs across multiple containers.

A correct way to get `LoginPath` is to use the options pattern's built-in support for DI:

```
public void ConfigureServices(IServiceCollection services)
{

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationSchem
e)
        .AddCookie();

    services.AddOptions<CookieAuthenticationOptions>(
                        CookieAuthenticationDefaults.AuthenticationScheme)
        .Configure<IMyService>((options, myService) =>
        {
            options.LoginPath = myService.GetLoginPath();
        });

    services.AddRazorPages();
}
```

- Disposable transient services are captured by the container for disposal. This can turn into a memory leak if resolved from the top level container.
- Enable scope validation to make sure the app doesn't have singletons that capture scoped services. For more information, see Scope validation.

Like all sets of recommendations, you may encounter situations where ignoring a recommendation is required. Exceptions are rare, mostly special cases within the framework itself.

DI is an *alternative* to static/global object access patterns. You may not be able to realize the benefits of DI if you mix it with static object access.

Recommended patterns for multi-tenancy in DI

Orchard Core is an application framework for building modular, multi-tenant applications on ASP.NET Core. For more information, see the Orchard Core Documentation.

See the Orchard Core samples for examples of how to build modular and multi-tenant apps using just the Orchard Core Framework without any of its CMS-specific features.

Framework-provided services

The `Startup.ConfigureServices` method registers services that the app uses, including platform features, such as Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `ConfigureServices` has services defined by the framework depending on how the host was configured. For apps based on the ASP.NET Core templates, the framework registers more than 250 services.

The following table lists a small sample of these framework-registered services:

SERVICE TYPE	LIFETIME
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	Transient
IHostApplicationLifetime	Singleton
IWebHostEnvironment	Singleton

SERVICE TYPE	LIFETIME
Microsoft.AspNetCore.Hosting.IStartup	Singleton
Microsoft.AspNetCore.Hosting.IStartupFilter	Transient
Microsoft.AspNetCore.Hosting.Server.IServer	Singleton
Microsoft.AspNetCore.Http.IHttpContextFactory	Transient
Microsoft.Extensions.Logging.ILogger<TCategoryName>	Singleton
Microsoft.Extensions.Logging.ILoggerFactory	Singleton
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<TOptions>	Transient
Microsoft.Extensions.Options.IOptions<TOptions>	Singleton
System.Diagnostics.DiagnosticSource	Singleton
System.Diagnostics.DiagnosticListener	Singleton

Additional resources

- Dependency injection into views in ASP.NET Core
- Dependency injection into controllers in ASP.NET Core
- Dependency injection in requirement handlers in ASP.NET Core
- ASP.NET Core Blazor dependency injection
- NDC Conference Patterns for DI app development
- App startup in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Four ways to dispose IDisposables in ASP.NET Core
- Writing Clean Code in ASP.NET Core with Dependency Injection (MSDN)
- Explicit Dependencies Principle
- Inversion of Control Containers and the Dependency Injection Pattern (Martin Fowler)
- How to register a service with multiple interfaces in ASP.NET Core DI

By Steve Smith, Scott Addie, and Brandon Dahler

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

For more information specific to dependency injection within MVC controllers, see Dependency injection into controllers in ASP.NET Core.

View or download sample code (how to download)

Overview of dependency injection

A *dependency* is any object that another object requires. Examine the following `MyDependency` class with a `WriteMessage` method that other classes in an app depend upon:

```
public class MyDependency
{
    public MyDependency()
    {
    }

    public Task WriteMessage(string message)
    {
        Console.WriteLine(
            $"MyDependency.WriteMessage called. Message: {message}");

        return Task.FromResult(0);
    }
}
```

An instance of the `MyDependency` class can be created to make the `WriteMessage` method available to a class. The `MyDependency` class is a dependency of the `IndexModel` class:

```
public class IndexModel : PageModel
{
    MyDependency _dependency = new MyDependency();

    public async Task OnGetAsync()
    {
        await _dependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

The class creates and directly depends on the `MyDependency` instance. Code dependencies (such as the previous example) are problematic and should be avoided for the following reasons:

- To replace `MyDependency` with a different implementation, the class must be modified.
- If `MyDependency` has dependencies, they must be configured by the class. In a large project with multiple classes depending on `MyDependency`, the configuration code becomes scattered across the app.
- This implementation is difficult to unit test. The app should use a mock or stub `MyDependency` class, which isn't possible with this approach.

Dependency injection addresses these problems through:

- The use of an interface or base class to abstract the dependency implementation.
- Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, IServiceProvider. Services are registered in the app's `Startup.ConfigureServices` method.
- *Injection* of the service into the constructor of the class where it's used. The

framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

In the sample app, the `IMyDependency` interface defines a method that the service provides to the app:

```
public interface IMyDependency
{
    Task WriteMessage(string message);
}
```

This interface is implemented by a concrete type, `MyDependency`:

```
public class MyDependency : IMyDependency
{
    private readonly ILogger<MyDependency> _logger;

    public MyDependency(ILogger<MyDependency> logger)
    {
        _logger = logger;
    }

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {Message}",
            message);

        return Task.FromResult(0);
    }
}
```

`MyDependency` requests an ILogger<TCategoryName> in its constructor. It's not unusual to use dependency injection in a chained fashion. Each requested dependency in turn requests its own dependencies. The container resolves the dependencies in the graph and returns the fully resolved service. The collective set of dependencies that must be resolved is typically referred to as a *dependency tree*, *dependency graph*, or *object graph*.

`IMyDependency` and `ILogger<TCategoryName>` must be registered in the service container. `IMyDependency` is registered in `Startup.ConfigureServices`. `ILogger<TCategoryName>` is registered by the logging abstractions infrastructure, so it's a framework-provided service registered by default by the framework.

The container resolves `ILogger<TCategoryName>` by taking advantage of (generic) open types, eliminating the need to register every (generic) constructed type:

```
services.AddSingleton(typeof(ILogger<>), typeof(Logger<>));
```

In the sample app, the `IMyDependency` service is registered with the concrete type `MyDependency`. The registration scopes the service lifetime to the lifetime of a single request. Service lifetimes are described later in this topic.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

NOTE

Each `services.Add{SERVICE_NAME}` extension method adds (and potentially configures) services. For example, `services.AddMvc()` adds the services Razor Pages and MVC require. We recommended that apps follow this convention. Place extension methods in the Microsoft.Extensions.DependencyInjection namespace to encapsulate groups of service registrations.

If the service's constructor requires a built in type, such as a `string`, the type can be injected by using configuration or the options pattern:

```
public class MyDependency : IMyDependency
{
    public MyDependency(IConfiguration config)
    {
        var myStringValue = config["MyStringKey"];

        // Use myStringValue
    }

    ...
}
```

An instance of the service is requested via the constructor of a class where the service is used and assigned to a private field. The field is used to access the service as necessary throughout the class.

In the sample app, the `IMyDependency` instance is requested and used to call the service's `WriteMessage` method:

```
public class IndexModel : PageModel
{
    private readonly IMyDependency _myDependency;

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

Services injected into Startup

Only the following service types can be injected into the `Startup` constructor when using the Generic Host (IHostBuilder):

- `IWebHostEnvironment`
- IHostEnvironment
- IConfiguration

Services can be injected into `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IOptions<MyOptions> options)
{
    ...
}
```

For more information, see App startup in ASP.NET Core.

Framework-provided services

The `Startup.ConfigureServices` method is responsible for defining the services that the app uses, including platform features, such as Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `ConfigureServices` has services defined by the framework depending on how the host was configured. It's not uncommon for an app based on an ASP.NET Core template to have hundreds of services registered by the framework. A small sample of framework-registered

services is listed in the following table.

SERVICE TYPE	LIFETIME
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	Transient
Microsoft.AspNetCore.Hosting.IApplicationLifetime	Singleton
Microsoft.AspNetCore.Hosting.IHostingEnvironment	Singleton
Microsoft.AspNetCore.Hosting.IStartup	Singleton
Microsoft.AspNetCore.Hosting.IStartupFilter	Transient
Microsoft.AspNetCore.Hosting.Server.IServer	Singleton
Microsoft.AspNetCore.Http.IHttpContextFactory	Transient
Microsoft.Extensions.Logging.ILogger<TCategoryName>	Singleton
Microsoft.Extensions.Logging.ILoggerFactory	Singleton
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<TOptions>	Transient
Microsoft.Extensions.Options.IOptions<TOptions>	Singleton
System.Diagnostics.DiagnosticSource	Singleton
System.Diagnostics.DiagnosticListener	Singleton

Register additional services with extension methods

When a service collection extension method is available to register a service (and its dependent services, if required), the convention is to use a single `Add{SERVICE_NAME}` extension method to register all of the services required by that service. The following code is an example of how to add additional services to the container using the extension methods AddDbContext<TContext> and AddIdentityCore:

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    ...
}
```

For more information, see the ServiceCollection class in the API documentation.

Service lifetimes

Choose an appropriate lifetime for each registered service. ASP.NET Core services can be configured with the following lifetimes:

Transient

Transient lifetime services (AddTransient) are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services.

In apps that process requests, transient services are disposed at the end of the request.

Scoped

Scoped lifetime services (AddScoped) are created once per client request (connection).

In apps that process requests, scoped services are disposed at the end of the request.

> **WARNING**
>
> When using a scoped service in a middleware, inject the service into the `Invoke` or `InvokeAsync` method. Don't inject via constructor injection because it forces the service to behave like a singleton. For more information, see Write custom ASP.NET Core middleware.

Singleton

Singleton lifetime services (AddSingleton) are created the first time they're requested (or when `Startup.ConfigureServices` is run and an instance is specified with the service registration). Every subsequent request uses the same instance. If the app requires singleton behavior, allowing the service container to manage the service's lifetime is recommended. Don't implement the singleton design pattern and provide user code to manage the object's lifetime in the class.

In apps that process requests, singleton services are disposed when the ServiceProvider is disposed at app shutdown.

> **WARNING**
>
> It's dangerous to resolve a scoped service from a singleton. It may cause the service to have incorrect state when processing subsequent requests.

Service registration methods

Service registration extension methods offer overloads that are useful in specific scenarios.

METHOD	AUTOMATIC OBJECT DISPOSAL	MULTIPLE IMPLEMENTATIONS	PASS ARGS
`Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()` Example: `services.AddSingleton<IMyDep, MyDep>();`	Yes	Yes	No
`Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION})` Examples: `services.AddSingleton<IMyDep>(sp => new MyDep());` `services.AddSingleton<IMyDep>(sp => new MyDep("A string!"));`	Yes	Yes	Yes
`Add{LIFETIME}<{IMPLEMENTATION}>()` Example: `services.AddSingleton<MyDep>();`	Yes	No	No
`AddSingleton<{SERVICE}>(new {IMPLEMENTATION})` Examples: `services.AddSingleton<IMyDep>(new MyDep());` `services.AddSingleton<IMyDep>(new MyDep("A string!"));`	No	Yes	Yes
`AddSingleton(new {IMPLEMENTATION})` Examples: `services.AddSingleton(new MyDep());` `services.AddSingleton(new MyDep("A string!"));`	No	No	Yes

For more information on type disposal, see the Disposal of services section. A common scenario for multiple implementations is mocking types for testing.

`TryAdd{LIFETIME}` methods only register the service if there isn't already an implementation registered.

In the following example, the first line registers `MyDependency` for `IMyDependency`. The second line has no effect because `IMyDependency` already has a registered implementation:

```
services.AddSingleton<IMyDependency, MyDependency>();
// The following line has no effect:
services.TryAddSingleton<IMyDependency, DifferentDependency>();
```

For more information, see:

- TryAdd
- TryAddTransient
- TryAddScoped
- TryAddSingleton

TryAddEnumerable(ServiceDescriptor) methods only register the service if there isn't already an implementation *of the same type*. Multiple services are resolved via `IEnumerable<{SERVICE}>`. When registering services, the developer only wants to add an instance if one of the same type hasn't already been added. Generally, this method is used by library authors to avoid registering two copies of an instance in the container.

In the following example, the first line registers `MyDep` for `IMyDep1`. The second line registers `MyDep` for `IMyDep2`. The third line has no effect because `IMyDep1` already has a registered implementation of `MyDep`:

```
public interface IMyDep1 {}
public interface IMyDep2 {}

public class MyDep : IMyDep1, IMyDep2 {}

services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep1, MyDep>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep2, MyDep>());
// Two registrations of MyDep for IMyDep1 is avoided by the following line:
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMyDep1, MyDep>());
```

Constructor injection behavior

Services can be resolved by two mechanisms:

- IServiceProvider
- ActivatorUtilities: Permits object creation without service registration in the dependency injection container. `ActivatorUtilities` is used with user-facing abstractions, such as Tag Helpers, MVC controllers, and model binders.

Constructors can accept arguments that aren't provided by dependency injection, but the arguments must assign default values.

When services are resolved by `IServiceProvider` or `ActivatorUtilities`, constructor injection requires a *public* constructor.

When services are resolved by `ActivatorUtilities`, constructor injection requires that only one applicable constructor exists. Constructor overloads are supported, but only one overload can exist whose arguments can all be fulfilled by dependency injection.

Entity Framework contexts

Entity Framework contexts are usually added to the service container using the scoped lifetime because web app database operations are normally scoped to the client request. The default lifetime is scoped if a lifetime isn't specified by an

AddDbContext<TContext> overload when registering the database context. Services of a given lifetime shouldn't use a database context with a shorter lifetime than the service.

Lifetime and registration options

To demonstrate the difference between the lifetime and registration options, consider the following interfaces that represent tasks as an operation with a unique identifier, `OperationId`. Depending on how the lifetime of an operations service is configured for the following interfaces, the container provides either the same or a different instance of the service when requested by a class:

```
public interface IOperation
{
    Guid OperationId { get; }
}

public interface IOperationTransient : IOperation
{
}

public interface IOperationScoped : IOperation
{
}

public interface IOperationSingleton : IOperation
{
}

public interface IOperationSingletonInstance : IOperation
{
}
```

The interfaces are implemented in the `Operation` class. The `Operation` constructor generates a GUID if one isn't supplied:

```
public class Operation : IOperationTransient,
    IOperationScoped,
    IOperationSingleton,
    IOperationSingletonInstance
{
    public Operation() : this(Guid.NewGuid())
    {
    }

    public Operation(Guid id)
    {
        OperationId = id;
    }

    public Guid OperationId { get; private set; }
}
```

An `OperationService` is registered that depends on each of the other `Operation` types. When `OperationService` is requested via dependency injection, it receives either a new instance of each service or an existing instance based on the lifetime of the dependent service.

- When transient services are created when requested from the container, the `OperationId` of the `IOperationTransient` service is different than the `OperationId`

of the `OperationService`. `OperationService` receives a new instance of the `IOperationTransient` class. The new instance yields a different `OperationId`.

- When scoped services are created per client request, the `OperationId` of the `IOperationScoped` service is the same as that of `OperationService` within a client request. Across client requests, both services share a different `OperationId` value.
- When singleton and singleton-instance services are created once and used across all client requests and all services, the `OperationId` is constant across all service requests.

```
public class OperationService
{
    public OperationService(
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance instanceOperation)
    {
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = instanceOperation;
    }

    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }
}
```

In `Startup.ConfigureServices`, each type is added to the container according to its named lifetime:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

The `IOperationSingletonInstance` service is using a specific instance with a known ID of `Guid.Empty`. It's clear when this type is in use (its GUID is all zeroes).

The sample app demonstrates object lifetimes within and between individual requests. The sample app's `IndexModel` requests each kind of `IOperation` type and the `OperationService`. The page then displays all of the page model class's and service's `OperationId` values through property assignments:

```
public class IndexModel : PageModel
{
    private readonly IMyDependency _myDependency;

    public IndexModel(
        IMyDependency myDependency,
        OperationService operationService,
        IOperationTransient transientOperation,
        IOperationScoped scopedOperation,
        IOperationSingleton singletonOperation,
        IOperationSingletonInstance singletonInstanceOperation)
    {
        _myDependency = myDependency;
        OperationService = operationService;
        TransientOperation = transientOperation;
        ScopedOperation = scopedOperation;
        SingletonOperation = singletonOperation;
        SingletonInstanceOperation = singletonInstanceOperation;
    }

    public OperationService OperationService { get; }
    public IOperationTransient TransientOperation { get; }
    public IOperationScoped ScopedOperation { get; }
    public IOperationSingleton SingletonOperation { get; }
    public IOperationSingletonInstance SingletonInstanceOperation { get; }

    public async Task OnGetAsync()
    {
        await _myDependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```

Two following output shows the results of two requests:

First request:

Controller operations:

Transient: d233e165-f417-469b-a866-1cf1935d2518
Scoped: 5d997e2d-55f5-4a64-8388-51c4e3a1ad19
Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

`OperationService` operations:

Transient: c6b049eb-1318-4e31-90f1-eb2dd849ff64
Scoped: 5d997e2d-55f5-4a64-8388-51c4e3a1ad19
Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

Second request:

Controller operations:

Transient: b63bd538-0a37-4ff1-90ba-081c5138dda0
Scoped: 31e820c5-4834-4d22-83fc-a60118acb9f4
Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

`OperationService` operations:

Transient: c4cbacb8-36a2-436d-81c8-8c1b78808aaf

Scoped: 31e820c5-4834-4d22-83fc-a60118acb9f4
Singleton: 01271bc1-9e31-48e7-8f7c-7261b040ded9
Instance: 00000000-0000-0000-0000-000000000000

Observe which of the `OperationId` values vary within a request and between requests:

- *Transient* objects are always different. The transient `OperationId` value for both the first and second client requests are different for both `OperationService` operations and across client requests. A new instance is provided to each service request and client request.
- *Scoped* objects are the same within a client request but different across client requests.
- *Singleton* objects are the same for every object and every request regardless of whether an `Operation` instance is provided in `Startup.ConfigureServices`.

Call services from main

Create an IServiceScope with IServiceScopeFactory.CreateScope to resolve a scoped service within the app's scope. This approach is useful to access a scoped service at startup to run initialization tasks. The following example shows how to obtain a context for the `MyScopedService` in `Program.Main`:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

public class Program
{
    public static async Task Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var serviceScope = host.Services.CreateScope())
        {
            var services = serviceScope.ServiceProvider;

            try
            {
                var serviceContext = services.GetRequiredService<MyScopedService>
();
                // Use the context here
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred.");
            }
        }

        await host.RunAsync();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Scope validation

When the app is running in the Development environment, the default service provider performs checks to verify that:

- Scoped services aren't directly or indirectly resolved from the root service provider.
- Scoped services aren't directly or indirectly injected into singletons.

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app/server's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton because it's only disposed by the root container when app/server is shut down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

For more information, see ASP.NET Core Web Host.

Request Services

The services available within an ASP.NET Core request from `HttpContext` are exposed through the HttpContext.RequestServices collection.

Request Services represent the services configured and requested as part of the app. When the objects specify dependencies, these are satisfied by the types found in `RequestServices`, not `ApplicationServices`.

Generally, the app shouldn't use these properties directly. Instead, request the types that classes require via class constructors and allow the framework inject the dependencies. This yields classes that are easier to test.

> **NOTE**
>
> Prefer requesting dependencies as constructor parameters to accessing the `RequestServices` collection.

Design services for dependency injection

Best practices are to:

- Design services to use dependency injection to obtain their dependencies.
- Avoid stateful, static classes and members. Design apps to use singleton services instead, which avoid creating global state.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make app classes small, well-factored, and easily tested.

If a class seems to have too many injected dependencies, this is generally a sign that the class has too many responsibilities and is violating the Single Responsibility Principle (SRP). Attempt to refactor the class by moving some of its responsibilities into a new class. Keep in mind that Razor Pages page model classes and MVC controller classes should focus on UI concerns. Business rules and data access implementation details should be kept in classes appropriate to these separate

concerns.

Disposal of services

The container calls Dispose for the IDisposable types it creates. If an instance is added to the container by user code, it isn't disposed automatically.

In the following example, the services are created by the service container and disposed automatically:

```
public class Service1 : IDisposable {}
public class Service2 : IDisposable {}

public interface IService3 {}
public class Service3 : IService3, IDisposable {}

public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<Service1>();
    services.AddSingleton<Service2>();
    services.AddSingleton<IService3>(sp => new Service3());
}
```

In the following example:

- The service instances aren't created by the service container.
- The intended service lifetimes aren't known by the framework.
- The framework doesn't dispose of the services automatically.
- If the services aren't explicitly disposed in developer code, they persist until the app shuts down.

```
public class Service1 : IDisposable {}
public class Service2 : IDisposable {}

public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<Service1>(new Service1());
    services.AddSingleton(new Service2());
}
```

IDisposable guidance for Transient and shared instances

Transient, limited lifetime

Scenario

The app requires an IDisposable instance with a transient lifetime for either of the following scenarios:

- The instance is resolved in the root scope.
- The instance should be disposed before the scope ends.

Solution

Use the factory pattern to create an instance outside of the parent scope. In this situation, the app would generally have a `Create` method that calls the final type's constructor directly. If the final type has other dependencies, the factory can:

- Receive an IServiceProvider in its constructor.
- Use ActivatorUtilities.CreateInstance to instantiate the instance outside the container, while using the container for its dependencies.

Shared Instance, limited lifetime

Scenario

The app requires a shared IDisposable instance across multiple services, but the IDisposable should have a limited lifetime.

Solution

Register the instance with a Scoped lifetime. Use IServiceScopeFactory.CreateScope to start and create a new IServiceScope. Use the scope's IServiceProvider to get required services. Dispose the scope when the lifetime should end.

General Guidelines

- Don't register IDisposable instances with a Transient scope. Use the factory pattern instead.
- Don't resolve Transient or Scoped IDisposable instances in the root scope. The only general exception is when the app creates/recreates and disposes the IServiceProvider, which isn't an ideal pattern.
- Receiving an IDisposable dependency via DI doesn't require that the receiver implement IDisposable itself. The receiver of the IDisposable dependency shouldn't call Dispose on that dependency.
- Scopes should be used to control lifetimes of services. Scopes aren't hierarchical, and there's no special connection among scopes.

Default service container replacement

The built-in service container is designed to serve the needs of the framework and most consumer apps. We recommend using the built-in container unless you need a specific feature that the built-in container doesn't support, such as:

- Property injection
- Injection based on name
- Child containers
- Custom lifetime management
- `Func<T>` support for lazy initialization
- Convention-based registration

The following third-party containers can be used with ASP.NET Core apps:

- Autofac
- Dryloc
- Grace
- LightInject
- Lamar
- Stashbox
- Unity

Thread safety

Create thread-safe singleton services. If a singleton service has a dependency on a transient service, the transient service may also require thread safety depending how it's used by the singleton.

The factory method of single service, such as the second argument to AddSingleton<TService>(IServiceCollection, Func<IServiceProvider,TService>),

doesn't need to be thread-safe. Like a type (`static`) constructor, it's guaranteed to be called once by a single thread.

Recommendations

- `async/await` and `Task` based service resolution is not supported. C# does not support asynchronous constructors; therefore, the recommended pattern is to use asynchronous methods after synchronously resolving the service.
- Avoid storing data and configuration directly in the service container. For example, a user's shopping cart shouldn't typically be added to the service container. Configuration should use the options pattern. Similarly, avoid "data holder" objects that only exist to allow access to some other object. It's better to request the actual item via DI.
- Avoid static access to services. For example, avoid statically-typing IApplicationBuilder.ApplicationServices for use elsewhere.
- Avoid using the *service locator pattern*, which mixes Inversion of Control strategies.
  - Don't invoke GetService to obtain a service instance when you can use DI instead:

    Incorrect:

    ```
    public class MyClass()

      public void MyMethod()
      {
          var optionsMonitor =
              _services.GetService<IOptionsMonitor<MyOptions>>();
          var option = optionsMonitor.CurrentValue.Option;

          ...
      }
    ```

    Correct:

    ```
    public class MyClass
    {
        private readonly IOptionsMonitor<MyOptions> _optionsMonitor;

        public MyClass(IOptionsMonitor<MyOptions> optionsMonitor)
        {
            _optionsMonitor = optionsMonitor;
        }

        public void MyMethod()
        {
            var option = _optionsMonitor.CurrentValue.Option;

            ...
        }
    }
    ```

- Avoid injecting a factory that resolves dependencies at runtime using GetService.
- Avoid static access to `HttpContext` (for example,

IHttpContextAccessor.HttpContext).

Like all sets of recommendations, you may encounter situations where ignoring a recommendation is required. Exceptions are rare, mostly special cases within the framework itself.

DI is an *alternative* to static/global object access patterns. You may not be able to realize the benefits of DI if you mix it with static object access.

Additional resources

- Dependency injection into views in ASP.NET Core
- Dependency injection into controllers in ASP.NET Core
- Dependency injection in requirement handlers in ASP.NET Core
- ASP.NET Core Blazor dependency injection
- App startup in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Four ways to dispose IDisposables in ASP.NET Core
- Writing Clean Code in ASP.NET Core with Dependency Injection (MSDN)
- Explicit Dependencies Principle
- Inversion of Control Containers and the Dependency Injection Pattern (Martin Fowler)
- How to register a service with multiple interfaces in ASP.NET Core DI

ASP.NET Core Middleware

9/22/2020 • 22 minutes to read • Edit Online

By Rick Anderson and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

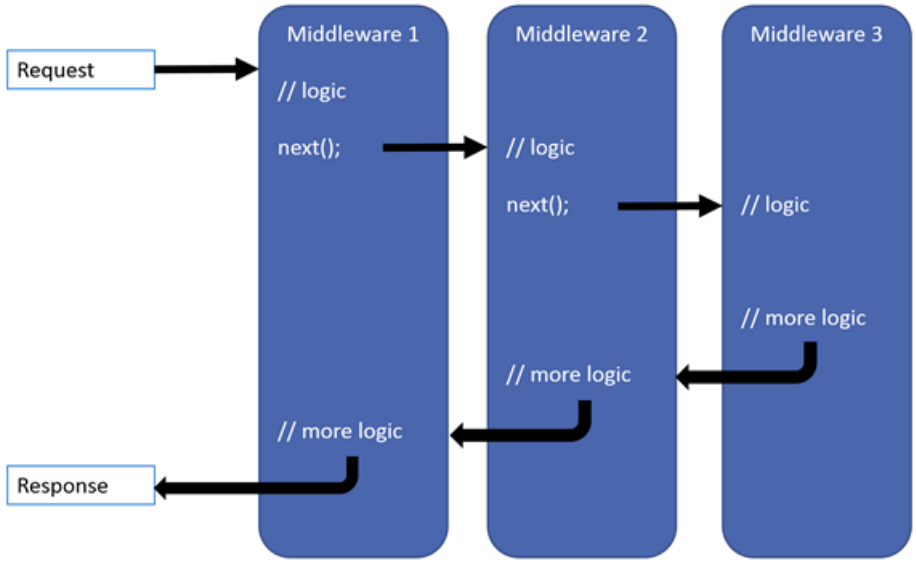
Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, also called *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline. When a middleware short-circuits, it's called a *terminal middleware* because it prevents further middleware from processing the request.

Migrate HTTP handlers and modules to ASP.NET Core middleware explains the difference between request pipelines in ASP.NET Core and ASP.NET 4.x and provides additional middleware samples.

Create a middleware pipeline with IApplicationBuilder

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



Each delegate can perform operations before and after the next delegate. Exception-handling delegates should be called early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This

case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request.

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

Chain multiple request delegates together with Use. The `next` parameter represents the next delegate in the pipeline. You can short-circuit the pipeline by *not* calling the *next* parameter. You can typically perform actions both before and after the next delegate, as the following example demonstrates:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

When a delegate doesn't pass a request to the next delegate, it's called *short-circuiting the request pipeline*. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static File Middleware can act as a *terminal middleware* by processing a request for a static file and short-circuiting the rest of the pipeline. Middleware added to the pipeline before the middleware that terminates further processing still processes code after their `next.Invoke` statements. However, see the following warning about attempting to write to a response that has already been sent.

> **WARNING**
>
> Don't call `next.Invoke` after the response has been sent to the client. Changes to HttpResponse after the response has started throw an exception. For example, setting headers and a status code throw an exception. Writing to the response body after calling `next`:
>
> - May cause a protocol violation. For example, writing more than the stated `Content-Length`.
> - May corrupt the body format. For example, writing an HTML footer to a CSS file.
>
> HasStarted is a useful hint to indicate if headers have been sent or the body has been written to.

Run delegates don't receive a `next` parameter. The first `Run` delegate is always terminal and terminates the pipeline. `Run` is a convention. Some middleware components may expose `Run[Middleware]` methods that run at the end of the pipeline:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

In the preceding example, the `Run` delegate writes `"Hello from 2nd delegate."` to the response and then terminates the pipeline. If another `Use` or `Run` delegate is added after the `Run` delegate, it's not called.

Middleware order

The following diagram shows the complete request processing pipeline for ASP.NET Core MVC and Razor Pages apps. You can see how, in a typical app, existing middlewares are ordered and where custom middlewares are added. You have full control over how to reorder existing middlewares or inject new custom middlewares as necessary for your scenarios.

The **Endpoint** middleware in the preceding diagram executes the filter pipeline for the corresponding app type—MVC or Razor Pages.

The order that middleware components are added in the `Startup.Configure` method defines the order in which the middleware components are invoked on requests and the reverse order for the response. The order is **critical** for security, performance, and functionality.

The following `Startup.Configure` method adds security-related middleware components in the recommended order:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    // app.UseCookiePolicy();

    app.UseRouting();
    // app.UseRequestLocalization();
    // app.UseCors();

    app.UseAuthentication();
    app.UseAuthorization();
    // app.UseSession();
    // app.UseResponseCaching();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

In the preceding code:

- Middleware that is not added when creating a new web app with individual users accounts is commented out.
- Not every middleware needs to go in this exact order, but many do. For example:
  - `UseCors`, `UseAuthentication`, and `UseAuthorization` must go in the order shown.
  - `UseCors` currently must go before `UseResponseCaching` due to this bug.

The following `Startup.Configure` method adds middleware components for common app scenarios:

1. Exception/error handling
   - When the app runs in the Development environment:
     - Developer Exception Page Middleware (UseDeveloperExceptionPage) reports app runtime errors.
     - Database Error Page Middleware reports database runtime errors.
   - When the app runs in the Production environment:
     - Exception Handler Middleware (UseExceptionHandler) catches exceptions thrown in the following middlewares.
     - HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) adds the `Strict-Transport-Security` header.
2. HTTPS Redirection Middleware (UseHttpsRedirection) redirects HTTP requests to HTTPS.
3. Static File Middleware (UseStaticFiles) returns static files and short-circuits further request processing.
4. Cookie Policy Middleware (UseCookiePolicy) conforms the app to the EU General Data Protection Regulation (GDPR) regulations.

5. Routing Middleware (UseRouting) to route requests.
6. Authentication Middleware (UseAuthentication) attempts to authenticate the user before they're allowed access to secure resources.
7. Authorization Middleware (UseAuthorization) authorizes a user to access secure resources.
8. Session Middleware (UseSession) establishes and maintains session state. If the app uses session state, call Session Middleware after Cookie Policy Middleware and before MVC Middleware.
9. Endpoint Routing Middleware (UseEndpoints with MapRazorPages) to add Razor Pages endpoints to the request pipeline.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseSession();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

In the preceding example code, each middleware extension method is exposed on IApplicationBuilder through the Microsoft.AspNetCore.Builder namespace.

UseExceptionHandler is the first middleware component added to the pipeline. Therefore, the Exception Handler Middleware catches any exceptions that occur in later calls.

Static File Middleware is called early in the pipeline so that it can handle requests and short-circuit without going through the remaining components. The Static File Middleware provides **no** authorization checks. Any files served by Static File Middleware, including those under *wwwroot*, are publicly available. For an approach to secure static files, see Static files in ASP.NET Core.

If the request isn't handled by the Static File Middleware, it's passed on to the Authentication Middleware (UseAuthentication), which performs authentication. Authentication doesn't short-circuit unauthenticated requests. Although Authentication Middleware authenticates requests, authorization (and rejection) occurs only after MVC selects a specific Razor Page or MVC controller and action.

The following example demonstrates a middleware order where requests for static files are handled by Static File Middleware before Response Compression Middleware. Static files aren't compressed with this middleware order. The Razor Pages responses can be compressed.

```
public void Configure(IApplicationBuilder app)
{
    // Static files aren't compressed by Static File Middleware.
    app.UseStaticFiles();

    app.UseResponseCompression();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

For Single Page Applications (SPAs), the SPA middleware UseSpaStaticFiles usually comes last in the middleware pipeline. The SPA middleware comes last:

- To allow all other middlewares to respond to matching requests first.
- To allow SPAs with client-side routing to run for all routes that are unrecognized by the server app.

For more details on SPAs, see the guides for the React and Angular project templates.

Forwarded Headers Middleware order

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Branch the middleware pipeline

Map extensions are used as a convention for branching the pipeline. `Map` branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

```
public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code.

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

When `Map` is used, the matched path segments are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

`Map` supports nesting, for example:

```
app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a" processing
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b" processing
    });
});
```

`Map` can also match multiple segments at once:

```
public class Startup
{
    private static void HandleMultiSeg(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map multiple segments.");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1/seg1", HandleMultiSeg);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate.");
        });
    }
}
```

MapWhen branches the request pipeline based on the result of the given predicate. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a predicate is used to detect the presence of a query string variable `branch`:

```
public class Startup
{
    private static void HandleBranch(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            var branchVer = context.Request.Query["branch"];
            await context.Response.WriteAsync($"Branch used = {branchVer}");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
                               HandleBranch);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code:

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/?branch=master	Branch used = master

UseWhen also branches the request pipeline based on the result of the given predicate. Unlike with `MapWhen`, this branch is rejoined to the main pipeline if it doesn't short-circuit or contain a terminal middleware:

```
public class Startup
{
    private readonly ILogger<Startup> _logger;

    public Startup(ILogger<Startup> logger)
    {
        _logger = logger;
    }

    private void HandleBranchAndRejoin(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            var branchVer = context.Request.Query["branch"];
            _logger.LogInformation("Branch used = {branchVer}", branchVer);

            // Do work that doesn't write to the Response.
            await next();
            // Do other work that doesn't write to the Response.
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseWhen(context => context.Request.Query.ContainsKey("branch"),
                               HandleBranchAndRejoin);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from main pipeline.");
        });
    }
}
```

In the preceding example, a response of "Hello from main pipeline." is written for all requests. If the request includes a query string variable `branch`, its value is logged before the main pipeline is rejoined.

Built-in middleware

ASP.NET Core ships with the following middleware components. The *Order* column provides notes on middleware placement in the request processing pipeline and under what conditions the middleware may terminate request processing. When a middleware short-circuits the request processing pipeline and prevents further downstream middleware from processing a request, it's called a *terminal middleware*. For more information on short-circuiting, see the Create a middleware pipeline with IApplicationBuilder section.

MIDDLEWARE	DESCRIPTION	ORDER
Authentication	Provides authentication support.	Before `HttpContext.User` is needed. Terminal for OAuth callbacks.
Authorization	Provides authorization support.	Immediately after the Authentication Middleware.
Cookie Policy	Tracks consent from users for storing personal information and enforces minimum standards for cookie fields, such as `secure` and `SameSite`.	Before middleware that issues cookies. Examples: Authentication, Session, MVC (TempData).

MIDDLEWARE	DESCRIPTION	ORDER
CORS	Configures Cross-Origin Resource Sharing.	Before components that use CORS. `UseCors` currently must go before `UseResponseCaching` due to this bug.
Diagnostics	Several separate middlewares that provide a developer exception page, exception handling, status code pages, and the default web page for new apps.	Before components that generate errors. Terminal for exceptions or serving the default web page for new apps.
Forwarded Headers	Forwards proxied headers onto the current request.	Before components that consume the updated fields. Examples: scheme, host, client IP, method.
Health Check	Checks the health of an ASP.NET Core app and its dependencies, such as checking database availability.	Terminal if a request matches a health check endpoint.
Header Propagation	Propagates HTTP headers from the incoming request to the outgoing HTTP Client requests.	
HTTP Method Override	Allows an incoming POST request to override the method.	Before components that consume the updated method.
HTTPS Redirection	Redirect all HTTP requests to HTTPS.	Before components that consume the URL.
HTTP Strict Transport Security (HSTS)	Security enhancement middleware that adds a special response header.	Before responses are sent and after components that modify requests. Examples: Forwarded Headers, URL Rewriting.
MVC	Processes requests with MVC/Razor Pages.	Terminal if a request matches a route.
OWIN	Interop with OWIN-based apps, servers, and middleware.	Terminal if the OWIN Middleware fully processes the request.
Response Caching	Provides support for caching responses.	Before components that require caching. `UseCORS` must come before `UseResponseCaching`.
Response Compression	Provides support for compressing responses.	Before components that require compression.
Request Localization	Provides localization support.	Before localization sensitive components.
Endpoint Routing	Defines and constrains request routes.	Terminal for matching routes.

MIDDLEWARE	DESCRIPTION	ORDER
SPA	Handles all requests from this point in the middleware chain by returning the default page for the Single Page Application (SPA)	Late in the chain, so that other middleware for serving static files, MVC actions, etc., takes precedence.
Session	Provides support for managing user sessions.	Before components that require Session.
Static Files	Provides support for serving static files and directory browsing.	Terminal if a request matches a file.
URL Rewrite	Provides support for rewriting URLs and redirecting requests.	Before components that consume the URL.
WebSockets	Enables the WebSockets protocol.	Before components that are required to accept WebSocket requests.

Additional resources

- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- Write custom ASP.NET Core middleware
- Test ASP.NET Core middleware
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

By Rick Anderson and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

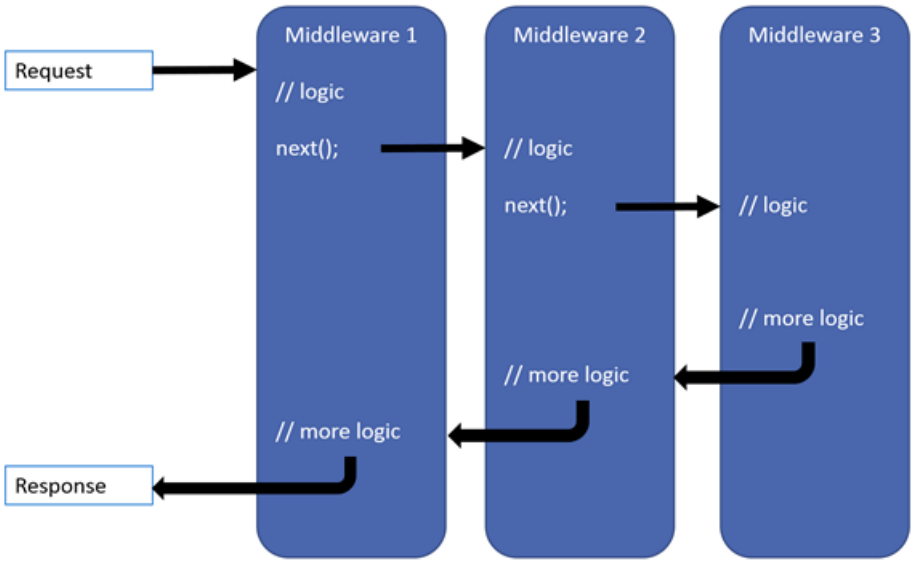
Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, also called *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline. When a middleware short-circuits, it's called a *terminal middleware* because it prevents further middleware from processing the request.

Migrate HTTP handlers and modules to ASP.NET Core middleware explains the difference between request pipelines in ASP.NET Core and ASP.NET 4.x and provides additional middleware samples.

Create a middleware pipeline with IApplicationBuilder

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



Each delegate can perform operations before and after the next delegate. Exception-handling delegates should be called early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request.

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

The first Run delegate terminates the pipeline.

Chain multiple request delegates together with Use. The `next` parameter represents the next delegate in the pipeline. You can short-circuit the pipeline by *not* calling the *next* parameter. You can typically perform actions both before and after the next delegate, as the following example demonstrates:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

When a delegate doesn't pass a request to the next delegate, it's called *short-circuiting the request pipeline*. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static File Middleware can act as a *terminal middleware* by processing a request for a static file and short-circuiting the rest of the pipeline. Middleware added to the pipeline before the middleware that terminates further processing still processes code after their `next.Invoke` statements. However, see the following warning about attempting to write to a response that has already been sent.

WARNING

Don't call `next.Invoke` after the response has been sent to the client. Changes to HttpResponse after the response has started throw an exception. For example, changes such as setting headers and a status code throw an exception. Writing to the response body after calling `next`:

- May cause a protocol violation. For example, writing more than the stated `Content-Length`.
- May corrupt the body format. For example, writing an HTML footer to a CSS file.

HasStarted is a useful hint to indicate if headers have been sent or the body has been written to.

Middleware order

The order that middleware components are added in the `Startup.Configure` method defines the order in which the middleware components are invoked on requests and the reverse order for the response. The order is **critical** for security, performance, and functionality.

The following `Startup.Configure` method adds security related middleware components in the recommended order:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    // app.UseRequestLocalization();
    // app.UseCors();

    app.UseAuthentication();
    // app.UseSession();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

In the preceding code:

- Middleware that is not added when creating a new web app with individual users accounts is commented out.
- Not every middleware needs to go in this exact order, but many do. For example, `UseCors` and `UseAuthentication` must go in the order shown.

The following `Startup.Configure` method adds middleware components for common app scenarios:

1. Exception/error handling
   - When the app runs in the Development environment:
     - Developer Exception Page Middleware (UseDeveloperExceptionPage) reports app runtime errors.
     - Database Error Page Middleware (`Microsoft.AspNetCore.Builder.DatabaseErrorPageExtensions.UseDatabaseErrorPage`) reports database runtime errors.
   - When the app runs in the Production environment:
     - Exception Handler Middleware (UseExceptionHandler) catches exceptions thrown in the following middlewares.
     - HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) adds the `Strict-Transport-Security` header.
2. HTTPS Redirection Middleware (UseHttpsRedirection) redirects HTTP requests to HTTPS.
3. Static File Middleware (UseStaticFiles) returns static files and short-circuits further request processing.
4. Cookie Policy Middleware (UseCookiePolicy) conforms the app to the EU General Data Protection Regulation (GDPR) regulations.
5. Authentication Middleware (UseAuthentication) attempts to authenticate the user before they're allowed access to secure resources.

6. Session Middleware (UseSession) establishes and maintains session state. If the app uses session state, call Session Middleware after Cookie Policy Middleware and before MVC Middleware.
7. MVC (UseMvc) to add MVC to the request pipeline.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseAuthentication();
    app.UseSession();
    app.UseMvc();
}
```

In the preceding example code, each middleware extension method is exposed on IApplicationBuilder through the Microsoft.AspNetCore.Builder namespace.

UseExceptionHandler is the first middleware component added to the pipeline. Therefore, the Exception Handler Middleware catches any exceptions that occur in later calls.

Static File Middleware is called early in the pipeline so that it can handle requests and short-circuit without going through the remaining components. The Static File Middleware provides **no** authorization checks. Any files served by Static File Middleware, including those under *wwwroot*, are publicly available. For an approach to secure static files, see Static files in ASP.NET Core.

If the request isn't handled by the Static File Middleware, it's passed on to the Authentication Middleware (UseAuthentication), which performs authentication. Authentication doesn't short-circuit unauthenticated requests. Although Authentication Middleware authenticates requests, authorization (and rejection) occurs only after MVC selects a specific Razor Page or MVC controller and action.

The following example demonstrates a middleware order where requests for static files are handled by Static File Middleware before Response Compression Middleware. Static files aren't compressed with this middleware order. The MVC responses from UseMvcWithDefaultRoute can be compressed.

```
public void Configure(IApplicationBuilder app)
{
    // Static files aren't compressed by Static File Middleware.
    app.UseStaticFiles();

    app.UseResponseCompression();

    app.UseMvcWithDefaultRoute();
}
```

Use, Run, and Map

Configure the HTTP pipeline using Use, Run, and Map. The `Use` method can short-circuit the pipeline (that is, if it doesn't call a `next` request delegate). `Run` is a convention, and some middleware

components may expose `Run[Middleware]` methods that run at the end of the pipeline.

Map extensions are used as a convention for branching the pipeline. `Map` branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

```
public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code.

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

When `Map` is used, the matched path segments are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

MapWhen branches the request pipeline based on the result of the given predicate. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a predicate is used to detect the presence of a query string variable `branch`:

```
public class Startup
{
    private static void HandleBranch(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            var branchVer = context.Request.Query["branch"];
            await context.Response.WriteAsync($"Branch used = {branchVer}");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
                               HandleBranch);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code.

REQUEST	RESPONSE
localhost:1234	Hello from non-Map delegate.
localhost:1234/?branch=master	Branch used = master

`Map` supports nesting, for example:

```
app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a" processing
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b" processing
    });
});
```

`Map` can also match multiple segments at once:

```
public class Startup
{
    private static void HandleMultiSeg(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map multiple segments.");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1/seg1", HandleMultiSeg);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate.");
        });
    }
}
```

Built-in middleware

ASP.NET Core ships with the following middleware components. The *Order* column provides notes on middleware placement in the request processing pipeline and under what conditions the middleware may terminate request processing. When a middleware short-circuits the request processing pipeline and prevents further downstream middleware from processing a request, it's called a *terminal middleware*. For more information on short-circuiting, see the Create a middleware pipeline with IApplicationBuilder section.

MIDDLEWARE	DESCRIPTION	ORDER
Authentication	Provides authentication support.	Before `HttpContext.User` is needed. Terminal for OAuth callbacks.
Cookie Policy	Tracks consent from users for storing personal information and enforces minimum standards for cookie fields, such as `secure` and `SameSite`.	Before middleware that issues cookies. Examples: Authentication, Session, MVC (TempData).
CORS	Configures Cross-Origin Resource Sharing.	Before components that use CORS.
Diagnostics	Several separate middlewares that provide a developer exception page, exception handling, status code pages, and the default web page for new apps.	Before components that generate errors. Terminal for exceptions or serving the default web page for new apps.
Forwarded Headers	Forwards proxied headers onto the current request.	Before components that consume the updated fields. Examples: scheme, host, client IP, method.

MIDDLEWARE	DESCRIPTION	ORDER
Health Check	Checks the health of an ASP.NET Core app and its dependencies, such as checking database availability.	Terminal if a request matches a health check endpoint.
HTTP Method Override	Allows an incoming POST request to override the method.	Before components that consume the updated method.
HTTPS Redirection	Redirect all HTTP requests to HTTPS.	Before components that consume the URL.
HTTP Strict Transport Security (HSTS)	Security enhancement middleware that adds a special response header.	Before responses are sent and after components that modify requests. Examples: Forwarded Headers, URL Rewriting.
MVC	Processes requests with MVC/Razor Pages.	Terminal if a request matches a route.
OWIN	Interop with OWIN-based apps, servers, and middleware.	Terminal if the OWIN Middleware fully processes the request.
Response Caching	Provides support for caching responses.	Before components that require caching.
Response Compression	Provides support for compressing responses.	Before components that require compression.
Request Localization	Provides localization support.	Before localization sensitive components.
Endpoint Routing	Defines and constrains request routes.	Terminal for matching routes.
Session	Provides support for managing user sessions.	Before components that require Session.
Static Files	Provides support for serving static files and directory browsing.	Terminal if a request matches a file.
URL Rewrite	Provides support for rewriting URLs and redirecting requests.	Before components that consume the URL.
WebSockets	Enables the WebSockets protocol.	Before components that are required to accept WebSocket requests.

Additional resources

- Write custom ASP.NET Core middleware
- Test ASP.NET Core middleware
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core

- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

.NET Generic Host

9/22/2020 • 32 minutes to read • Edit Online

The ASP.NET Core templates create a .NET Core Generic Host, HostBuilder.

Host definition

A *host* is an object that encapsulates an app's resources, such as:

- Dependency injection (DI)
- Logging
- Configuration
- `IHostedService` implementations

When a host starts, it calls IHostedService.StartAsync on each implementation of IHostedService registered in the service container's collection of hosted services. In a web app, one of the `IHostedService` implementations is a web service that starts an HTTP server implementation.

The main reason for including all of the app's interdependent resources in one object is lifetime management: control over app startup and graceful shutdown.

Set up a host

The host is typically configured, built, and run by code in the `Program` class. The `Main` method:

- Calls a `CreateHostBuilder` method to create and configure a builder object.
- Calls `Build` and `Run` methods on the builder object.

The ASP.NET Core web templates generate the following code to create a Generic Host:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code creates a Generic Host using non-HTTP workload. The `IHostedService` implementation is added to the DI container:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
               services.AddHostedService<Worker>();
            });
}
```

For an HTTP workload, the `Main` method is the same but `CreateHostBuilder` calls `ConfigureWebHostDefaults`:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

The preceding code is generated by the ASP.NET Core templates.

If the app uses Entity Framework Core, don't change the name or signature of the `CreateHostBuilder` method. The Entity Framework Core tools expect to find a `CreateHostBuilder` method that configures the host without running the app. For more information, see Design-time DbContext Creation.

Default builder settings

The CreateDefaultBuilder method:

- Sets the content root to the path returned by GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with `DOTNET_`.
  - Command-line arguments.
- Loads app configuration from:
  - *appsettings.json*.
  - *appsettings.{Environment}.json*.
  - Secret Manager when the app runs in the `Development` environment.
  - Environment variables.
  - Command-line arguments.
- Adds the following logging providers:
  - Console
  - Debug
  - EventSource
  - EventLog (only when running on Windows)
- Enables scope validation and dependency validation when the environment is Development.

The `ConfigureWebHostDefaults` method:

- Loads host configuration from environment variables prefixed with `ASPNETCORE_`.
- Sets Kestrel server as the web server and configures it using the app's hosting configuration providers. For

the Kestrel server's default options, see Kestrel web server implementation in ASP.NET Core.

- Adds Host Filtering middleware.
- Adds Forwarded Headers middleware if `ASPNETCORE_FORWARDEDHEADERS_ENABLED` equals `true`.
- Enables IIS integration. For the IIS default options, see Host ASP.NET Core on Windows with IIS.

The Settings for all app types and Settings for web apps sections later in this article show how to override default builder settings.

Framework-provided services

The following services are registered automatically:

- IHostApplicationLifetime
- IHostLifetime
- IHostEnvironment / IWebHostEnvironment

For more information on framework-provided services, see Dependency injection in ASP.NET Core.

IHostApplicationLifetime

Inject the IHostApplicationLifetime (formerly `IApplicationLifetime`) service into any class to handle post-startup and graceful shutdown tasks. Three properties on the interface are cancellation tokens used to register app start and app stop event handler methods. The interface also includes a `StopApplication` method.

The following example is an `IHostedService` implementation that registers `IHostApplicationLifetime` events:

```
internal class LifetimeEventsHostedService : IHostedService
{
    private readonly ILogger _logger;
    private readonly IHostApplicationLifetime _appLifetime;

    public LifetimeEventsHostedService(
        ILogger<LifetimeEventsHostedService> logger,
        IHostApplicationLifetime appLifetime)
    {
        _logger = logger;
        _appLifetime = appLifetime;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _appLifetime.ApplicationStarted.Register(OnStarted);
        _appLifetime.ApplicationStopping.Register(OnStopping);
        _appLifetime.ApplicationStopped.Register(OnStopped);

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }

    private void OnStarted()
    {
        _logger.LogInformation("OnStarted has been called.");

        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        _logger.LogInformation("OnStopping has been called.");

        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        _logger.LogInformation("OnStopped has been called.");

        // Perform post-stopped activities here
    }
}
```

IHostLifetime

The IHostLifetime implementation controls when the host starts and when it stops. The last implementation registered is used.

`Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` is the default `IHostLifetime` implementation. `ConsoleLifetime`:

- Listens for `Ctrl+C`/SIGINT or SIGTERM and calls StopApplication to start the shutdown process.
- Unblocks extensions such as RunAsync and WaitForShutdownAsync.

IHostEnvironment

Inject the IHostEnvironment service into a class to get information about the following settings:

- ApplicationName
- EnvironmentName
- ContentRootPath

Web apps implement the `IWebHostEnvironment` interface, which inherits `IHostEnvironment` and adds the WebRootPath.

Host configuration

Host configuration is used for the properties of the IHostEnvironment implementation.

Host configuration is available from HostBuilderContext.Configuration inside ConfigureAppConfiguration. After `ConfigureAppConfiguration`, `HostBuilderContext.Configuration` is replaced with the app config.

To add host configuration, call ConfigureHostConfiguration on `IHostBuilder`. `ConfigureHostConfiguration` can be called multiple times with additive results. The host uses whichever option sets a value last on a given key.

The environment variable provider with prefix `DOTNET_` and command-line arguments are included by `CreateDefaultBuilder`. For web apps, the environment variable provider with prefix `ASPNETCORE_` is added. The prefix is removed when the environment variables are read. For example, the environment variable value for `ASPNETCORE_ENVIRONMENT` becomes the host configuration value for the `environment` key.

The following example creates host configuration:

```
// using Microsoft.Extensions.Configuration;

Host.CreateDefaultBuilder(args)
    .ConfigureHostConfiguration(configHost =>
    {
        configHost.SetBasePath(Directory.GetCurrentDirectory());
        configHost.AddJsonFile("hostsettings.json", optional: true);
        configHost.AddEnvironmentVariables(prefix: "PREFIX_");
        configHost.AddCommandLine(args);
    });
```

App configuration

App configuration is created by calling ConfigureAppConfiguration on `IHostBuilder`. `ConfigureAppConfiguration` can be called multiple times with additive results. The app uses whichever option sets a value last on a given key.

The configuration created by `ConfigureAppConfiguration` is available at HostBuilderContext.Configuration for subsequent operations and as a service from DI. The host configuration is also added to the app configuration.

For more information, see Configuration in ASP.NET Core.

Settings for all app types

This section lists host settings that apply to both HTTP and non-HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix.

ApplicationName

The IHostEnvironment.ApplicationName property is set from host configuration during host construction.

Key: `applicationName`
Type: `string`

Default: The name of the assembly that contains the app's entry point.
Environment variable: `<PREFIX_>APPLICATIONNAME`

To set this value, use the environment variable.

ContentRoot

The IHostEnvironment.ContentRootPath property determines where the host begins searching for content files. If the path doesn't exist, the host fails to start.

Key: `contentRoot`
Type: `string`
Default: The folder where the app assembly resides.
Environment variable: `<PREFIX_>CONTENTROOT`

To set this value, use the environment variable or call `UseContentRoot` on `IHostBuilder`:

```
Host.CreateDefaultBuilder(args)
    .UseContentRoot("c:\\content-root")
    //...
```

For more information, see:

- Fundamentals: Content root
- WebRoot

EnvironmentName

The IHostEnvironment.EnvironmentName property can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case-sensitive.

Key: `environment`
Type: `string`
Default: `Production`
Environment variable: `<PREFIX_>ENVIRONMENT`

To set this value, use the environment variable or call `UseEnvironment` on `IHostBuilder`:

```
Host.CreateDefaultBuilder(args)
    .UseEnvironment("Development")
    //...
```

ShutdownTimeout

HostOptions.ShutdownTimeout sets the timeout for StopAsync. The default value is five seconds. During the timeout period, the host:

- Triggers IHostApplicationLifetime.ApplicationStopping.
- Attempts to stop hosted services, logging errors for services that fail to stop.

If the timeout period expires before all of the hosted services stop, any remaining active services are stopped when the app shuts down. The services stop even if they haven't finished processing. If services require additional time to stop, increase the timeout.

Key: `shutdownTimeoutSeconds`
Type: `int`
Default: 5 seconds
Environment variable: `<PREFIX_>SHUTDOWNTIMEOUTSECONDS`

To set this value, use the environment variable or configure `HostOptions`. The following example sets the timeout to 20 seconds:

```
Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
    {
        services.Configure<HostOptions>(option =>
        {
            option.ShutdownTimeout = System.TimeSpan.FromSeconds(20);
        });
    });
```

Settings for web apps

Some host settings apply only to HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix.

Extension methods on `IWebHostBuilder` are available for these settings. Code samples that show how to call the extension methods assume `webBuilder` is an instance of `IWebHostBuilder`, as in the following example:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.CaptureStartupErrors(true);
            webBuilder.UseStartup<Startup>();
        });
```

CaptureStartupErrors

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

Key: `captureStartupErrors`
Type: `bool` (`true` or `1`)
Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.
Environment variable: `<PREFIX_>CAPTURESTARTUPERRORS`

To set this value, use configuration or call `CaptureStartupErrors`:

```
webBuilder.CaptureStartupErrors(true);
```

DetailedErrors

When enabled, or when the environment is `Development`, the app captures detailed errors.

Key: `detailedErrors`
Type: `bool` (`true` or `1`)
Default: `false`
Environment variable: `<PREFIX_>_DETAILEDERRORS`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting(WebHostDefaults.DetailedErrorsKey, "true");
```

HostingStartupAssemblies

A semicolon-delimited string of hosting startup assemblies to load on startup. Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

Key: `hostingStartupAssemblies`
Type: `string`
Default: Empty string
Environment variable: `<PREFIX_>_HOSTINGSTARTUPASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting(WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2");
```

HostingStartupExcludeAssemblies

A semicolon-delimited string of hosting startup assemblies to exclude on startup.

Key: `hostingStartupExcludeAssemblies`
Type: `string`
Default: Empty string
Environment variable: `<PREFIX_>_HOSTINGSTARTUPEXCLUDEASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey, "assembly1;assembly2");
```

HTTPS_Port

The HTTPS redirect port. Used in enforcing HTTPS.

Key: `https_port`
Type: `string`
Default: A default value isn't set.
Environment variable: `<PREFIX_>HTTPS_PORT`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting("https_port", "8080");
```

PreferHostingUrls

Indicates whether the host should listen on the URLs configured with the `IWebHostBuilder` instead of those URLs configured with the `IServer` implementation.

Key: `preferHostingUrls`
Type: `bool` (`true` or `1`)
Default: `true`
Environment variable: `<PREFIX_>_PREFERHOSTINGURLS`

To set this value, use the environment variable or call `PreferHostingUrls`:

```
webBuilder.PreferHostingUrls(false);
```

PreventHostingStartup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. For more information, see Use hosting startup assemblies in ASP.NET Core.

Key: `preventHostingStartup`
Type: `bool` (`true` or `1`)
Default: `false`
Environment variable: `<PREFIX_>_PREVENTHOSTINGSTARTUP`

To set this value, use the environment variable or call `UseSetting` :

```
webBuilder.UseSetting(WebHostDefaults.PreventHostingStartupKey, "true");
```

StartupAssembly

The assembly to search for the `Startup` class.

Key: `startupAssembly`
Type: `string`
Default: The app's assembly
Environment variable: `<PREFIX_>STARTUPASSEMBLY`

To set this value, use the environment variable or call `UseStartup` . `UseStartup` can take an assembly name (`string`) or a type (`TStartup`). If multiple `UseStartup` methods are called, the last one takes precedence.

```
webBuilder.UseStartup("StartupAssemblyName");
```

```
webBuilder.UseStartup<Startup>();
```

URLs

A semicolon-delimited list of IP addresses or host addresses with ports and protocols that the server should listen on for requests. For example, `http://localhost:123` . Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary among servers.

Key: `urls`
Type: `string`
Default: `http://localhost:5000` and `https://localhost:5001`
Environment variable: `<PREFIX_>URLS`

To set this value, use the environment variable or call `UseUrls` :

```
webBuilder.UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002");
```

Kestrel has its own endpoint configuration API. For more information, see Kestrel web server implementation in ASP.NET Core.

WebRoot

The IWebHostEnvironment.WebRootPath property determines the relative path to the app's static assets. If the path doesn't exist, a no-op file provider is used.

Key: `webroot`
Type: `string`

Default: The default is `wwwroot`. The path to *{content root}/wwwroot* must exist.
Environment variable: `<PREFIX_>WEBROOT`

To set this value, use the environment variable or call `UseWebRoot` on `IWebHostBuilder`:

```
webBuilder.UseWebRoot("public");
```

For more information, see:

- Fundamentals: Web root
- ContentRoot

Manage the host lifetime

Call methods on the built IHost implementation to start and stop the app. These methods affect all IHostedService implementations that are registered in the service container.

Run

Run runs the app and blocks the calling thread until the host is shut down.

RunAsync

RunAsync runs the app and returns a Task that completes when the cancellation token or shutdown is triggered.

RunConsoleAsync

RunConsoleAsync enables console support, builds and starts the host, and waits for `Ctrl+C`/SIGINT or SIGTERM to shut down.

Start

Start starts the host synchronously.

StartAsync

StartAsync starts the host and returns a Task that completes when the cancellation token or shutdown is triggered.

WaitForStartAsync is called at the start of `StartAsync`, which waits until it's complete before continuing. This can be used to delay startup until signaled by an external event.

StopAsync

StopAsync attempts to stop the host within the provided timeout.

WaitForShutdown

WaitForShutdown blocks the calling thread until shutdown is triggered by the IHostLifetime, such as via `Ctrl+C`/SIGINT or SIGTERM.

WaitForShutdownAsync

WaitForShutdownAsync returns a Task that completes when shutdown is triggered via the given token and calls StopAsync.

External control

Direct control of the host lifetime can be achieved using methods that can be called externally:

```
public class Program
{
    private IHost _host;

    public Program()
    {
        _host = new HostBuilder()
            .Build();
    }

    public async Task StartAsync()
    {
        _host.StartAsync();
    }

    public async Task StopAsync()
    {
        using (_host)
        {
            await _host.StopAsync(TimeSpan.FromSeconds(5));
        }
    }
}
```

ASP.NET Core apps configure and launch a host. The host is responsible for app startup and lifetime management.

This article covers the ASP.NET Core Generic Host (HostBuilder), which is used for apps that don't process HTTP requests.

The purpose of Generic Host is to decouple the HTTP pipeline from the Web Host API to enable a wider array of host scenarios. Messaging, background tasks, and other non-HTTP workloads based on Generic Host benefit from cross-cutting capabilities, such as configuration, dependency injection (DI), and logging.

Generic Host is new in ASP.NET Core 2.1 and isn't suitable for web hosting scenarios. For web hosting scenarios, use the Web Host. Generic Host will replace Web Host in a future release and act as the primary host API in both HTTP and non-HTTP scenarios.

View or download sample code (how to download)

When running the sample app in Visual Studio Code, use an *external or integrated terminal*. Don't run the sample in an `internalConsole`.

To set the console in Visual Studio Code:

1. Open the *.vscode/launch.json* file.
2. In the **.NET Core Launch (console)** configuration, locate the **console** entry. Set the value to either `externalTerminal` or `integratedTerminal`.

Introduction

The Generic Host library is available in the Microsoft.Extensions.Hosting namespace and provided by the Microsoft.Extensions.Hosting package. The `Microsoft.Extensions.Hosting` package is included in the Microsoft.AspNetCore.App metapackage (ASP.NET Core 2.1 or later).

IHostedService is the entry point to code execution. Each IHostedService implementation is executed in the order of service registration in ConfigureServices. StartAsync is called on each IHostedService when the host starts, and StopAsync is called in reverse registration order when the host shuts down gracefully.

Set up a host

IHostBuilder is the main component that libraries and apps use to initialize, build, and run the host:

```
public static async Task Main(string[] args)
{
    var host = new HostBuilder()
        .Build();

    await host.RunAsync();
}
```

Options

HostOptions configure options for the IHost.

Shutdown timeout

ShutdownTimeout sets the timeout for StopAsync. The default value is five seconds.

The following option configuration in `Program.Main` increases the default five-second shutdown timeout to 20 seconds:

```
var host = new HostBuilder()
    .ConfigureServices((hostContext, services) =>
    {
        services.Configure<HostOptions>(option =>
        {
            option.ShutdownTimeout = System.TimeSpan.FromSeconds(20);
        });
    })
    .Build();
```

Default services

The following services are registered during host initialization:

- Environment (IHostingEnvironment)
- HostBuilderContext
- Configuration (IConfiguration)
- IApplicationLifetime (`Microsoft.Extensions.Hosting.Internal.ApplicationLifetime`)
- IHostLifetime (`Microsoft.Extensions.Hosting.Internal.ConsoleLifetime`)
- IHost
- Options (AddOptions)
- Logging (AddLogging)

Host configuration

Host configuration is created by:

- Calling extension methods on IHostBuilder to set the content root and environment.
- Reading configuration from configuration providers in ConfigureHostConfiguration.

Extension methods

Application key (name)

The IHostingEnvironment.ApplicationName property is set from host configuration during host construction. To set the value explicitly, use the HostDefaults.ApplicationKey:

Key: `applicationName`
Type: `string`
Default: The name of the assembly containing the app's entry point.
Set using: `HostBuilderContext.HostingEnvironment.ApplicationName`
Environment variable: `<PREFIX_>APPLICATIONNAME` (`<PREFIX_>` is optional and user-defined)

Content root

This setting determines where the host begins searching for content files.

Key: `contentRoot`
Type: `string`
Default: Defaults to the folder where the app assembly resides.
Set using: `UseContentRoot`
Environment variable: `<PREFIX_>CONTENTROOT` (`<PREFIX_>` is optional and user-defined)

If the path doesn't exist, the host fails to start.

```
var host = new HostBuilder()
    .UseContentRoot("c:\\<content-root>")
```

For more information, see Fundamentals: Content root.

Environment

Sets the app's environment.

Key: `environment`
Type: `string`
Default: `Production`
Set using: `UseEnvironment`
Environment variable: `<PREFIX_>ENVIRONMENT` (`<PREFIX_>` is optional and user-defined)

The environment can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case-sensitive.

```
var host = new HostBuilder()
    .UseEnvironment(EnvironmentName.Development)
```

ConfigureHostConfiguration

ConfigureHostConfiguration uses an IConfigurationBuilder to create an IConfiguration for the host. The host configuration is used to initialize the IHostingEnvironment for use in the app's build process.

ConfigureHostConfiguration can be called multiple times with additive results. The host uses whichever option sets a value last on a given key.

No providers are included by default. You must explicitly specify whatever configuration providers the app requires in ConfigureHostConfiguration, including:

- File configuration (for example, from a *hostsettings.json* file).
- Environment variable configuration.
- Command-line argument configuration.
- Any other required configuration providers.

File configuration of the host is enabled by specifying the app's base path with `SetBasePath` followed by a call to one of the file configuration providers. The sample app uses a JSON file, *hostsettings.json*, and calls AddJsonFile to consume the file's host configuration settings.

To add environment variable configuration of the host, call AddEnvironmentVariables on the host builder. `AddEnvironmentVariables` accepts an optional user-defined prefix. The sample app uses a prefix of `PREFIX_`. The prefix is removed when the environment variables are read. When the sample app's host is configured, the environment variable value for `PREFIX_ENVIRONMENT` becomes the host configuration value for the `environment` key.

During development when using Visual Studio or running an app with `dotnet run`, environment variables may be set in the *Properties/launchSettings.json* file. In Visual Studio Code, environment variables may be set in the *.vscode/launch.json* file during development. For more information, see Use multiple environments in ASP.NET Core.

Command-line configuration is added by calling AddCommandLine. Command-line configuration is added last to permit command-line arguments to override configuration provided by the earlier configuration providers.

hostsettings.json:

```
{
  "environment": "Development"
}
```

Additional configuration can be provided with the applicationName and contentRoot keys.

Example `HostBuilder` configuration using ConfigureHostConfiguration:

```
var host = new HostBuilder()
    .ConfigureHostConfiguration(configHost =>
    {
        configHost.SetBasePath(Directory.GetCurrentDirectory());
        configHost.AddJsonFile("hostsettings.json", optional: true);
        configHost.AddEnvironmentVariables(prefix: "PREFIX_");
        configHost.AddCommandLine(args);
    })
```

ConfigureAppConfiguration

App configuration is created by calling ConfigureAppConfiguration on the IHostBuilder implementation. ConfigureAppConfiguration uses an IConfigurationBuilder to create an IConfiguration for the app. ConfigureAppConfiguration can be called multiple times with additive results. The app uses whichever option sets a value last on a given key. The configuration created by ConfigureAppConfiguration is available at HostBuilderContext.Configuration for subsequent operations and in Services.

App configuration automatically receives host configuration provided by ConfigureHostConfiguration.

Example app configuration using ConfigureAppConfiguration:

```
var host = new HostBuilder()
    .ConfigureAppConfiguration((hostContext, configApp) =>
    {
        configApp.SetBasePath(Directory.GetCurrentDirectory());
        configApp.AddJsonFile("appsettings.json", optional: true);
        configApp.AddJsonFile(
            $"appsettings.{hostContext.HostingEnvironment.EnvironmentName}.json",
            optional: true);
        configApp.AddEnvironmentVariables(prefix: "PREFIX_");
        configApp.AddCommandLine(args);
    })
```

appsettings.json:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

appsettings.Development.json:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

appsettings.Production.json:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Error",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

To move settings files to the output directory, specify the settings files as MSBuild project items in the project file. The sample app moves its JSON app settings files and *hostsettings.json* with the following `<Content>` item:

```
<ItemGroup>
  <Content Include="**\*.json" Exclude="bin\**\*;obj\**\*"
      CopyToOutputDirectory="PreserveNewest" />
</ItemGroup>
```

> **NOTE**
>
> Configuration extension methods, such as AddJsonFile and AddEnvironmentVariables require additional NuGet packages, such as Microsoft.Extensions.Configuration.Json and Microsoft.Extensions.Configuration.EnvironmentVariables. Unless the app uses the Microsoft.AspNetCore.App metapackage, these packages must be added to the project in addition to the core Microsoft.Extensions.Configuration package. For more information, see Configuration in ASP.NET Core.

ConfigureServices

ConfigureServices adds services to the app's dependency injection container. ConfigureServices can be called multiple times with additive results.

A hosted service is a class with background task logic that implements the IHostedService interface. For more information, see Background tasks with hosted services in ASP.NET Core.

The sample app uses the `AddHostedService` extension method to add a service for lifetime events, `LifetimeEventsHostedService`, and a timed background task, `TimedHostedService`, to the app:

```
var host = new HostBuilder()
    .ConfigureServices((hostContext, services) =>
    {
        if (hostContext.HostingEnvironment.IsDevelopment())
        {
            // Development service configuration
        }
        else
        {
            // Non-development service configuration
        }

        services.AddHostedService<LifetimeEventsHostedService>();
        services.AddHostedService<TimedHostedService>();
    })
```

ConfigureLogging

ConfigureLogging adds a delegate for configuring the provided ILoggingBuilder. ConfigureLogging may be called multiple times with additive results.

```
var host = new HostBuilder()
    .ConfigureLogging((hostContext, configLogging) =>
    {
        configLogging.AddConsole();
        configLogging.AddDebug();
    })
```

UseConsoleLifetime

UseConsoleLifetime listens for `Ctrl+C`/SIGINT or SIGTERM and calls StopApplication to start the shutdown process. UseConsoleLifetime unblocks extensions such as RunAsync and WaitForShutdownAsync. `Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` is pre-registered as the default lifetime implementation. The last lifetime registered is used.

```
var host = new HostBuilder()
    .UseConsoleLifetime()
```

Container configuration

To support plugging in other containers, the host can accept an IServiceProviderFactory<TContainerBuilder>. Providing a factory isn't part of the DI container registration but is instead a host intrinsic used to create the concrete DI container. UseServiceProviderFactory(IServiceProviderFactory<TContainerBuilder>) overrides the default factory used to create the app's service provider.

Custom container configuration is managed by the ConfigureContainer method. ConfigureContainer provides a strongly-typed experience for configuring the container on top of the underlying host API. ConfigureContainer can be called multiple times with additive results.

Create a service container for the app:

```
namespace GenericHostSample
{
    internal class ServiceContainer
    {
    }
}
```

Provide a service container factory:

```
using System;
using Microsoft.Extensions.DependencyInjection;

namespace GenericHostSample
{
    internal class ServiceContainerFactory :
        IServiceProviderFactory<ServiceContainer>
    {
        public ServiceContainer CreateBuilder(
            IServiceCollection services)
        {
            return new ServiceContainer();
        }

        public IServiceProvider CreateServiceProvider(
            ServiceContainer containerBuilder)
        {
            throw new NotImplementedException();
        }
    }
}
```

Use the factory and configure the custom service container for the app:

```
var host = new HostBuilder()
    .UseServiceProviderFactory<ServiceContainer>(new ServiceContainerFactory())
    .ConfigureContainer<ServiceContainer>((hostContext, container) =>
    {
    })
```

Extensibility

Host extensibility is performed with extension methods on IHostBuilder. The following example shows how an extension method extends an IHostBuilder implementation with the TimedHostedService example demonstrated in Background tasks with hosted services in ASP.NET Core.

```
var host = new HostBuilder()
    .UseHostedService<TimedHostedService>()
    .Build();

await host.StartAsync();
```

An app establishes the `UseHostedService` extension method to register the hosted service passed in `T`:

```
using System;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

public static class Extensions
{
    public static IHostBuilder UseHostedService<T>(this IHostBuilder hostBuilder)
        where T : class, IHostedService, IDisposable
    {
        return hostBuilder.ConfigureServices(services =>
            services.AddHostedService<T>());
    }
}
```

Manage the host

The IHost implementation is responsible for starting and stopping the IHostedService implementations that are registered in the service container.

Run

Run runs the app and blocks the calling thread until the host is shut down:

```
public class Program
{
    public void Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        host.Run();
    }
}
```

RunAsync

RunAsync runs the app and returns a Task that completes when the cancellation token or shutdown is triggered:

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        await host.RunAsync();
    }
}
```

RunConsoleAsync

RunConsoleAsync enables console support, builds and starts the host, and waits for `Ctrl+C`/SIGINT or SIGTERM to shut down.

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var hostBuilder = new HostBuilder();

        await hostBuilder.RunConsoleAsync();
    }
}
```

Start and StopAsync

Start starts the host synchronously.

StopAsync attempts to stop the host within the provided timeout.

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            host.Start();

            await host.StopAsync(TimeSpan.FromSeconds(5));
        }
    }
}
```

StartAsync and StopAsync

StartAsync starts the app.

StopAsync stops the app.

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            await host.StartAsync();

            await host.StopAsync();
        }
    }
}
```

WaitForShutdown

WaitForShutdown is triggered via the IHostLifetime, such as `Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` (listens for `Ctrl+C`/SIGINT or SIGTERM).

WaitForShutdown calls StopAsync.

```
public class Program
{
    public void Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            host.Start();

            host.WaitForShutdown();
        }
    }
}
```

WaitForShutdownAsync

WaitForShutdownAsync returns a Task that completes when shutdown is triggered via the given token and calls StopAsync.

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var host = new HostBuilder()
            .Build();

        using (host)
        {
            await host.StartAsync();

            await host.WaitForShutdownAsync();
        }

    }
}
```

External control

External control of the host can be achieved using methods that can be called externally:

```
public class Program
{
    private IHost _host;

    public Program()
    {
        _host = new HostBuilder()
            .Build();
    }

    public async Task StartAsync()
    {
        _host.StartAsync();
    }

    public async Task StopAsync()
    {
        using (_host)
        {
            await _host.StopAsync(TimeSpan.FromSeconds(5));
        }
    }
}
```

WaitForStartAsync is called at the start of StartAsync, which waits until it's complete before continuing. This can be used to delay startup until signaled by an external event.

IHostingEnvironment interface

IHostingEnvironment provides information about the app's hosting environment. Use constructor injection to obtain the IHostingEnvironment in order to use its properties and extension methods:

```
public class MyClass
{
    private readonly IHostingEnvironment _env;

    public MyClass(IHostingEnvironment env)
    {
        _env = env;
    }

    public void DoSomething()
    {
        var environmentName = _env.EnvironmentName;
    }
}
```

For more information, see Use multiple environments in ASP.NET Core.

IApplicationLifetime interface

IApplicationLifetime allows for post-startup and shutdown activities, including graceful shutdown requests. Three properties on the interface are cancellation tokens used to register Action methods that define startup and shutdown events.

CANCELLATION TOKEN	TRIGGERED WHEN...
ApplicationStarted	The host has fully started.

CANCELLATION TOKEN	TRIGGERED WHEN...
ApplicationStopped	The host is completing a graceful shutdown. All requests should be processed. Shutdown blocks until this event completes.
ApplicationStopping	The host is performing a graceful shutdown. Requests may still be processing. Shutdown blocks until this event completes.

Constructor-inject the IApplicationLifetime service into any class. The sample app uses constructor injection into a `LifetimeEventsHostedService` class (an IHostedService implementation) to register the events.

LifetimeEventsHostedService.cs:

```
internal class LifetimeEventsHostedService : IHostedService
{
    private readonly ILogger _logger;
    private readonly IApplicationLifetime _appLifetime;

    public LifetimeEventsHostedService(
        ILogger<LifetimeEventsHostedService> logger,
        IApplicationLifetime appLifetime)
    {
        _logger = logger;
        _appLifetime = appLifetime;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _appLifetime.ApplicationStarted.Register(OnStarted);
        _appLifetime.ApplicationStopping.Register(OnStopping);
        _appLifetime.ApplicationStopped.Register(OnStopped);

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }

    private void OnStarted()
    {
        _logger.LogInformation("OnStarted has been called.");

        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        _logger.LogInformation("OnStopping has been called.");

        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        _logger.LogInformation("OnStopped has been called.");

        // Perform post-stopped activities here
    }
}
```

StopApplication requests termination of the app. The following class uses StopApplication to gracefully shut down an app when the class's `Shutdown` method is called:

```
public class MyClass
{
    private readonly IApplicationLifetime _appLifetime;

    public MyClass(IApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

The ASP.NET Core templates create a .NET Core Generic Host (HostBuilder).

Host definition

A *host* is an object that encapsulates an app's resources, such as:

- Dependency injection (DI)
- Logging
- Configuration
- `IHostedService` implementations

When a host starts, it calls IHostedService.StartAsync on each implementation of IHostedService registered in the service container's collection of hosted services. In a web app, one of the `IHostedService` implementations is a web service that starts an HTTP server implementation.

The main reason for including all of the app's interdependent resources in one object is lifetime management: control over app startup and graceful shutdown.

Set up a host

The host is typically configured, built, and run by code in the `Program` class. The `Main` method:

- Calls a `CreateHostBuilder` method to create and configure a builder object.
- Calls `Build` and `Run` methods on the builder object.

The ASP.NET Core web templates generate the following code to create a host:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code creates a non-HTTP workload with a `IHostedService` implementation added to the DI container.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
               services.AddHostedService<Worker>();
            });
}
```

For an HTTP workload, the `Main` method is the same but `CreateHostBuilder` calls `ConfigureWebHostDefaults`:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

If the app uses Entity Framework Core, don't change the name or signature of the `CreateHostBuilder` method. The Entity Framework Core tools expect to find a `CreateHostBuilder` method that configures the host without running the app. For more information, see Design-time DbContext Creation.

Default builder settings

The CreateDefaultBuilder method:

- Sets the content root to the path returned by GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with `DOTNET_`.
  - Command-line arguments.
- Loads app configuration from:
  - *appsettings.json*.
  - *appsettings.{Environment}.json*.

    - Secret Manager when the app runs in the `Development` environment.
    - Environment variables.
    - Command-line arguments.
- Adds the following logging providers:
    - Console
    - Debug
    - EventSource
    - EventLog (only when running on Windows)
- Enables scope validation and dependency validation when the environment is Development.

The `ConfigureWebHostDefaults` method:

- Loads host configuration from environment variables prefixed with `ASPNETCORE_`.
- Sets Kestrel server as the web server and configures it using the app's hosting configuration providers. For the Kestrel server's default options, see Kestrel web server implementation in ASP.NET Core.
- Adds Host Filtering middleware.
- Adds Forwarded Headers middleware if `ASPNETCORE_FORWARDEDHEADERS_ENABLED` equals `true`.
- Enables IIS integration. For the IIS default options, see Host ASP.NET Core on Windows with IIS.

The Settings for all app types and Settings for web apps sections later in this article show how to override default builder settings.

Framework-provided services

The following services are registered automatically:

- IHostApplicationLifetime
- IHostLifetime
- IHostEnvironment / IWebHostEnvironment

For more information on framework-provided services, see Dependency injection in ASP.NET Core.

IHostApplicationLifetime

Inject the IHostApplicationLifetime (formerly `IApplicationLifetime`) service into any class to handle post-startup and graceful shutdown tasks. Three properties on the interface are cancellation tokens used to register app start and app stop event handler methods. The interface also includes a `StopApplication` method.

The following example is an `IHostedService` implementation that registers `IHostApplicationLifetime` events:

```
internal class LifetimeEventsHostedService : IHostedService
{
    private readonly ILogger _logger;
    private readonly IHostApplicationLifetime _appLifetime;

    public LifetimeEventsHostedService(
        ILogger<LifetimeEventsHostedService> logger,
        IHostApplicationLifetime appLifetime)
    {
        _logger = logger;
        _appLifetime = appLifetime;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _appLifetime.ApplicationStarted.Register(OnStarted);
        _appLifetime.ApplicationStopping.Register(OnStopping);
        _appLifetime.ApplicationStopped.Register(OnStopped);

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }

    private void OnStarted()
    {
        _logger.LogInformation("OnStarted has been called.");

        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        _logger.LogInformation("OnStopping has been called.");

        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        _logger.LogInformation("OnStopped has been called.");

        // Perform post-stopped activities here
    }
}
```

IHostLifetime

The IHostLifetime implementation controls when the host starts and when it stops. The last implementation registered is used.

`Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` is the default `IHostLifetime` implementation. `ConsoleLifetime`:

- Listens for `Ctrl+C`/SIGINT or SIGTERM and calls StopApplication to start the shutdown process.
- Unblocks extensions such as RunAsync and WaitForShutdownAsync.

IHostEnvironment

Inject the IHostEnvironment service into a class to get information about the following settings:

- ApplicationName
- EnvironmentName
- ContentRootPath

Web apps implement the `IWebHostEnvironment` interface, which inherits `IHostEnvironment` and adds the WebRootPath.

Host configuration

Host configuration is used for the properties of the IHostEnvironment implementation.

Host configuration is available from HostBuilderContext.Configuration inside ConfigureAppConfiguration. After `ConfigureAppConfiguration`, `HostBuilderContext.Configuration` is replaced with the app config.

To add host configuration, call ConfigureHostConfiguration on `IHostBuilder`. `ConfigureHostConfiguration` can be called multiple times with additive results. The host uses whichever option sets a value last on a given key.

The environment variable provider with prefix `DOTNET_` and command-line arguments are included by `CreateDefaultBuilder`. For web apps, the environment variable provider with prefix `ASPNETCORE_` is added. The prefix is removed when the environment variables are read. For example, the environment variable value for `ASPNETCORE_ENVIRONMENT` becomes the host configuration value for the `environment` key.

The following example creates host configuration:

```
// using Microsoft.Extensions.Configuration;

Host.CreateDefaultBuilder(args)
    .ConfigureHostConfiguration(configHost =>
    {
        configHost.SetBasePath(Directory.GetCurrentDirectory());
        configHost.AddJsonFile("hostsettings.json", optional: true);
        configHost.AddEnvironmentVariables(prefix: "PREFIX_");
        configHost.AddCommandLine(args);
    });
```

App configuration

App configuration is created by calling ConfigureAppConfiguration on `IHostBuilder`. `ConfigureAppConfiguration` can be called multiple times with additive results. The app uses whichever option sets a value last on a given key.

The configuration created by `ConfigureAppConfiguration` is available at HostBuilderContext.Configuration for subsequent operations and as a service from DI. The host configuration is also added to the app configuration.

For more information, see Configuration in ASP.NET Core.

Settings for all app types

This section lists host settings that apply to both HTTP and non-HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix.

ApplicationName

The IHostEnvironment.ApplicationName property is set from host configuration during host construction.

Key: `applicationName`
Type: `string`

Default: The name of the assembly that contains the app's entry point.
Environment variable: `<PREFIX_>APPLICATIONNAME`

To set this value, use the environment variable.

ContentRoot

The IHostEnvironment.ContentRootPath property determines where the host begins searching for content files. If the path doesn't exist, the host fails to start.

Key: `contentRoot`
Type: `string`
Default: The folder where the app assembly resides.
Environment variable: `<PREFIX_>CONTENTROOT`

To set this value, use the environment variable or call `UseContentRoot` on `IHostBuilder`:

```
Host.CreateDefaultBuilder(args)
    .UseContentRoot("c:\\content-root")
    //...
```

For more information, see:

- Fundamentals: Content root
- WebRoot

EnvironmentName

The IHostEnvironment.EnvironmentName property can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case-sensitive.

Key: `environment`
Type: `string`
Default: `Production`
Environment variable: `<PREFIX_>ENVIRONMENT`

To set this value, use the environment variable or call `UseEnvironment` on `IHostBuilder`:

```
Host.CreateDefaultBuilder(args)
    .UseEnvironment("Development")
    //...
```

ShutdownTimeout

HostOptions.ShutdownTimeout sets the timeout for StopAsync. The default value is five seconds. During the timeout period, the host:

- Triggers IHostApplicationLifetime.ApplicationStopping.
- Attempts to stop hosted services, logging errors for services that fail to stop.

If the timeout period expires before all of the hosted services stop, any remaining active services are stopped when the app shuts down. The services stop even if they haven't finished processing. If services require additional time to stop, increase the timeout.

Key: `shutdownTimeoutSeconds`
Type: `int`
Default: 5 seconds
Environment variable: `<PREFIX_>SHUTDOWNTIMEOUTSECONDS`

To set this value, use the environment variable or configure `HostOptions`. The following example sets the timeout to 20 seconds:

```
Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
    {
        services.Configure<HostOptions>(option =>
        {
            option.ShutdownTimeout = System.TimeSpan.FromSeconds(20);
        });
    });
```

Disable app configuration reload on change

By default, *appsettings.json* and *appsettings.{Environment}.json* are reloaded when the file changes. To disable this reload behavior in ASP.NET Core 5.0 Preview 3 or later, set the `hostBuilder:reloadConfigOnChange` key to `false`.

Key: `hostBuilder:reloadConfigOnChange`
Type: `bool` (`true` or `1`)
Default: `true`
Command-line argument: `hostBuilder:reloadConfigOnChange`
Environment variable: `<PREFIX_>hostBuilder:reloadConfigOnChange`

> **WARNING**
> The colon (`:`) separator doesn't work with environment variable hierarchical keys on all platforms. For more information, see Environment variables.

Settings for web apps

Some host settings apply only to HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix.

Extension methods on `IWebHostBuilder` are available for these settings. Code samples that show how to call the extension methods assume `webBuilder` is an instance of `IWebHostBuilder`, as in the following example:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.CaptureStartupErrors(true);
            webBuilder.UseStartup<Startup>();
        });
```

CaptureStartupErrors

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

Key: `captureStartupErrors`
Type: `bool` (`true` or `1`)
Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.
Environment variable: `<PREFIX_>CAPTURESTARTUPERRORS`

To set this value, use configuration or call `CaptureStartupErrors`:

```
webBuilder.CaptureStartupErrors(true);
```

DetailedErrors

When enabled, or when the environment is `Development`, the app captures detailed errors.

Key: `detailedErrors`
Type: `bool` (`true` or `1`)
Default: `false`
Environment variable: `<PREFIX_>_DETAILEDERRORS`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting(WebHostDefaults.DetailedErrorsKey, "true");
```

HostingStartupAssemblies

A semicolon-delimited string of hosting startup assemblies to load on startup. Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

Key: `hostingStartupAssemblies`
Type: `string`
Default: Empty string
Environment variable: `<PREFIX_>_HOSTINGSTARTUPASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting(WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2");
```

HostingStartupExcludeAssemblies

A semicolon-delimited string of hosting startup assemblies to exclude on startup.

Key: `hostingStartupExcludeAssemblies`
Type: `string`
Default: Empty string
Environment variable: `<PREFIX_>_HOSTINGSTARTUPEXCLUDEASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey, "assembly1;assembly2");
```

HTTPS_Port

The HTTPS redirect port. Used in enforcing HTTPS.

Key: `https_port`
Type: `string`
Default: A default value isn't set.
Environment variable: `<PREFIX_>HTTPS_PORT`

To set this value, use configuration or call `UseSetting`:

```
webBuilder.UseSetting("https_port", "8080");
```

PreferHostingUrls

Indicates whether the host should listen on the URLs configured with the `IWebHostBuilder` instead of those URLs configured with the `IServer` implementation.

Key: `preferHostingUrls`
Type: `bool` (`true` or `1`)
Default: `true`
Environment variable: `<PREFIX_>_PREFERHOSTINGURLS`

To set this value, use the environment variable or call `PreferHostingUrls` :

```
webBuilder.PreferHostingUrls(false);
```

PreventHostingStartup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. For more information, see Use hosting startup assemblies in ASP.NET Core.

Key: `preventHostingStartup`
Type: `bool` (`true` or `1`)
Default: `false`
Environment variable: `<PREFIX_>_PREVENTHOSTINGSTARTUP`

To set this value, use the environment variable or call `UseSetting` :

```
webBuilder.UseSetting(WebHostDefaults.PreventHostingStartupKey, "true");
```

StartupAssembly

The assembly to search for the `Startup` class.

Key: `startupAssembly`
Type: `string`
Default: The app's assembly
Environment variable: `<PREFIX_>STARTUPASSEMBLY`

To set this value, use the environment variable or call `UseStartup` . `UseStartup` can take an assembly name (`string`) or a type (`TStartup`). If multiple `UseStartup` methods are called, the last one takes precedence.

```
webBuilder.UseStartup("StartupAssemblyName");
```

```
webBuilder.UseStartup<Startup>();
```

URLs

A semicolon-delimited list of IP addresses or host addresses with ports and protocols that the server should listen on for requests. For example, `http://localhost:123` . Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary among servers.

Key: `urls`
Type: `string`
Default: `http://localhost:5000` and `https://localhost:5001`
Environment variable: `<PREFIX_>URLS`

To set this value, use the environment variable or call `UseUrls`:

```
webBuilder.UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002");
```

Kestrel has its own endpoint configuration API. For more information, see Kestrel web server implementation in ASP.NET Core.

WebRoot

The IWebHostEnvironment.WebRootPath property determines the relative path to the app's static assets. If the path doesn't exist, a no-op file provider is used.

Key: `webroot`
Type: `string`
Default: The default is `wwwroot`. The path to *{content root}/wwwroot* must exist.
Environment variable: `<PREFIX_>WEBROOT`

To set this value, use the environment variable or call `UseWebRoot` on `IWebHostBuilder`:

```
webBuilder.UseWebRoot("public");
```

For more information, see:

- Fundamentals: Web root
- ContentRoot

Manage the host lifetime

Call methods on the built IHost implementation to start and stop the app. These methods affect all IHostedService implementations that are registered in the service container.

Run

Run runs the app and blocks the calling thread until the host is shut down.

RunAsync

RunAsync runs the app and returns a Task that completes when the cancellation token or shutdown is triggered.

RunConsoleAsync

RunConsoleAsync enables console support, builds and starts the host, and waits for `Ctrl+C`/SIGINT or SIGTERM to shut down.

Start

Start starts the host synchronously.

StartAsync

StartAsync starts the host and returns a Task that completes when the cancellation token or shutdown is triggered.

WaitForStartAsync is called at the start of `StartAsync`, which waits until it's complete before continuing. This

can be used to delay startup until signaled by an external event.

StopAsync

StopAsync attempts to stop the host within the provided timeout.

WaitForShutdown

WaitForShutdown blocks the calling thread until shutdown is triggered by the IHostLifetime, such as via `Ctrl+C`/SIGINT or SIGTERM.

WaitForShutdownAsync

WaitForShutdownAsync returns a Task that completes when shutdown is triggered via the given token and calls StopAsync.

External control

Direct control of the host lifetime can be achieved using methods that can be called externally:

```
public class Program
{
    private IHost _host;

    public Program()
    {
        _host = new HostBuilder()
            .Build();
    }

    public async Task StartAsync()
    {
        _host.StartAsync();
    }

    public async Task StopAsync()
    {
        using (_host)
        {
            await _host.StopAsync(TimeSpan.FromSeconds(5));
        }
    }
}
```

Additional resources

- Background tasks with hosted services in ASP.NET Core

ASP.NET Core Web Host

9/22/2020 • 17 minutes to read • Edit Online

ASP.NET Core apps configure and launch a *host*. The host is responsible for app startup and lifetime management. At a minimum, the host configures a server and a request processing pipeline. The host can also set up logging, dependency injection, and configuration.

This article covers the Web Host, which remains available only for backward compatibility. The Generic Host is recommended for all app types.

This article covers the Web Host, which is for hosting web apps. For other kinds of apps, use the Generic Host.

Set up a host

Create a host using an instance of IWebHostBuilder. This is typically performed in the app's entry point, the `Main` method.

In the project templates, `Main` is located in *Program.cs*. A typical app calls CreateDefaultBuilder to start setting up a host:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

The code that calls `CreateDefaultBuilder` is in a method named `CreateWebHostBuilder`, which separates it from the code in `Main` that calls `Run` on the builder object. This separation is required if you use Entity Framework Core tools. The tools expect to find a `CreateWebHostBuilder` method that they can call at design time to configure the host without running the app. An alternative is to implement `IDesignTimeDbContextFactory`. For more information, see Design-time DbContext Creation.

`CreateDefaultBuilder` performs the following tasks:

- Configures Kestrel server as the web server using the app's hosting configuration providers. For the Kestrel server's default options, see Kestrel web server implementation in ASP.NET Core.
- Sets the content root to the path returned by Directory.GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with `ASPNETCORE_` (for example, `ASPNETCORE_ENVIRONMENT`).
  - Command-line arguments.
- Loads app configuration in the following order from:
  - *appsettings.json*.
  - *appsettings.{Environment}.json*.
  - Secret Manager when the app runs in the `Development` environment using the entry assembly.
  - Environment variables.

    - Command-line arguments.
- Configures logging for console and debug output. Logging includes log filtering rules specified in a Logging configuration section of an *appsettings.json* or *appsettings.{Environment}.json* file.
- When running behind IIS with the ASP.NET Core Module, `CreateDefaultBuilder` enables IIS Integration, which configures the app's base address and port. IIS Integration also configures the app to capture startup errors. For the IIS default options, see Host ASP.NET Core on Windows with IIS.
- Sets ServiceProviderOptions.ValidateScopes to `true` if the app's environment is Development. For more information, see Scope validation.

The configuration defined by `CreateDefaultBuilder` can be overridden and augmented by ConfigureAppConfiguration, ConfigureLogging, and other methods and extension methods of IWebHostBuilder. A few examples follow:

- ConfigureAppConfiguration is used to specify additional `IConfiguration` for the app. The following `ConfigureAppConfiguration` call adds a delegate to include app configuration in the *appsettings.xml* file. `ConfigureAppConfiguration` may be called multiple times. Note that this configuration doesn't apply to the host (for example, server URLs or environment). See the Host configuration values section.

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureAppConfiguration((hostingContext, config) =>
    {
        config.AddXmlFile("appsettings.xml", optional: true, reloadOnChange: true);
    })
    ...
```

- The following `ConfigureLogging` call adds a delegate to configure the minimum logging level (SetMinimumLevel) to LogLevel.Warning. This setting overrides the settings in *appsettings.Development.json* (`LogLevel.Debug`) and *appsettings.Production.json* (`LogLevel.Error`) configured by `CreateDefaultBuilder` . `ConfigureLogging` may be called multiple times.

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureLogging(logging =>
    {
        logging.SetMinimumLevel(LogLevel.Warning);
    })
    ...
```

- The following call to `ConfigureKestrel` overrides the default Limits.MaxRequestBodySize of 30,000,000 bytes established when Kestrel was configured by `CreateDefaultBuilder` :

```
WebHost.CreateDefaultBuilder(args)
    .ConfigureKestrel((context, options) =>
    {
        options.Limits.MaxRequestBodySize = 20000000;
    });
```

- The following call to UseKestrel overrides the default Limits.MaxRequestBodySize of 30,000,000 bytes established when Kestrel was configured by `CreateDefaultBuilder` :

```
WebHost.CreateDefaultBuilder(args)
    .UseKestrel(options =>
    {
        options.Limits.MaxRequestBodySize = 20000000;
    });
```

The content root determines where the host searches for content files, such as MVC view files. When the app is started from the project's root folder, the project's root folder is used as the content root. This is the default used in Visual Studio and the dotnet new templates.

For more information on app configuration, see Configuration in ASP.NET Core.

> **NOTE**
> As an alternative to using the static `CreateDefaultBuilder` method, creating a host from WebHostBuilder is a supported approach with ASP.NET Core 2.x.

When setting up a host, Configure and ConfigureServices methods can be provided. If a `Startup` class is specified, it must define a `Configure` method. For more information, see App startup in ASP.NET Core. Multiple calls to `ConfigureServices` append to one another. Multiple calls to `Configure` or `UseStartup` on the `WebHostBuilder` replace previous settings.

Host configuration values

WebHostBuilder relies on the following approaches to set the host configuration values:

- Host builder configuration, which includes environment variables with the format `ASPNETCORE_{configurationKey}`. For example, `ASPNETCORE_ENVIRONMENT`.
- Extensions such as UseContentRoot and UseConfiguration (see the Override configuration section).
- UseSetting and the associated key. When setting a value with `UseSetting`, the value is set as a string regardless of the type.

The host uses whichever option sets a value last. For more information, see Override configuration in the next section.

Application Key (Name)

The `IWebHostEnvironment.ApplicationName` property is automatically set when UseStartup or Configure is called during host construction. The value is set to the name of the assembly containing the app's entry point. To set the value explicitly, use the WebHostDefaults.ApplicationKey:

The IHostingEnvironment.ApplicationName property is automatically set when UseStartup or Configure is called during host construction. The value is set to the name of the assembly containing the app's entry point. To set the value explicitly, use the WebHostDefaults.ApplicationKey:

Key: applicationName
Type: *string*
Default: The name of the assembly containing the app's entry point.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_APPLICATIONNAME`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.ApplicationKey, "CustomApplicationName")
```

Capture Startup Errors

This setting controls the capture of startup errors.

Key: captureStartupErrors
Type: *bool* (`true` or `1`)
Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.
Set using: `CaptureStartupErrors`

Environment variable: `ASPNETCORE_CAPTURESTARTUPERRORS`

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

```
WebHost.CreateDefaultBuilder(args)
    .CaptureStartupErrors(true)
```

Content root

This setting determines where ASP.NET Core begins searching for content files.

Key: contentRoot
Type: *string*
Default: Defaults to the folder where the app assembly resides.
Set using: `UseContentRoot`
Environment variable: `ASPNETCORE_CONTENTROOT`

The content root is also used as the base path for the web root. If the content root path doesn't exist, the host fails to start.

```
WebHost.CreateDefaultBuilder(args)
    .UseContentRoot("c:\\<content-root>")
```

For more information, see:

- Fundamentals: Content root
- Web root

Detailed Errors

Determines if detailed errors should be captured.

Key: detailedErrors
Type: *bool* (`true` or `1`)
Default: false
Set using: `UseSetting`
Environment variable: `ASPNETCORE_DETAILEDERRORS`

When enabled (or when the Environment is set to `Development`), the app captures detailed exceptions.

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.DetailedErrorsKey, "true")
```

Environment

Sets the app's environment.

Key: environment
Type: *string*
Default: Production
Set using: `UseEnvironment`
Environment variable: `ASPNETCORE_ENVIRONMENT`

The environment can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case sensitive. By default, the *Environment* is read from the `ASPNETCORE_ENVIRONMENT` environment variable. When using Visual Studio, environment variables may be set in the *launchSettings.json* file.

For more information, see Use multiple environments in ASP.NET Core.

```
WebHost.CreateDefaultBuilder(args)
    .UseEnvironment(EnvironmentName.Development)
```

Hosting Startup Assemblies

Sets the app's hosting startup assemblies.

Key: hostingStartupAssemblies
Type: *string*
Default: Empty string
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`

A semicolon-delimited string of hosting startup assemblies to load on startup.

Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2")
```

HTTPS Port

Set the HTTPS redirect port. Used in enforcing HTTPS.

Key: https_port **Type**: *string* **Default**: A default value isn't set. **Set using**: `UseSetting` **Environment variable**: `ASPNETCORE_HTTPS_PORT`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting("https_port", "8080")
```

Hosting Startup Exclude Assemblies

A semicolon-delimited string of hosting startup assemblies to exclude on startup.

Key: hostingStartupExcludeAssemblies
Type: *string*
Default: Empty string
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey, "assembly1;assembly2")
```

Prefer Hosting URLs

Indicates whether the host should listen on the URLs configured with the `WebHostBuilder` instead of those configured with the `IServer` implementation.

Key: preferHostingUrls
Type: *bool* (`true` or `1`)
Default: true
Set using: `PreferHostingUrls`

Environment variable: `ASPNETCORE_PREFERHOSTINGURLS`

```
WebHost.CreateDefaultBuilder(args)
    .PreferHostingUrls(false)
```

Prevent Hosting Startup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. For more information, see Use hosting startup assemblies in ASP.NET Core.

Key: preventHostingStartup
Type: *bool* (`true` or `1`)
Default: false
Set using: `UseSetting`
Environment variable: `ASPNETCORE_PREVENTHOSTINGSTARTUP`

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.PreventHostingStartupKey, "true")
```

Server URLs

Indicates the IP addresses or host addresses with ports and protocols that the server should listen on for requests.

Key: urls
Type: *string*
Default: http://localhost:5000
Set using: `UseUrls`
Environment variable: `ASPNETCORE_URLS`

Set to a semicolon-separated (;) list of URL prefixes to which the server should respond. For example, `http://localhost:123`. Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary among servers.

```
WebHost.CreateDefaultBuilder(args)
    .UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")
```

Kestrel has its own endpoint configuration API. For more information, see Kestrel web server implementation in ASP.NET Core.

Shutdown Timeout

Specifies the amount of time to wait for Web Host to shut down.

Key: shutdownTimeoutSeconds
Type: *int*
Default: 5
Set using: `UseShutdownTimeout`
Environment variable: `ASPNETCORE_SHUTDOWNTIMEOUTSECONDS`

Although the key accepts an *int* with `UseSetting` (for example, `.UseSetting(WebHostDefaults.ShutdownTimeoutKey, "10")`), the UseShutdownTimeout extension method takes a TimeSpan.

During the timeout period, hosting:

- Triggers IApplicationLifetime.ApplicationStopping.
- Attempts to stop hosted services, logging any errors for services that fail to stop.

If the timeout period expires before all of the hosted services stop, any remaining active services are stopped when the app shuts down. The services stop even if they haven't finished processing. If services require additional time to stop, increase the timeout.

```
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
```

Startup Assembly

Determines the assembly to search for the `Startup` class.

Key: startupAssembly
Type: *string*
Default: The app's assembly
Set using: `UseStartup`
Environment variable: `ASPNETCORE_STARTUPASSEMBLY`

The assembly by name (`string`) or type (`TStartup`) can be referenced. If multiple `UseStartup` methods are called, the last one takes precedence.

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup("StartupAssemblyName")
```

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<TStartup>()
```

Web root

Sets the relative path to the app's static assets.

Key: webroot
Type: *string*
Default: The default is `wwwroot`. The path to *{content root}/wwwroot* must exist. If the path doesn't exist, a no-op file provider is used.
Set using: `UseWebRoot`
Environment variable: `ASPNETCORE_WEBROOT`

```
WebHost.CreateDefaultBuilder(args)
    .UseWebRoot("public")
```

For more information, see:

- Fundamentals: Web root
- Content root

Override configuration

Use Configuration to configure Web Host. In the following example, host configuration is optionally specified in a *hostsettings.json* file. Any configuration loaded from the *hostsettings.json* file may be overridden by command-line arguments. The built configuration (in `config`) is used to configure the host with UseConfiguration. `IWebHostBuilder` configuration is added to the app's configuration, but the converse isn't true—

`ConfigureAppConfiguration` doesn't affect the `IWebHostBuilder` configuration.

Overriding the configuration provided by `UseUrls` with *hostsettings.json* config first, command-line argument config second:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var config = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("hostsettings.json", optional: true)
            .AddCommandLine(args)
            .Build();

        return WebHost.CreateDefaultBuilder(args)
            .UseUrls("http://*:5000")
            .UseConfiguration(config)
            .Configure(app =>
            {
                app.Run(context =>
                    context.Response.WriteAsync("Hello, World!"));
            });
    }
}
```

hostsettings.json:

```
{
    urls: "http://*:5005"
}
```

> **NOTE**
>
> UseConfiguration only copies keys from the provided `IConfiguration` to the host builder configuration. Therefore, setting `reloadOnChange: true` for JSON, INI, and XML settings files has no effect.

To specify the host run on a particular URL, the desired value can be passed in from a command prompt when executing dotnet run. The command-line argument overrides the `urls` value from the *hostsettings.json* file, and the server listens on port 8080:

```
dotnet run --urls "http://*:8080"
```

Manage the host

Run

The `Run` method starts the web app and blocks the calling thread until the host is shut down:

```
host.Run();
```

Start

Run the host in a non-blocking manner by calling its `Start` method:

```
using (host)
{
    host.Start();
    Console.ReadLine();
}
```

If a list of URLs is passed to the `Start` method, it listens on the URLs specified:

```
var urls = new List<string>()
{
    "http://*:5000",
    "http://localhost:5001"
};

var host = new WebHostBuilder()
    .UseKestrel()
    .UseStartup<Startup>()
    .Start(urls.ToArray());

using (host)
{
    Console.ReadLine();
}
```

The app can initialize and start a new host using the pre-configured defaults of `CreateDefaultBuilder` using a static convenience method. These methods start the server without console output and with WaitForShutdown wait for a break (Ctrl-C/SIGINT or SIGTERM):

Start(RequestDelegate app)

Start with a `RequestDelegate`:

```
using (var host = WebHost.Start(app => app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!" `WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, RequestDelegate app)

Start with a URL and `RequestDelegate`:

```
using (var host = WebHost.Start("http://localhost:8080", app => app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **Start(RequestDelegate app)**, except the app responds on `http://localhost:8080`.

Start(Action<IRouteBuilder> routeBuilder)

Use an instance of `IRouteBuilder` (Microsoft.AspNetCore.Routing) to use routing middleware:

```
using (var host = WebHost.Start(router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]}, {data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Use the following browser requests with the example:

REQUEST	RESPONSE
`http://localhost:5000/hello/Martin`	Hello, Martin!
`http://localhost:5000/buenosdias/Catrina`	Buenos dias, Catrina!
`http://localhost:5000/throw/ooops!`	Throws an exception with string "ooops!"
`http://localhost:5000/throw`	Throws an exception with string "Uh oh!"
`http://localhost:5000/Sante/Kevin`	Sante, Kevin!
`http://localhost:5000`	Hello World!

`WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, Action<IRouteBuilder> routeBuilder)

Use a URL and an instance of `IRouteBuilder`:

```
using (var host = WebHost.Start("http://localhost:8080", router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]}, {data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **Start(Action<IRouteBuilder> routeBuilder)**, except the app responds at `http://localhost:8080`.

StartWith(Action<IApplicationBuilder> app)

Provide a delegate to configure an `IApplicationBuilder`:

```
using (var host = WebHost.StartWith(app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        };
    })))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!" `WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

StartWith(string url, Action<IApplicationBuilder> app)

Provide a URL and a delegate to configure an `IApplicationBuilder`:

```
using (var host = WebHost.StartWith("http://localhost:8080", app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        };
    })))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **StartWith(Action<IApplicationBuilder> app)**, except the app responds on `http://localhost:8080`.

IWebHostEnvironment interface

The `IWebHostEnvironment` interface provides information about the app's web hosting environment. Use constructor injection to obtain the `IWebHostEnvironment` in order to use its properties and extension methods:

```
public class CustomFileReader
{
    private readonly IWebHostEnvironment _env;

    public CustomFileReader(IWebHostEnvironment env)
    {
        _env = env;
    }

    public string ReadFile(string filePath)
    {
        var fileProvider = _env.WebRootFileProvider;
        // Process the file here
    }
}
```

A convention-based approach can be used to configure the app at startup based on the environment. Alternatively, inject the `IWebHostEnvironment` into the `Startup` constructor for use in `ConfigureServices`:

```
public class Startup
{
    public Startup(IWebHostEnvironment env)
    {
        HostingEnvironment = env;
    }

    public IWebHostEnvironment HostingEnvironment { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }

        var contentRootPath = HostingEnvironment.ContentRootPath;
    }
}
```

NOTE

In addition to the `IsDevelopment` extension method, `IWebHostEnvironment` offers `IsStaging`, `IsProduction`, and `IsEnvironment(string environmentName)` methods. For more information, see Use multiple environments in ASP.NET Core.

The `IWebHostEnvironment` service can also be injected directly into the `Configure` method for setting up the processing pipeline:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        // In Development, use the Developer Exception Page
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // In Staging/Production, route exceptions to /error
        app.UseExceptionHandler("/error");
    }

    var contentRootPath = env.ContentRootPath;
}
```

`IWebHostEnvironment` can be injected into the `Invoke` method when creating custom middleware:

```
public async Task Invoke(HttpContext context, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        // Configure middleware for Development
    }
    else
    {
        // Configure middleware for Staging/Production
    }

    var contentRootPath = env.ContentRootPath;
}
```

IHostingEnvironment interface

The IHostingEnvironment interface provides information about the app's web hosting environment. Use constructor injection to obtain the `IHostingEnvironment` in order to use its properties and extension methods:

```
public class CustomFileReader
{
    private readonly IHostingEnvironment _env;

    public CustomFileReader(IHostingEnvironment env)
    {
        _env = env;
    }

    public string ReadFile(string filePath)
    {
        var fileProvider = _env.WebRootFileProvider;
        // Process the file here
    }
}
```

A convention-based approach can be used to configure the app at startup based on the environment. Alternatively, inject the `IHostingEnvironment` into the `Startup` constructor for use in `ConfigureServices`:

```
public class Startup
{
    public Startup(IHostingEnvironment env)
    {
        HostingEnvironment = env;
    }

    public IHostingEnvironment HostingEnvironment { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }

        var contentRootPath = HostingEnvironment.ContentRootPath;
    }
}
```

NOTE

In addition to the `IsDevelopment` extension method, `IHostingEnvironment` offers `IsStaging`, `IsProduction`, and `IsEnvironment(string environmentName)` methods. For more information, see Use multiple environments in ASP.NET Core.

The `IHostingEnvironment` service can also be injected directly into the `Configure` method for setting up the processing pipeline:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        // In Development, use the Developer Exception Page
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // In Staging/Production, route exceptions to /error
        app.UseExceptionHandler("/error");
    }

    var contentRootPath = env.ContentRootPath;
}
```

`IHostingEnvironment` can be injected into the `Invoke` method when creating custom middleware:

```
public async Task Invoke(HttpContext context, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        // Configure middleware for Development
    }
    else
    {
        // Configure middleware for Staging/Production
    }

    var contentRootPath = env.ContentRootPath;
}
```

IHostApplicationLifetime interface

`IHostApplicationLifetime` allows for post-startup and shutdown activities. Three properties on the interface are cancellation tokens used to register `Action` methods that define startup and shutdown events.

CANCELLATION TOKEN	TRIGGERED WHEN...
`ApplicationStarted`	The host has fully started.
`ApplicationStopped`	The host is completing a graceful shutdown. All requests should be processed. Shutdown blocks until this event completes.
`ApplicationStopping`	The host is performing a graceful shutdown. Requests may still be processing. Shutdown blocks until this event completes.

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IHostApplicationLifetime appLifetime)
    {
        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);

        Console.CancelKeyPress += (sender, eventArgs) =>
        {
            appLifetime.StopApplication();
            // Don't terminate the process immediately, wait for the Main thread to exit gracefully.
            eventArgs.Cancel = true;
        };
    }

    private void OnStarted()
    {
        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        // Perform post-stopped activities here
    }
}
```

`StopApplication` requests termination of the app. The following class uses `StopApplication` to gracefully shut down an app when the class's `Shutdown` method is called:

```
public class MyClass
{
    private readonly IHostApplicationLifetime _appLifetime;

    public MyClass(IHostApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

IApplicationLifetime interface

IApplicationLifetime allows for post-startup and shutdown activities. Three properties on the interface are cancellation tokens used to register `Action` methods that define startup and shutdown events.

CANCELLATION TOKEN	TRIGGERED WHEN...
ApplicationStarted	The host has fully started.

CANCELLATION TOKEN	TRIGGERED WHEN...
ApplicationStopped	The host is completing a graceful shutdown. All requests should be processed. Shutdown blocks until this event completes.
ApplicationStopping	The host is performing a graceful shutdown. Requests may still be processing. Shutdown blocks until this event completes.

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IApplicationLifetime appLifetime)
    {
        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);

        Console.CancelKeyPress += (sender, eventArgs) =>
        {
            appLifetime.StopApplication();
            // Don't terminate the process immediately, wait for the Main thread to exit gracefully.
            eventArgs.Cancel = true;
        };
    }

    private void OnStarted()
    {
        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        // Perform on-stopping activities here
    }

    private void OnStopped()
    {
        // Perform post-stopped activities here
    }
}
```

StopApplication requests termination of the app. The following class uses `StopApplication` to gracefully shut down an app when the class's `Shutdown` method is called:

```
public class MyClass
{
    private readonly IApplicationLifetime _appLifetime;

    public MyClass(IApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

Scope validation

CreateDefaultBuilder sets ServiceProviderOptions.ValidateScopes to `true` if the app's environment is Development.

When `ValidateScopes` is set to `true`, the default service provider performs checks to verify that:

- Scoped services aren't directly or indirectly resolved from the root service provider.
- Scoped services aren't directly or indirectly injected into singletons.

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app/server's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton because it's only disposed by the root container when app/server is shut down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

To always validate scopes, including in the Production environment, configure the ServiceProviderOptions with UseDefaultServiceProvider on the host builder:

```
WebHost.CreateDefaultBuilder(args)
    .UseDefaultServiceProvider((context, options) => {
        options.ValidateScopes = true;
    })
```

Additional resources

- Host ASP.NET Core on Windows with IIS
- Host ASP.NET Core on Linux with Nginx
- Host ASP.NET Core on Linux with Apache
- Host ASP.NET Core in a Windows Service

Web server implementations in ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

By Tom Dykstra, Steve Smith, Stephen Halter, and Chris Ross

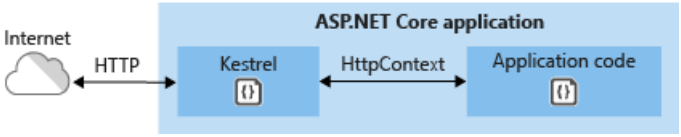
An ASP.NET Core app runs with an in-process HTTP server implementation. The server implementation listens for HTTP requests and surfaces them to the app as a set of request features composed into an HttpContext.

Kestrel

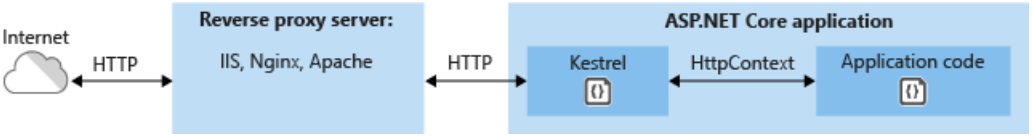
Kestrel is the default web server specified by the ASP.NET Core project templates.

Use Kestrel:

- By itself as an edge server processing requests directly from a network, including the Internet.



- With a *reverse proxy server*, such as Internet Information Services (IIS), Nginx, or Apache. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.



Either hosting configuration—with or without a reverse proxy server—is supported.

For Kestrel configuration guidance and information on when to use Kestrel in a reverse proxy configuration, see Kestrel web server implementation in ASP.NET Core.

- Windows
- macOS
- Linux

ASP.NET Core ships with the following:

- Kestrel server is the default, cross-platform HTTP server implementation.
- IIS HTTP Server is an in-process server for IIS.
- HTTP.sys server is a Windows-only HTTP server based on the HTTP.sys kernel driver and HTTP Server API.

When using IIS or IIS Express, the app either runs:

- In the same process as the IIS worker process (the in-process hosting model) with the IIS HTTP Server. *In-process* is the recommended configuration.
- In a process separate from the IIS worker process (the out-of-process hosting model) with the Kestrel server.

The ASP.NET Core Module is a native IIS module that handles native IIS requests between IIS and the in-process IIS HTTP Server or Kestrel. For more information, see ASP.NET Core Module.

Hosting models

Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the Windows Process Activation Service (WAS).

Using out-of-process hosting, ASP.NET Core apps run in a process separate from the IIS worker process, and the module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS).

For more information and configuration guidance, see the following topics:

- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module

- Windows
- macOS
- Linux

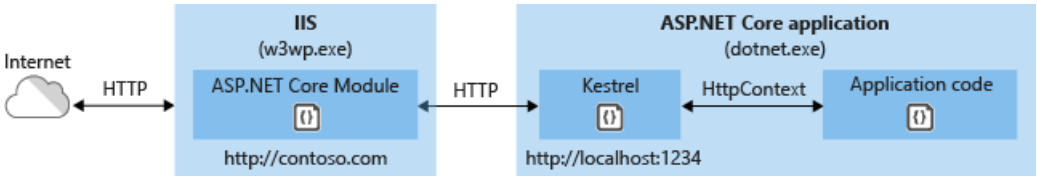
ASP.NET Core ships with the following:

- Kestrel server is the default, cross-platform HTTP server.
- HTTP.sys server is a Windows-only HTTP server based on the HTTP.sys kernel driver and HTTP Server API.

When using IIS or IIS Express, the app runs in a process separate from the IIS worker process (*out-of-process*) with the Kestrel server.

Because ASP.NET Core apps run in a process separate from the IIS worker process, the module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the IIS Integration Middleware configures the server to listen on `http://localhost:{port}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.

For IIS and ASP.NET Core Module configuration guidance, see the following topics:

- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module

Nginx with Kestrel

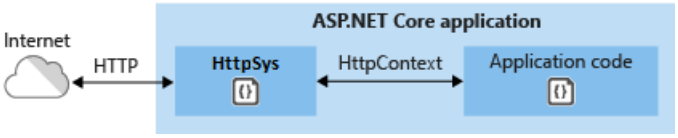
For information on how to use Nginx on Linux as a reverse proxy server for Kestrel, see Host ASP.NET Core on Linux with Nginx.

Apache with Kestrel

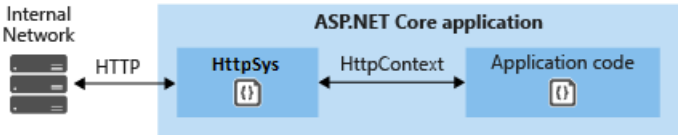
For information on how to use Apache on Linux as a reverse proxy server for Kestrel, see Host ASP.NET Core on Linux with Apache.

HTTP.sys

If ASP.NET Core apps are run on Windows, HTTP.sys is an alternative to Kestrel. Kestrel is generally recommended for best performance. HTTP.sys can be used in scenarios where the app is exposed to the Internet and required capabilities are supported by HTTP.sys but not Kestrel. For more information, see HTTP.sys web server implementation in ASP.NET Core.



HTTP.sys can also be used for apps that are only exposed to an internal network.



For HTTP.sys configuration guidance, see HTTP.sys web server implementation in ASP.NET Core.

ASP.NET Core server infrastructure

The IApplicationBuilder available in the `Startup.Configure` method exposes the ServerFeatures property of type IFeatureCollection. Kestrel and HTTP.sys only expose a single feature each, IServerAddressesFeature, but different server implementations may expose additional functionality.

`IServerAddressesFeature` can be used to find out which port the server implementation has bound at runtime.

Custom servers

If the built-in servers don't meet the app's requirements, a custom server implementation can be created. The Open Web Interface for .NET (OWIN) guide demonstrates how to write a Nowin-based IServer implementation. Only the feature interfaces that the app uses require implementation, though at a minimum IHttpRequestFeature and IHttpResponseFeature must be supported.

Server startup

The server is launched when the Integrated Development Environment (IDE) or editor starts the app:

- Visual Studio: Launch profiles can be used to start the app and server with either IIS Express/ASP.NET Core Module or the console.
- Visual Studio Code: The app and server are started by Omnisharp, which activates the CoreCLR debugger.
- Visual Studio for Mac: The app and server are started by the Mono Soft-Mode Debugger.

When launching the app from a command prompt in the project's folder, dotnet run launches the app and server

(Kestrel and HTTP.sys only). The configuration is specified by the `-c|--configuration` option, which is set to either `Debug` (default) or `Release`.

A *launchSettings.json* file provides configuration when launching an app with `dotnet run` or with a debugger built into tooling, such as Visual Studio. If launch profiles are present in a *launchSettings.json* file, use the `--launch-profile {PROFILE NAME}` option with the `dotnet run` command or select the profile in Visual Studio. For more information, see dotnet run and .NET Core distribution packaging.

HTTP/2 support

HTTP/2 is supported with ASP.NET Core in the following deployment scenarios:

- Kestrel
  - Operating system
    - Windows Server 2016/Windows 10 or later†
    - Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
    - HTTP/2 will be supported on macOS in a future release.
  - Target framework: .NET Core 2.2 or later
- HTTP.sys
  - Windows Server 2016/Windows 10 or later
  - Target framework: Not applicable to HTTP.sys deployments.
- IIS (in-process)
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Target framework: .NET Core 2.2 or later
- IIS (out-of-process)
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Public-facing edge server connections use HTTP/2, but the reverse proxy connection to Kestrel uses HTTP/1.1.
  - Target framework: Not applicable to IIS out-of-process deployments.

†Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

- HTTP.sys
  - Windows Server 2016/Windows 10 or later
  - Target framework: Not applicable to HTTP.sys deployments.
- IIS (out-of-process)
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Public-facing edge server connections use HTTP/2, but the reverse proxy connection to Kestrel uses HTTP/1.1.
  - Target framework: Not applicable to IIS out-of-process deployments.

An HTTP/2 connection must use Application-Layer Protocol Negotiation (ALPN) and TLS 1.2 or later. For more information, see the topics that pertain to your server deployment scenarios.

Additional resources

- Kestrel web server implementation in ASP.NET Core
- ASP.NET Core Module

- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service
- Host ASP.NET Core on Linux with Nginx
- Host ASP.NET Core on Linux with Apache
- HTTP.sys web server implementation in ASP.NET Core

Configuration in ASP.NET Core

9/22/2020 • 60 minutes to read • Edit Online

By Rick Anderson and Kirk Larkin

Configuration in ASP.NET Core is performed using one or more configuration providers. Configuration providers read configuration data from key-value pairs using a variety of configuration sources:

- Settings files, such as *appsettings.json*
- Environment variables
- Azure Key Vault
- Azure App Configuration
- Command-line arguments
- Custom providers, installed or created
- Directory files
- In-memory .NET objects

View or download sample code (how to download)

Default configuration

ASP.NET Core web apps created with dotnet new or Visual Studio generate the following code:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

CreateDefaultBuilder provides default configuration for the app in the following order:

1. ChainedConfigurationProvider : Adds an existing `IConfiguration` as a source. In the default configuration case, adds the host configuration and setting it as the first source for the *app* configuration.
2. appsettings.json using the JSON configuration provider.
3. *appsettings.*`Environment`*.json* using the JSON configuration provider. For example, *appsettings.**Production**.json* and *appsettings.**Development**.json*.
4. App secrets when the app runs in the `Development` environment.
5. Environment variables using the Environment Variables configuration provider.
6. Command-line arguments using the Command-line configuration provider.

Configuration providers that are added later override previous key settings. For example, if

`MyKey` is set in both *appsettings.json* and the environment, the environment value is used. Using the default configuration providers, the Command-line configuration provider overrides all other providers.

For more information on `CreateDefaultBuilder`, see Default builder settings.

The following code displays the enabled configuration providers in the order they were added:

```
public class Index2Model : PageModel
{
    private IConfigurationRoot ConfigRoot;

    public Index2Model(IConfiguration configRoot)
    {
        ConfigRoot = (IConfigurationRoot)configRoot;
    }

    public ContentResult OnGet()
    {
        string str = "";
        foreach (var provider in ConfigRoot.Providers.ToList())
        {
            str += provider.ToString() + "\n";
        }

        return Content(str);
    }
}
```

appsettings.json

Consider the following *appsettings.json* file:

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "MyKey":  "My appsettings.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

The following code from the sample download displays several of the preceding configurations settings:

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

The default JsonConfigurationProvider loads configuration in the following order:

1. *appsettings.json*
2. *appsettings.*`Environment`*.json* : For example, the *appsettings.**Production**.json* and *appsettings.**Development**.json* files. The environment version of the file is loaded based on the IHostingEnvironment.EnvironmentName. For more information, see Use multiple environments in ASP.NET Core.

appsettings.`Environment`*.json* values override keys in *appsettings.json*. For example, by default:

- In development, *appsettings.**Development**.json* configuration overwrites values found in *appsettings.json*.
- In production, *appsettings.**Production**.json* configuration overwrites values found in *appsettings.json*. For example, when deploying the app to Azure.

Bind hierarchical configuration data using the options pattern

The preferred way to read related configuration values is using the options pattern. For example, to read the following configuration values:

```
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  }
```

Create the following `PositionOptions` class:

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; }
    public string Name { get; set; }
}
```

An options class:

- Must be non-abstract with a public parameterless constructor.
- All public read-write properties of the type are bound.
- Fields are ***not*** bound. In the preceding code, `Position` is not bound. The `Position` property is used so the string `"Position"` doesn't need to be hard coded in the app when binding the class to a configuration provider.

The following code:

- Calls ConfigurationBinder.Bind to bind the `PositionOptions` class to the `Position` section.
- Displays the `Position` configuration data.

```
public class Test22Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test22Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var positionOptions = new PositionOptions();
        Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

`ConfigurationBinder.Get<T>` binds and returns the specified type. `ConfigurationBinder.Get<T>` may be more convenient than using `ConfigurationBinder.Bind`. The following code shows how to use `ConfigurationBinder.Get<T>` with the `PositionOptions` class:

```
public class Test21Model : PageModel
{
    private readonly IConfiguration Configuration;
    public PositionOptions positionOptions { get; private set; }

    public Test21Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        positionOptions = Configuration.GetSection(PositionOptions.Position)
                                                    .Get<PositionOptions>();

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has

started are read.

An alternative approach when using the ***options pattern*** is to bind the `Position` section and add it to the dependency injection service container. In the following code, `PositionOptions` is added to the service container with Configure and bound to configuration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<PositionOptions>(Configuration.GetSection(
                                        PositionOptions.Position));
    services.AddRazorPages();
}
```

Using the preceding code, the following code reads the position options:

```
public class Test2Model : PageModel
{
    private readonly PositionOptions _options;

    public Test2Model(IOptions<PositionOptions> options)
    {
        _options = options.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Title: {_options.Title} \n" +
                       $"Name: {_options.Name}");
    }
}
```

In the preceding code, changes to the JSON configuration file after the app has started are ***not*** read. To read changes after the app has started, use IOptionsSnapshot.

Using the default configuration, the *appsettings.json* and *appsettings.*`Environment`*.json* files are enabled with reloadOnChange: true. Changes made to the *appsettings.json* and *appsettings.*`Environment`*.json* file ***after*** the app starts are read by the JSON configuration provider.

See JSON configuration provider in this document for information on adding additional JSON configuration files.

Combining service collection

Consider the following `ConfigureServices` method, which registers services and configures options:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<PositionOptions>(
        Configuration.GetSection(PositionOptions.Position));
    services.Configure<ColorOptions>(
        Configuration.GetSection(ColorOptions.Color));

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddScoped<IMyDependency2, MyDependency2>();

    services.AddRazorPages();
}
```

Related groups of registrations can be moved to an extension method to register services. For example, the configuration services are added to the following class:

```
using ConfigSample.Options;
using Microsoft.Extensions.Configuration;

namespace Microsoft.Extensions.DependencyInjection
{
    public static class MyConfigServiceCollectionExtensions
    {
        public static IServiceCollection AddConfig(
             this IServiceCollection services, IConfiguration config)
        {
            services.Configure<PositionOptions>(
                config.GetSection(PositionOptions.Position));
            services.Configure<ColorOptions>(
                config.GetSection(ColorOptions.Color));

            return services;
        }
    }
}
```

The remaining services are registered in a similar class. The following `ConfigureServices` method uses the new extension methods to register the services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddConfig(Configuration)
            .AddMyDependencyGroup();

    services.AddRazorPages();
}
```

Note: Each `services.Add{GROUP_NAME}` extension method adds and potentially configures services. For example, AddControllersWithViews adds the services MVC controllers with views require, and AddRazorPages adds the services Razor Pages requires. We recommended that apps follow this naming convention. Place extension methods in the Microsoft.Extensions.DependencyInjection namespace to encapsulate groups of service registrations.

Security and secret manager

Configuration data guidelines:

- Never store passwords or other sensitive data in configuration provider code or in plain text configuration files. The Secret manager can be used to store secrets in development.
- Don't use production secrets in development or test environments.
- Specify secrets outside of the project so that they can't be accidentally committed to a source code repository.

By default, Secret manager reads configuration settings after *appsettings.json* and *appsettings.* `Environment` *.json*.

For more information on storing passwords or other sensitive data:

- Use multiple environments in ASP.NET Core
- Safe storage of app secrets in development in ASP.NET Core: Includes advice on using

environment variables to store sensitive data. The Secret Manager uses the File configuration provider to store user secrets in a JSON file on the local system.

Azure Key Vault safely stores app secrets for ASP.NET Core apps. For more information, see Azure Key Vault Configuration Provider in ASP.NET Core.

Environment variables

Using the default configuration, the EnvironmentVariablesConfigurationProvider loads configuration from environment variable key-value pairs after reading *appsettings.json*, *appsettings.*`Environment`*.json*, and Secret manager. Therefore, key values read from the environment override values read from *appsettings.json*, *appsettings.*`Environment`*.json*, and Secret manager.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

The following `set` commands:

- Set the environment keys and values of the preceding example on Windows.
- Test the settings when using the sample download. The `dotnet run` command must be run in the project directory.

```
set MyKey="My key from Environment"
set Position__Title=Environment_Editor
set Position__Name=Environment_Rick
dotnet run
```

The preceding environment settings:

- Are only set in processes launched from the command window they were set in.
- Won't be read by browsers launched with Visual Studio.

The following setx commands can be used to set the environment keys and values on Windows. Unlike `set`, `setx` settings are persisted. `/M` sets the variable in the system environment. If the `/M` switch isn't used, a user environment variable is set.

```
setx MyKey "My key from setx Environment" /M
setx Position__Title Setx_Environment_Editor /M
setx Position__Name Environment_Rick /M
```

To test that the preceding commands override *appsettings.json* and *appsettings.*`Environment`*.json*:

- With Visual Studio: Exit and restart Visual Studio.
- With the CLI: Start a new command window and enter `dotnet run`.

Call AddEnvironmentVariables with a string to specify a prefix for environment variables:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddEnvironmentVariables(prefix: "MyCustomPrefix_");
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

In the preceding code:

- `config.AddEnvironmentVariables(prefix: "MyCustomPrefix_")` is added after the default configuration providers. For an example of ordering the configuration providers, see JSON configuration provider.
- Environment variables set with the `MyCustomPrefix_` prefix override the default configuration providers. This includes environment variables without the prefix.

The prefix is stripped off when the configuration key-value pairs are read.

The following commands test the custom prefix:

```
set MyCustomPrefix_MyKey="My key with MyCustomPrefix_ Environment"
set MyCustomPrefix_Position__Title=Editor_with_customPrefix
set MyCustomPrefix_Position__Name=Environment_Rick_cp
dotnet run
```

The default configuration loads environment variables and command line arguments prefixed with `DOTNET_` and `ASPNETCORE_`. The `DOTNET_` and `ASPNETCORE_` prefixes are used by ASP.NET Core for host and app configuration, but not for user configuration. For more information on host and app configuration, see .NET Generic Host.

On Azure App Service, select **New application setting** on the **Settings > Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

For more information, see Azure Apps: Override app configuration using the Azure Portal.

See Connection string prefixes for information on Azure database connection strings.

Environment variables set in launchSettings.json

Environment variables set in *launchSettings.json* override those set in the system environment.

Command-line

Using the default configuration, the CommandLineConfigurationProvider loads configuration from command-line argument key-value pairs after the following configuration sources:

- *appsettings.json* and *appsettings.*`Environment`*.json* files.
- App secrets (Secret Manager) in the Development environment.
- Environment variables.

By default, configuration values set on the command-line override configuration values set with all the other configuration providers.

Command-line arguments

The following command sets keys and values using `=`:

```
dotnet run MyKey="My key from command line" Position:Title=Cmd Position:Name=Cmd_Rick
```

The following command sets keys and values using `/`:

```
dotnet run /MyKey "Using /" /Position:Title=Cmd_ /Position:Name=Cmd_Rick
```

The following command sets keys and values using `--`:

```
dotnet run --MyKey "Using --" --Position:Title=Cmd-- --Position:Name=Cmd--Rick
```

The key value:

- Must follow `=`, or the key must have a prefix of `--` or `/` when the value follows a space.
- Isn't required if `=` is used. For example, `MySetting=`.

Within the same command, don't mix command-line argument key-value pairs that use `=` with key-value pairs that use a space.

Switch mappings

Switch mappings allow **key** name replacement logic. Provide a dictionary of switch replacements to the AddCommandLine method.

When the switch mappings dictionary is used, the dictionary is checked for a key that matches the key provided by a command-line argument. If the command-line key is found in the dictionary, the dictionary value is passed back to set the key-value pair into the app's configuration. A switch mapping is required for any command-line key prefixed with a single dash (`-`).

Switch mappings dictionary key rules:

- Switches must start with `-` or `--`.
- The switch mappings dictionary must not contain duplicate keys.

To use a switch mappings dictionary, pass it into the call to `AddCommandLine`:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        var switchMappings = new Dictionary<string, string>()
         {
             { "-k1", "key1" },
             { "-k2", "key2" },
             { "--alt3", "key3" },
             { "--alt4", "key4" },
             { "--alt5", "key5" },
             { "--alt6", "key6" },
         };

        return Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddCommandLine(args, switchMappings);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
    }
}
```

The following code shows the key values for the replaced keys:

```
public class Test3Model : PageModel
{
    private readonly IConfiguration Config;

    public Test3Model(IConfiguration configuration)
    {
        Config = configuration;
    }

    public ContentResult OnGet()
    {
        return Content(
                $"Key1: '{Config["Key1"]}'\n" +
                $"Key2: '{Config["Key2"]}'\n" +
                $"Key3: '{Config["Key3"]}'\n" +
                $"Key4: '{Config["Key4"]}'\n" +
                $"Key5: '{Config["Key5"]}'\n" +
                $"Key6: '{Config["Key6"]}'");
    }
}
```

The following command works to test key replacement:

```
dotnet run -k1 value1 -k2 value2 --alt3=value2 /alt4=value3 --alt5 value5 /alt6 value6
```

For apps that use switch mappings, the call to `CreateDefaultBuilder` shouldn't pass arguments. The `CreateDefaultBuilder` method's `AddCommandLine` call doesn't include mapped switches, and there's no way to pass the switch-mapping dictionary to `CreateDefaultBuilder`. The solution isn't to pass the arguments to `CreateDefaultBuilder` but instead to allow the `ConfigurationBuilder`

method's `AddCommandLine` method to process both the arguments and the switch-mapping dictionary.

Hierarchical configuration data

The Configuration API reads hierarchical configuration data by flattening the hierarchical data with the use of a delimiter in the configuration keys.

The sample download contains the following *appsettings.json* file:

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "MyKey":  "My appsettings.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

The following code from the sample download displays several of the configurations settings:

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

The preferred way to read hierarchical configuration data is using the options pattern. For more information, see Bind hierarchical configuration data in this document.

GetSection and GetChildren methods are available to isolate sections and children of a section in the configuration data. These methods are described later in GetSection, GetChildren, and Exists.

Configuration keys and values

Configuration keys:

- Are case-insensitive. For example, `ConnectionString` and `connectionstring` are treated as equivalent keys.
- If a key and value is set in more than one configuration providers, the value from the last provider added is used. For more information, see Default configuration.
- Hierarchical keys
  - Within the Configuration API, a colon separator (`:`) works on all platforms.
  - In environment variables, a colon separator may not work on all platforms. A double underscore, `__`, is supported by all platforms and is automatically converted into a colon `:`.
  - In Azure Key Vault, hierarchical keys use `--` as a separator. The Azure Key Vault configuration provider automatically replaces `--` with a `:` when the secrets are loaded into the app's configuration.
- The ConfigurationBinder supports binding arrays to objects using array indices in configuration keys. Array binding is described in the Bind an array to a class section.

Configuration values:

- Are strings.
- Null values can't be stored in configuration or bound to objects.

Configuration providers

The following table shows the configuration providers available to ASP.NET Core apps.

PROVIDER	PROVIDES CONFIGURATION FROM
Azure Key Vault configuration provider	Azure Key Vault
Azure App configuration provider	Azure App Configuration
Command-line configuration provider	Command-line parameters
Custom configuration provider	Custom source
Environment Variables configuration provider	Environment variables
File configuration provider	INI, JSON, and XML files
Key-per-file configuration provider	Directory files
Memory configuration provider	In-memory collections
Secret Manager	File in the user profile directory

Configuration sources are read in the order that their configuration providers are specified. Order configuration providers in code to suit the priorities for the underlying configuration sources that the app requires.

A typical sequence of configuration providers is:

1. *appsettings.json*

2. *appsettings.*`Environment`*.json*
3. Secret Manager
4. Environment variables using the Environment Variables configuration provider.
5. Command-line arguments using the Command-line configuration provider.

A common practice is to add the Command-line configuration provider last in a series of providers to allow command-line arguments to override configuration set by the other providers.

The preceding sequence of providers is used in the default configuration.

Connection string prefixes

The Configuration API has special processing rules for four connection string environment variables. These connection strings are involved in configuring Azure connection strings for the app environment. Environment variables with the prefixes shown in the table are loaded into the app with the default configuration or when no prefix is supplied to `AddEnvironmentVariables`.

CONNECTION STRING PREFIX	PROVIDER
`CUSTOMCONNSTR_`	Custom provider
`MYSQLCONNSTR_`	MySQL
`SQLAZURECONNSTR_`	Azure SQL Database
`SQLCONNSTR_`	SQL Server

When an environment variable is discovered and loaded into configuration with any of the four prefixes shown in the table:

- The configuration key is created by removing the environment variable prefix and adding a configuration key section (`ConnectionStrings`).
- A new configuration key-value pair is created that represents the database connection provider (except for `CUSTOMCONNSTR_`, which has no stated provider).

ENVIRONMENT VARIABLE KEY	CONVERTED CONFIGURATION KEY	PROVIDER CONFIGURATION ENTRY
`CUSTOMCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Configuration entry not created.
`MYSQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` : Value: `MySql.Data.MySqlClient`
`SQLAZURECONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` : Value: `System.Data.SqlClient`
`SQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` : Value: `System.Data.SqlClient`

ENVIRONMENT VARIABLE KEY	CONVERTED CONFIGURATION KEY	PROVIDER CONFIGURATION ENTRY
`SQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` : Value: `System.Data.SqlClient`

File configuration provider

FileConfigurationProvider is the base class for loading configuration from the file system. The following configuration providers derive from `FileConfigurationProvider`:

- INI configuration provider
- JSON configuration provider
- XML configuration provider

INI configuration provider

The IniConfigurationProvider loads configuration from INI file key-value pairs at runtime.

The following code clears all the configuration providers and adds several configuration providers:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.Sources.Clear();

                var env = hostingContext.HostingEnvironment;

                config.AddIniFile("MyIniConfig.ini", optional: true, reloadOnChange: true)
                      .AddIniFile($"MyIniConfig.{env.EnvironmentName}.ini",
                                  optional: true, reloadOnChange: true);

                config.AddEnvironmentVariables();

                if (args != null)
                {
                    config.AddCommandLine(args);
                }
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

In the preceding code, settings in the *MyIniConfig.ini* and *MyIniConfig.* `Environment` *.ini* files are overridden by settings in the:

- Environment variables configuration provider

- Command-line configuration provider.

The sample download contains the following *MyIniConfig.ini* file:

```
MyKey="MyIniConfig.ini Value"

[Position]
Title="My INI Config title"
Name="My INI Config name"

[Logging:LogLevel]
Default=Information
Microsoft=Warning
```

The following code from the sample download displays several of the preceding configurations settings:

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

JSON configuration provider

The JsonConfigurationProvider loads configuration from JSON file key-value pairs.

Overloads can specify:

- Whether the file is optional.
- Whether the configuration is reloaded if the file changes.

Consider the following code:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("MyConfig.json",
                    optional: true,
                    reloadOnChange: true);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The preceding code:

- Configures the JSON configuration provider to load the *MyConfig.json* file with the following options:
  - `optional: true` : The file is optional.
  - `reloadOnChange: true` : The file is reloaded when changes are saved.
- Reads the default configuration providers before the *MyConfig.json* file. Settings in the *MyConfig.json* file override setting in the default configuration providers, including the Environment variables configuration provider and the Command-line configuration provider.

You typically ***don't*** want a custom JSON file overriding values set in the Environment variables configuration provider and the Command-line configuration provider.

The following code clears all the configuration providers and adds several configuration providers:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.Sources.Clear();

                var env = hostingContext.HostingEnvironment;

                config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                      .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                                   optional: true, reloadOnChange: true);

                config.AddJsonFile("MyConfig.json", optional: true, reloadOnChange: true)
                      .AddJsonFile($"MyConfig.{env.EnvironmentName}.json",
                                   optional: true, reloadOnChange: true);

                config.AddEnvironmentVariables();

                if (args != null)
                {
                    config.AddCommandLine(args);
                }
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

In the preceding code, settings in the *MyConfig.json* and *MyConfig.*`Environment`*.json* files:

- Override settings in the *appsettings.json* and *appsettings.*`Environment`*.json* files.
- Are overridden by settings in the Environment variables configuration provider and the Command-line configuration provider.

The sample download contains the following *MyConfig.json* file:

```
{
  "Position": {
    "Title": "My Config title",
    "Name": "My Config Smith"
  },
  "MyKey":  "MyConfig.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

The following code from the sample download displays several of the preceding configurations settings:

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

XML configuration provider

The XmlConfigurationProvider loads configuration from XML file key-value pairs at runtime.

The following code clears all the configuration providers and adds several configuration providers:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.Sources.Clear();

                var env = hostingContext.HostingEnvironment;

                config.AddXmlFile("MyXMLFile.xml", optional: true, reloadOnChange: true)
                      .AddXmlFile($"MyXMLFile.{env.EnvironmentName}.xml",
                                  optional: true, reloadOnChange: true);

                config.AddEnvironmentVariables();

                if (args != null)
                {
                    config.AddCommandLine(args);
                }
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

In the preceding code, settings in the *MyXMLFile.xml* and *MyXMLFile.`Environment`.xml* files are overridden by settings in the:

- Environment variables configuration provider
- Command-line configuration provider.

The sample download contains the following *MyXMLFile.xml* file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <MyKey>MyXMLFile Value</MyKey>
  <Position>
    <Title>Title from  MyXMLFile</Title>
    <Name>Name from MyXMLFile</Name>
  </Position>
  <Logging>
    <LogLevel>
      <Default>Information</Default>
      <Microsoft>Warning</Microsoft>
    </LogLevel>
  </Logging>
</configuration>
```

The following code from the sample download displays several of the preceding configurations settings:

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

Repeating elements that use the same element name work if the `name` attribute is used to distinguish the elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="section0">
    <key name="key0">value 00</key>
    <key name="key1">value 01</key>
  </section>
  <section name="section1">
    <key name="key0">value 10</key>
    <key name="key1">value 11</key>
  </section>
</configuration>
```

The following code reads the previous configuration file and displays the keys and values:

```
public class IndexModel : PageModel
{
    private readonly IConfiguration Configuration;

    public IndexModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var key00 = "section:section0:key:key0";
        var key01 = "section:section0:key:key1";
        var key10 = "section:section1:key:key0";
        var key11 = "section:section1:key:key1";

        var val00 = Configuration[key00];
        var val01 = Configuration[key01];
        var val10 = Configuration[key10];
        var val11 = Configuration[key11];

        return Content($"{key00} value: {val00} \n" +
                       $"{key01} value: {val01} \n" +
                       $"{key10} value: {val10} \n" +
                       $"{key10} value: {val11} \n"
                       );
    }
}
```

Attributes can be used to supply values:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
  </section>
</configuration>
```

The previous configuration file loads the following keys with `value`:

- key:attribute
- section:key:attribute

Key-per-file configuration provider

The KeyPerFileConfigurationProvider uses a directory's files as configuration key-value pairs.

The key is the file name. The value contains the file's contents. The Key-per-file configuration provider is used in Docker hosting scenarios.

To activate key-per-file configuration, call the AddKeyPerFile extension method on an instance of ConfigurationBuilder. The `directoryPath` to the files must be an absolute path.

Overloads permit specifying:

- An `Action<KeyPerFileConfigurationSource>` delegate that configures the source.
- Whether the directory is optional and the path to the directory.

The double-underscore (`__`) is used as a configuration key delimiter in file names. For example, the file name `Logging__LogLevel__System` produces the configuration key `Logging:LogLevel:System`.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    var path = Path.Combine(
        Directory.GetCurrentDirectory(), "path/to/files");
    config.AddKeyPerFile(directoryPath: path, optional: true);
})
```

Memory configuration provider

The MemoryConfigurationProvider uses an in-memory collection as configuration key-value pairs.

The following code adds a memory collection to the configuration system:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        var Dict = new Dictionary<string, string>
        {
           {"MyKey", "Dictionary MyKey Value"},
           {"Position:Title", "Dictionary_Title"},
           {"Position:Name", "Dictionary_Name" },
           {"Logging:LogLevel:Default", "Warning"}
        };

        return Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddInMemoryCollection(Dict);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
    }
}
```

The following code from the sample download displays the preceding configurations settings:

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

In the preceding code, `config.AddInMemoryCollection(Dict)` is added after the default configuration providers. For an example of ordering the configuration providers, see JSON configuration provider.

See Bind an array for another example using `MemoryConfigurationProvider`.

GetValue

`ConfigurationBinder.GetValue<T>` extracts a single value from configuration with a specified key and converts it to the specified type:

```
public class TestNumModel : PageModel
{
    private readonly IConfiguration Configuration;

    public TestNumModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var number = Configuration.GetValue<int>("NumberKey", 99);
        return Content($"{number}");
    }
}
```

In the preceding code, if `NumberKey` isn't found in the configuration, the default value of `99` is used.

GetSection, GetChildren, and Exists

For the examples that follow, consider the following *MySubsection.json* file:

```
{
  "section0": {
    "key0": "value00",
    "key1": "value01"
  },
  "section1": {
    "key0": "value10",
    "key1": "value11"
  },
  "section2": {
    "subsection0": {
      "key0": "value200",
      "key1": "value201"
    },
    "subsection1": {
      "key0": "value210",
      "key1": "value211"
    }
  }
}
```

The following code adds *MySubsection.json* to the configuration providers:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("MySubsection.json",
                    optional: true,
                    reloadOnChange: true);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

GetSection

IConfiguration.GetSection returns a configuration subsection with the specified subsection key.

The following code returns values for `section1`:

```
public class TestSectionModel : PageModel
{
    private readonly IConfiguration Config;

    public TestSectionModel(IConfiguration configuration)
    {
        Config = configuration.GetSection("section1");
    }

    public ContentResult OnGet()
    {
        return Content(
                $"section1:key0: '{Config["key0"]}'\n" +
                $"section1:key1: '{Config["key1"]}'");
    }
}
```

The following code returns values for `section2:subsection0`:

```
public class TestSection2Model : PageModel
{
    private readonly IConfiguration Config;

    public TestSection2Model(IConfiguration configuration)
    {
        Config = configuration.GetSection("section2:subsection0");
    }

    public ContentResult OnGet()
    {
        return Content(
                $"section2:subsection0:key0 '{Config["key0"]}'\n" +
                $"section2:subsection0:key1:'{Config["key1"]}'");
    }
}
```

`GetSection` never returns `null`. If a matching section isn't found, an empty `IConfigurationSection` is returned.

When `GetSection` returns a matching section, Value isn't populated. A Key and Path are returned when the section exists.

GetChildren and Exists

The following code calls IConfiguration.GetChildren and returns values for `section2:subsection0`:

```
public class TestSection4Model : PageModel
{
    private readonly IConfiguration Config;

    public TestSection4Model(IConfiguration configuration)
    {
        Config = configuration;
    }

    public ContentResult OnGet()
    {
        string s = null;
        var selection = Config.GetSection("section2");
        if (!selection.Exists())
        {
            throw new System.Exception("section2 does not exist.");
        }
        var children = selection.GetChildren();

        foreach (var subSection in children)
        {
            int i = 0;
            var key1 = subSection.Key + ":key" + i++.ToString();
            var key2 = subSection.Key + ":key" + i.ToString();
            s += key1 + " value: " + selection[key1] + "\n";
            s += key2 + " value: " + selection[key2] + "\n";
        }
        return Content(s);
    }
}
```

The preceding code calls ConfigurationExtensions.Exists to verify the section exists:

Bind an array

The ConfigurationBinder.Bind supports binding arrays to objects using array indices in configuration keys. Any array format that exposes a numeric key segment is capable of array binding to a POCO class array.

Consider *MyArray.json* from the sample download:

```
{
  "array": {
    "entries": {
      "0": "value00",
      "1": "value10",
      "2": "value20",
      "4": "value40",
      "5": "value50"
    }
  }
}
```

The following code adds *MyArray.json* to the configuration providers:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("MyArray.json",
                    optional: true,
                    reloadOnChange: true);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code reads the configuration and displays the values:

```
public class ArrayModel : PageModel
{
    private readonly IConfiguration Config;
    public ArrayExample _array { get; private set; }

    public ArrayModel(IConfiguration config)
    {
        Config = config;
    }

    public ContentResult OnGet()
    {
        _array = Config.GetSection("array").Get<ArrayExample>();
        string s = null;

        for (int j = 0; j < _array.Entries.Length; j++)
        {
            s += $"Index: {j}  Value:  {_array.Entries[j]} \n";
        }

        return Content(s);
    }
}
```

The preceding code returns the following output:

```
Index: 0  Value: value00
Index: 1  Value: value10
Index: 2  Value: value20
Index: 3  Value: value40
Index: 4  Value: value50
```

In the preceding output, Index 3 has value `value40`, corresponding to `"4": "value40",` in *MyArray.json*. The bound array indices are continuous and not bound to the configuration key index. The configuration binder isn't capable of binding null values or creating null entries in bound objects

The following code loads the `array:entries` configuration with the AddInMemoryCollection

extension method:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        var arrayDict = new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            //              3   Skipped
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

        return Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddInMemoryCollection(arrayDict);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
    }
}
```

The following code reads the configuration in the `arrayDict` `Dictionary` and displays the values:

```
public class ArrayModel : PageModel
{
    private readonly IConfiguration Config;
    public ArrayExample _array { get; private set; }

    public ArrayModel(IConfiguration config)
    {
        Config = config;
    }

    public ContentResult OnGet()
    {
        _array = Config.GetSection("array").Get<ArrayExample>();
        string s = null;

        for (int j = 0; j < _array.Entries.Length; j++)
        {
            s += $"Index: {j}  Value:  {_array.Entries[j]} \n";
        }

        return Content(s);
    }
}
```

The preceding code returns the following output:

```
Index: 0  Value: value0
Index: 1  Value: value1
Index: 2  Value: value2
Index: 3  Value: value4
Index: 4  Value: value5
```

Index #3 in the bound object holds the configuration data for the `array:4` configuration key and its value of `value4`. When configuration data containing an array is bound, the array indices in the configuration keys are used to iterate the configuration data when creating the object. A null value can't be retained in configuration data, and a null-valued entry isn't created in a bound object when an array in configuration keys skip one or more indices.

The missing configuration item for index #3 can be supplied before binding to the `ArrayExample` instance by any configuration provider that reads the index #3 key/value pair. Consider the following *Value3.json* file from the sample download:

```
{
  "array:entries:3": "value3"
}
```

The following code includes configuration for *Value3.json* and the `arrayDict` `Dictionary`:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        var arrayDict = new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            //              3   Skipped
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

        return Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddInMemoryCollection(arrayDict);
                config.AddJsonFile("Value3.json",
                                   optional: false, reloadOnChange: false);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
    }
}
```

The following code reads the preceding configuration and displays the values:

```
public class ArrayModel : PageModel
{
    private readonly IConfiguration Config;
    public ArrayExample _array { get; private set; }

    public ArrayModel(IConfiguration config)
    {
        Config = config;
    }

    public ContentResult OnGet()
    {
        _array = Config.GetSection("array").Get<ArrayExample>();
        string s = null;

        for (int j = 0; j < _array.Entries.Length; j++)
        {
            s += $"Index: {j}  Value:  {_array.Entries[j]} \n";
        }

        return Content(s);
    }
}
```

The preceding code returns the following output:

```
Index: 0  Value: value0
Index: 1  Value: value1
Index: 2  Value: value2
Index: 3  Value: value3
Index: 4  Value: value4
Index: 5  Value: value5
```

Custom configuration providers aren't required to implement array binding.

Custom configuration provider

The sample app demonstrates how to create a basic configuration provider that reads configuration key-value pairs from a database using Entity Framework (EF).

The provider has the following characteristics:

- The EF in-memory database is used for demonstration purposes. To use a database that requires a connection string, implement a secondary `ConfigurationBuilder` to supply the connection string from another configuration provider.
- The provider reads a database table into configuration at startup. The provider doesn't query the database on a per-key basis.
- Reload-on-change isn't implemented, so updating the database after the app starts has no effect on the app's configuration.

Define an `EFConfigurationValue` entity for storing configuration values in the database.

Models/EFConfigurationValue.cs:

```
public class EFConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}
```

Add an `EFConfigurationContext` to store and access the configured values.

EFConfigurationProvider/EFConfigurationContext.cs:

```
// using Microsoft.EntityFrameworkCore;

public class EFConfigurationContext : DbContext
{
    public EFConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<EFConfigurationValue> Values { get; set; }
}
```

Create a class that implements IConfigurationSource.

EFConfigurationProvider/EFConfigurationSource.cs:

```
// using Microsoft.EntityFrameworkCore;
// using Microsoft.Extensions.Configuration;

public class EFConfigurationSource : IConfigurationSource
{
    private readonly Action<DbContextOptionsBuilder> _optionsAction;

    public EFConfigurationSource(Action<DbContextOptionsBuilder> optionsAction)
    {
        _optionsAction = optionsAction;
    }

    public IConfigurationProvider Build(IConfigurationBuilder builder)
    {
        return new EFConfigurationProvider(_optionsAction);
    }
}
```

Create the custom configuration provider by inheriting from ConfigurationProvider. The configuration provider initializes the database when it's empty. Since configuration keys are case-insensitive, the dictionary used to initialize the database is created with the case-insensitive comparer (StringComparer.OrdinalIgnoreCase).

EFConfigurationProvider/EFConfigurationProvider.cs:

```
// using Microsoft.EntityFrameworkCore;
// using Microsoft.Extensions.Configuration;

public class EFConfigurationProvider : ConfigurationProvider
{
    public EFConfigurationProvider(Action<DbContextOptionsBuilder> optionsAction)
    {
        OptionsAction = optionsAction;
    }

    Action<DbContextOptionsBuilder> OptionsAction { get; }

    public override void Load()
    {
        var builder = new DbContextOptionsBuilder<EFConfigurationContext>();

        OptionsAction(builder);

        using (var dbContext = new EFConfigurationContext(builder.Options))
        {
            dbContext.Database.EnsureCreated();

            Data = !dbContext.Values.Any()
                ? CreateAndSaveDefaultValues(dbContext)
                : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
        }
    }

    private static IDictionary<string, string> CreateAndSaveDefaultValues(
        EFConfigurationContext dbContext)
    {
        // Quotes (c)2005 Universal Pictures: Serenity
        // https://www.uphe.com/movies/serenity
        var configValues =
            new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase)
            {
                { "quote1", "I aim to misbehave." },
                { "quote2", "I swallowed a bug." },
                { "quote3", "You can't stop the signal, Mal." }
            };

        dbContext.Values.AddRange(configValues
            .Select(kvp => new EFConfigurationValue
                {
                    Id = kvp.Key,
                    Value = kvp.Value
                })
            .ToArray());

        dbContext.SaveChanges();

        return configValues;
    }
}
```

An `AddEFConfiguration` extension method permits adding the configuration source to a `ConfigurationBuilder`.

Extensions/EntityFrameworkExtensions.cs:

```
// using Microsoft.EntityFrameworkCore;
// using Microsoft.Extensions.Configuration;

public static class EntityFrameworkExtensions
{
    public static IConfigurationBuilder AddEFConfiguration(
        this IConfigurationBuilder builder,
        Action<DbContextOptionsBuilder> optionsAction)
    {
        return builder.Add(new EFConfigurationSource(optionsAction));
    }
}
```

The following code shows how to use the custom `EFConfigurationProvider` in *Program.cs*:

```
// using Microsoft.EntityFrameworkCore;

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            config.AddEFConfiguration(
                options => options.UseInMemoryDatabase("InMemoryDb"));
        })
```

Access configuration in Startup

The following code displays configuration data in `Startup` methods:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
        Console.WriteLine($"MyKey : {Configuration["MyKey"]}");
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        Console.WriteLine($"Position:Title : {Configuration["Position:Title"]}");

        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

For an example of accessing configuration using startup convenience methods, see App startup: Convenience methods.

Access configuration in Razor Pages

The following code displays configuration data in a Razor Page:

```
@page
@model Test5Model
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

Configuration value for 'MyKey': @Configuration["MyKey"]
```

In the following code, `MyOptions` is added to the service container with Configure and bound to configuration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MyOptions>(Configuration.GetSection("MyOptions"));

    services.AddRazorPages();
}
```

The following markup uses the `@inject` Razor directive to resolve and display the options values:

```
@page
@model SampleApp.Pages.Test3Model
@using Microsoft.Extensions.Options
@inject IOptions<MyOptions> optionsAccessor


<p><b>Option1:</b> @optionsAccessor.Value.Option1</p>
<p><b>Option2:</b> @optionsAccessor.Value.Option2</p>
```

Access configuration in a MVC view file

The following code displays configuration data in a MVC view:

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

Configuration value for 'MyKey': @Configuration["MyKey"]
```

Configure options with a delegate

Options configured in a delegate override values set in the configuration providers.

Configuring options with a delegate is demonstrated as Example 2 in the sample app.

In the following code, an IConfigureOptions<TOptions> service is added to the service container. It uses a delegate to configure values for `MyOptions`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MyOptions>(myOptions =>
    {
        myOptions.Option1 = "Value configured in delegate";
        myOptions.Option2 = 500;
    });

    services.AddRazorPages();
}
```

The following code displays the options values:

```
public class Test2Model : PageModel
{
    private readonly IOptions<MyOptions> _optionsDelegate;

    public Test2Model(IOptions<MyOptions> optionsDelegate )
    {
        _optionsDelegate = optionsDelegate;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_optionsDelegate.Value.Option1} \n" +
                       $"Option2: {_optionsDelegate.Value.Option2}");
    }
}
```

In the preceding example, the values of `Option1` and `Option2` are specified in *appsettings.json* and then overridden by the configured delegate.

Host versus app configuration

Before the app is configured and started, a *host* is configured and launched. The host is responsible for app startup and lifetime management. Both the app and the host are configured using the configuration providers described in this topic. Host configuration key-value pairs are also included in the app's configuration. For more information on how the configuration providers are used when the host is built and how configuration sources affect host configuration, see ASP.NET Core fundamentals.

Default host configuration

For details on the default configuration when using the Web Host, see the .

- Host configuration is provided from:
  - Environment variables prefixed with `DOTNET_` (for example, `DOTNET_ENVIRONMENT`) using the Environment Variables configuration provider. The prefix (`DOTNET_`) is stripped when the configuration key-value pairs are loaded.
  - Command-line arguments using the Command-line configuration provider.
- Web Host default configuration is established (`ConfigureWebHostDefaults`):
  - Kestrel is used as the web server and configured using the app's configuration providers.
  - Add Host Filtering Middleware.
  - Add Forwarded Headers Middleware if the `ASPNETCORE_FORWARDEDHEADERS_ENABLED` environment variable is set to `true`.
  - Enable IIS integration.

Other configuration

This topic only pertains to *app configuration*. Other aspects of running and hosting ASP.NET Core apps are configured using configuration files not covered in this topic:

- *launch.json/launchSettings.json* are tooling configuration files for the Development environment, described:
  - In Use multiple environments in ASP.NET Core.
  - Across the documentation set where the files are used to configure ASP.NET Core apps for Development scenarios.

- *web.config* is a server configuration file, described in the following topics:
  - Host ASP.NET Core on Windows with IIS
  - ASP.NET Core Module

Environment variables set in *launchSettings.json* override those set in the system environment.

For more information on migrating app configuration from earlier versions of ASP.NET, see Migrate from ASP.NET to ASP.NET Core.

Add configuration from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Startup` class. For more information, see Use hosting startup assemblies in ASP.NET Core.

Additional resources

- Configuration source code
- Options pattern in ASP.NET Core
- ASP.NET Core Blazor configuration

App configuration in ASP.NET Core is based on key-value pairs established by *configuration providers*. Configuration providers read configuration data into key-value pairs from a variety of configuration sources:

- Azure Key Vault
- Azure App Configuration
- Command-line arguments
- Custom providers (installed or created)
- Directory files
- Environment variables
- In-memory .NET objects
- Settings files

Configuration packages for common configuration provider scenarios (Microsoft.Extensions.Configuration) are included in the Microsoft.AspNetCore.App metapackage.

Code examples that follow and in the sample app use the Microsoft.Extensions.Configuration namespace:

```
using Microsoft.Extensions.Configuration;
```

The *options pattern* is an extension of the configuration concepts described in this topic. Options use classes to represent groups of related settings. For more information, see Options pattern in ASP.NET Core.

View or download sample code (how to download)

Host versus app configuration

Before the app is configured and started, a *host* is configured and launched. The host is responsible for app startup and lifetime management. Both the app and the host are configured using the configuration providers described in this topic. Host configuration key-value pairs are

also included in the app's configuration. For more information on how the configuration providers are used when the host is built and how configuration sources affect host configuration, see ASP.NET Core fundamentals.

Other configuration

This topic only pertains to *app configuration*. Other aspects of running and hosting ASP.NET Core apps are configured using configuration files not covered in this topic:

- *launch.json/launchSettings.json* are tooling configuration files for the Development environment, described:
  - In Use multiple environments in ASP.NET Core.
  - Across the documentation set where the files are used to configure ASP.NET Core apps for Development scenarios.
- *web.config* is a server configuration file, described in the following topics:
  - Host ASP.NET Core on Windows with IIS
  - ASP.NET Core Module

For more information on migrating app configuration from earlier versions of ASP.NET, see Migrate from ASP.NET to ASP.NET Core.

Default configuration

Web apps based on the ASP.NET Core dotnet new templates call CreateDefaultBuilder when building a host. `CreateDefaultBuilder` provides default configuration for the app in the following order:

The following applies to apps using the Web Host. For details on the default configuration when using the Generic Host, see the latest version of this topic.

- Host configuration is provided from:
  - Environment variables prefixed with `ASPNETCORE_` (for example, `ASPNETCORE_ENVIRONMENT`) using the Environment Variables Configuration Provider. The prefix (`ASPNETCORE_`) is stripped when the configuration key-value pairs are loaded.
  - Command-line arguments using the Command-line Configuration Provider.
- App configuration is provided from:
  - *appsettings.json* using the File Configuration Provider.
  - *appsettings.{Environment}.json* using the File Configuration Provider.
  - Secret Manager when the app runs in the `Development` environment using the entry assembly.
  - Environment variables using the Environment Variables Configuration Provider.
  - Command-line arguments using the Command-line Configuration Provider.

Security

Adopt the following practices to secure sensitive configuration data:

- Never store passwords or other sensitive data in configuration provider code or in plain text configuration files.
- Don't use production secrets in development or test environments.
- Specify secrets outside of the project so that they can't be accidentally committed to a source code repository.

For more information, see the following topics:

- Use multiple environments in ASP.NET Core
- Safe storage of app secrets in development in ASP.NET Core: Includes advice on using environment variables to store sensitive data. The Secret Manager uses the File Configuration Provider to store user secrets in a JSON file on the local system. The File Configuration Provider is described later in this topic.

Azure Key Vault safely stores app secrets for ASP.NET Core apps. For more information, see Azure Key Vault Configuration Provider in ASP.NET Core.

Hierarchical configuration data

The Configuration API is capable of maintaining hierarchical configuration data by flattening the hierarchical data with the use of a delimiter in the configuration keys.

In the following JSON file, four keys exist in a structured hierarchy of two sections:

```
{
  "section0": {
    "key0": "value",
    "key1": "value"
  },
  "section1": {
    "key0": "value",
    "key1": "value"
  }
}
```

When the file is read into configuration, unique keys are created to maintain the original hierarchical data structure of the configuration source. The sections and keys are flattened with the use of a colon (`:`) to maintain the original structure:

- section0:key0
- section0:key1
- section1:key0
- section1:key1

GetSection and GetChildren methods are available to isolate sections and children of a section in the configuration data. These methods are described later in GetSection, GetChildren, and Exists.

Conventions

Sources and providers

At app startup, configuration sources are read in the order that their configuration providers are specified.

Configuration providers that implement change detection have the ability to reload configuration when an underlying setting is changed. For example, the File Configuration Provider (described later in this topic) and the Azure Key Vault Configuration Provider implement change detection.

IConfiguration is available in the app's dependency injection (DI) container. IConfiguration can be injected into a Razor Pages PageModel or MVC Controller to obtain configuration for the class.

In the following examples, the `_config` field is used to access configuration values:

```
public class IndexModel : PageModel
{
    private readonly IConfiguration _config;

    public IndexModel(IConfiguration config)
    {
        _config = config;
    }
}
```

```
public class HomeController : Controller
{
    private readonly IConfiguration _config;

    public HomeController(IConfiguration config)
    {
        _config = config;
    }
}
```

Configuration providers can't utilize DI, as it's not available when they're set up by the host.

Keys

Configuration keys adopt the following conventions:

- Keys are case-insensitive. For example, `ConnectionString` and `connectionstring` are treated as equivalent keys.
- If a value for the same key is set by the same or different configuration providers, the last value set on the key is the value used. For more information on duplicate JSON keys, see this GitHub issue.
- Hierarchical keys
  - Within the Configuration API, a colon separator (`:`) works on all platforms.
  - In environment variables, a colon separator may not work on all platforms. A double underscore (`__`) is supported by all platforms and is automatically converted into a colon.
  - In Azure Key Vault, hierarchical keys use `--` (two dashes) as a separator. Write code to replace the dashes with a colon when the secrets are loaded into the app's configuration.
- The ConfigurationBinder supports binding arrays to objects using array indices in configuration keys. Array binding is described in the Bind an array to a class section.

Values

Configuration values adopt the following conventions:

- Values are strings.
- Null values can't be stored in configuration or bound to objects.

Providers

The following table shows the configuration providers available to ASP.NET Core apps.

PROVIDER	PROVIDES CONFIGURATION FROM...

PROVIDER	PROVIDES CONFIGURATION FROM...
Azure Key Vault Configuration Provider (*Security* topics)	Azure Key Vault
Azure App Configuration Provider (Azure documentation)	Azure App Configuration
Command-line Configuration Provider	Command-line parameters
Custom configuration provider	Custom source
Environment Variables Configuration Provider	Environment variables
File Configuration Provider	Files (INI, JSON, XML)
Key-per-file Configuration Provider	Directory files
Memory Configuration Provider	In-memory collections
User secrets (Secret Manager) (*Security* topics)	File in the user profile directory

Configuration sources are read in the order that their configuration providers are specified at startup. The configuration providers described in this topic are described in alphabetical order, not in the order that the code arranges them. Order configuration providers in code to suit the priorities for the underlying configuration sources that the app requires.

A typical sequence of configuration providers is:

1. Files (*appsettings.json*, *appsettings.{Environment}.json*, where `{Environment}` is the app's current hosting environment)
2. Azure Key Vault
3. User secrets (Secret Manager) (Development environment only)
4. Environment variables
5. Command-line arguments

A common practice is to position the Command-line Configuration Provider last in a series of providers to allow command-line arguments to override configuration set by the other providers.

The preceding sequence of providers is used when a new host builder is initialized with `CreateDefaultBuilder`. For more information, see the Default configuration section.

Configure the host builder with UseConfiguration

To configure the host builder, call UseConfiguration on the host builder with the configuration.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
    var dict = new Dictionary<string, string>
    {
        {"MemoryCollectionKey1", "value1"},
        {"MemoryCollectionKey2", "value2"}
    };

    var config = new ConfigurationBuilder()
        .AddInMemoryCollection(dict)
        .Build();

    return WebHost.CreateDefaultBuilder(args)
        .UseConfiguration(config)
        .UseStartup<Startup>();
}
```

ConfigureAppConfiguration

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration providers in addition to those added automatically by `CreateDefaultBuilder`:

```
public class Program
{
    public static Dictionary<string, string> arrayDict =
        new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddInMemoryCollection(arrayDict);
                config.AddJsonFile(
                    "json_array.json", optional: false, reloadOnChange: false);
                config.AddJsonFile(
                    "starship.json", optional: false, reloadOnChange: false);
                config.AddXmlFile(
                    "tvshow.xml", optional: false, reloadOnChange: false);
                config.AddEFConfiguration(
                    options => options.UseInMemoryDatabase("InMemoryDb"));
                config.AddCommandLine(args);
            })
            .UseStartup<Startup>();
}
```

Override previous configuration with command-line arguments

To provide app configuration that can be overridden with command-line arguments, call `AddCommandLine` last:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    // Call other providers here
    config.AddCommandLine(args);
})
```

Remove providers added by CreateDefaultBuilder

To remove the providers added by `CreateDefaultBuilder`, call Clear on the IConfigurationBuilder.Sources first:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.Sources.Clear();
    // Add providers here
})
```

Consume configuration during app startup

Configuration supplied to the app in `ConfigureAppConfiguration` is available during the app's startup, including `Startup.ConfigureServices`. For more information, see the Access configuration during startup section.

Command-line Configuration Provider

The CommandLineConfigurationProvider loads configuration from command-line argument key-value pairs at runtime.

To activate command-line configuration, the AddCommandLine extension method is called on an instance of ConfigurationBuilder.

`AddCommandLine` is automatically called when `CreateDefaultBuilder(string [])` is called. For more information, see the Default configuration section.

`CreateDefaultBuilder` also loads:

- Optional configuration from *appsettings.json* and *appsettings.{Environment}.json* files.
- User secrets (Secret Manager) in the Development environment.
- Environment variables.

`CreateDefaultBuilder` adds the Command-line Configuration Provider last. Command-line arguments passed at runtime override configuration set by the other providers.

`CreateDefaultBuilder` acts when the host is constructed. Therefore, command-line configuration activated by `CreateDefaultBuilder` can affect how the host is configured.

For apps based on the ASP.NET Core templates, `AddCommandLine` has already been called by `CreateDefaultBuilder`. To add additional configuration providers and maintain the ability to override configuration from those providers with command-line arguments, call the app's additional providers in `ConfigureAppConfiguration` and call `AddCommandLine` last.

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    // Call other providers here
    config.AddCommandLine(args);
})
```

Example

The sample app takes advantage of the static convenience method `CreateDefaultBuilder` to build the host, which includes a call to AddCommandLine.

1. Open a command prompt in the project's directory.
2. Supply a command-line argument to the `dotnet run` command, `dotnet run CommandLineKey=CommandLineValue`.
3. After the app is running, open a browser to the app at `http://localhost:5000`.
4. Observe that the output contains the key-value pair for the configuration command-line argument provided to `dotnet run`.

Arguments

The value must follow an equals sign (`=`), or the key must have a prefix (`--` or `/`) when the value follows a space. The value isn't required if an equals sign is used (for example, `CommandLineKey=`).

KEY PREFIX	EXAMPLE
No prefix	`CommandLineKey1=value1`
Two dashes (`--`)	`--CommandLineKey2=value2`, `--CommandLineKey2 value2`
Forward slash (`/`)	`/CommandLineKey3=value3`, `/CommandLineKey3 value3`

Within the same command, don't mix command-line argument key-value pairs that use an equals sign with key-value pairs that use a space.

Example commands:

```
dotnet run CommandLineKey1=value1 --CommandLineKey2=value2 /CommandLineKey3=value3
dotnet run --CommandLineKey1 value1 /CommandLineKey2 value2
dotnet run CommandLineKey1= CommandLineKey2=value2
```

Switch mappings

Switch mappings allow key name replacement logic. When manually building configuration with a ConfigurationBuilder, provide a dictionary of switch replacements to the AddCommandLine method.

When the switch mappings dictionary is used, the dictionary is checked for a key that matches the key provided by a command-line argument. If the command-line key is found in the dictionary, the dictionary value (the key replacement) is passed back to set the key-value pair into the app's configuration. A switch mapping is required for any command-line key prefixed with a single dash (`-`).

Switch mappings dictionary key rules:

- Switches must start with a dash (`-`) or double-dash (`--`).
- The switch mappings dictionary must not contain duplicate keys.

Create a switch mappings dictionary. In the following example, two switch mappings are created:

```
public static readonly Dictionary<string, string> _switchMappings =
    new Dictionary<string, string>
    {
        { "-CLKey1", "CommandLineKey1" },
        { "-CLKey2", "CommandLineKey2" }
    };
```

When the host is built, call `AddCommandLine` with the switch mappings dictionary:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.AddCommandLine(args, _switchMappings);
})
```

For apps that use switch mappings, the call to `CreateDefaultBuilder` shouldn't pass arguments. The `CreateDefaultBuilder` method's `AddCommandLine` call doesn't include mapped switches, and there's no way to pass the switch mapping dictionary to `CreateDefaultBuilder`. The solution isn't to pass the arguments to `CreateDefaultBuilder` but instead to allow the `ConfigurationBuilder` method's `AddCommandLine` method to process both the arguments and the switch mapping dictionary.

After the switch mappings dictionary is created, it contains the data shown in the following table.

KEY	VALUE
`-CLKey1`	`CommandLineKey1`
`-CLKey2`	`CommandLineKey2`

If the switch-mapped keys are used when starting the app, configuration receives the configuration value on the key supplied by the dictionary:

```
dotnet run -CLKey1=value1 -CLKey2=value2
```

After running the preceding command, configuration contains the values shown in the following table.

KEY	VALUE
`CommandLineKey1`	`value1`
`CommandLineKey2`	`value2`

Environment Variables Configuration Provider

The EnvironmentVariablesConfigurationProvider loads configuration from environment variable key-value pairs at runtime.

To activate environment variables configuration, call the AddEnvironmentVariables extension method on an instance of ConfigurationBuilder.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

Azure App Service permits setting environment variables in the Azure Portal that can override app configuration using the Environment Variables Configuration Provider. For more information, see Azure Apps: Override app configuration using the Azure Portal.

`AddEnvironmentVariables` is used to load environment variables prefixed with `ASPNETCORE_` for host configuration when a new host builder is initialized with the Web Host and `CreateDefaultBuilder` is called. For more information, see the Default configuration section.

`CreateDefaultBuilder` also loads:

- App configuration from unprefixed environment variables by calling `AddEnvironmentVariables` without a prefix.
- Optional configuration from *appsettings.json* and *appsettings.{Environment}.json* files.
- User secrets (Secret Manager) in the Development environment.
- Command-line arguments.

The Environment Variables Configuration Provider is called after configuration is established from user secrets and *appsettings* files. Calling the provider in this position allows the environment variables read at runtime to override configuration set by user secrets and *appsettings* files.

To provide app configuration from additional environment variables, call the app's additional providers in `ConfigureAppConfiguration` and call `AddEnvironmentVariables` with the prefix:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.AddEnvironmentVariables(prefix: "PREFIX_");
})
```

Call `AddEnvironmentVariables` last to allow environment variables with the given prefix to override values from other providers.

Example

The sample app takes advantage of the static convenience method `CreateDefaultBuilder` to build the host, which includes a call to `AddEnvironmentVariables`.

1. Run the sample app. Open a browser to the app at `http://localhost:5000`.
2. Observe that the output contains the key-value pair for the environment variable `ENVIRONMENT`. The value reflects the environment in which the app is running, typically `Development` when running locally.

To keep the list of environment variables rendered by the app short, the app filters environment variables. See the sample app's *Pages/Index.cshtml.cs* file.

To expose all of the environment variables available to the app, change the `FilteredConfiguration` in *Pages/Index.cshtml.cs* to the following:

```
FilteredConfiguration = _config.AsEnumerable();
```

Prefixes

Environment variables loaded into the app's configuration are filtered when supplying a prefix to the `AddEnvironmentVariables` method. For example, to filter environment variables on the prefix `CUSTOM_`, supply the prefix to the configuration provider:

```
var config = new ConfigurationBuilder()
    .AddEnvironmentVariables("CUSTOM_")
    .Build();
```

The prefix is stripped off when the configuration key-value pairs are created.

When the host builder is created, host configuration is provided by environment variables. For more information on the prefix used for these environment variables, see the Default configuration section.

Connection string prefixes

The Configuration API has special processing rules for four connection string environment variables involved in configuring Azure connection strings for the app environment. Environment variables with the prefixes shown in the table are loaded into the app if no prefix is supplied to `AddEnvironmentVariables`.

CONNECTION STRING PREFIX	PROVIDER
`CUSTOMCONNSTR_`	Custom provider
`MYSQLCONNSTR_`	MySQL
`SQLAZURECONNSTR_`	Azure SQL Database
`SQLCONNSTR_`	SQL Server

When an environment variable is discovered and loaded into configuration with any of the four prefixes shown in the table:

- The configuration key is created by removing the environment variable prefix and adding a configuration key section (`ConnectionStrings`).
- A new configuration key-value pair is created that represents the database connection provider (except for `CUSTOMCONNSTR_`, which has no stated provider).

ENVIRONMENT VARIABLE KEY	CONVERTED CONFIGURATION KEY	PROVIDER CONFIGURATION ENTRY
`CUSTOMCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Configuration entry not created.
`MYSQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` : Value: `MySql.Data.MySqlClient`

ENVIRONMENT VARIABLE KEY	CONVERTED CONFIGURATION KEY	PROVIDER CONFIGURATION ENTRY
`SQLAZURECONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` : Value: `System.Data.SqlClient`
`SQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` : Value: `System.Data.SqlClient`

Example

A custom connection string environment variable is created on the server:

- Name: `CUSTOMCONNSTR_ReleaseDB`
- Value: `Data Source=ReleaseSQLServer;Initial Catalog=MyReleaseDB;Integrated Security=True`

If `IConfiguration` is injected and assigned to a field named `_config`, read the value:

```
_config["ConnectionStrings:ReleaseDB"]
```

File Configuration Provider

FileConfigurationProvider is the base class for loading configuration from the file system. The following configuration providers are dedicated to specific file types:

- INI Configuration Provider
- JSON Configuration Provider
- XML Configuration Provider

INI Configuration Provider

The IniConfigurationProvider loads configuration from INI file key-value pairs at runtime.

To activate INI file configuration, call the AddIniFile extension method on an instance of ConfigurationBuilder.

The colon can be used to as a section delimiter in INI file configuration.

Overloads permit specifying:

- Whether the file is optional.
- Whether the configuration is reloaded if the file changes.
- The IFileProvider used to access the file.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.AddIniFile(
        "config.ini", optional: true, reloadOnChange: true);
})
```

A generic example of an INI configuration file:

```
[section0]
key0=value
key1=value

[section1]
subsection:key=value

[section2:subsection0]
key=value

[section2:subsection1]
key=value
```

The previous configuration file loads the following keys with `value`:

- section0:key0
- section0:key1
- section1:subsection:key
- section2:subsection0:key
- section2:subsection1:key

JSON Configuration Provider

The JsonConfigurationProvider loads configuration from JSON file key-value pairs during runtime.

To activate JSON file configuration, call the AddJsonFile extension method on an instance of ConfigurationBuilder.

Overloads permit specifying:

- Whether the file is optional.
- Whether the configuration is reloaded if the file changes.
- The IFileProvider used to access the file.

`AddJsonFile` is automatically called twice when a new host builder is initialized with `CreateDefaultBuilder`. The method is called to load configuration from:

- *appsettings.json*: This file is read first. The environment version of the file can override the values provided by the *appsettings.json* file.
- *appsettings.{Environment}.json*: The environment version of the file is loaded based on the IHostingEnvironment.EnvironmentName.

For more information, see the Default configuration section.

`CreateDefaultBuilder` also loads:

- Environment variables.
- User secrets (Secret Manager) in the Development environment.
- Command-line arguments.

The JSON Configuration Provider is established first. Therefore, user secrets, environment variables, and command-line arguments override configuration set by the *appsettings* files.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration for files other than *appsettings.json* and *appsettings.{Environment}.json*:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.AddJsonFile(
        "config.json", optional: true, reloadOnChange: true);
})
```

Example

The sample app takes advantage of the static convenience method `CreateDefaultBuilder` to build the host, which includes two calls to `AddJsonFile`:

- The first call to `AddJsonFile` loads configuration from *appsettings.json*:

  ```
  {
    "Logging": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "AllowedHosts": "*"
  }
  ```

- The second call to `AddJsonFile` loads configuration from *appsettings.{Environment}.json*. For *appsettings.Development.json* in the sample app, the following file is loaded:

  ```
  {
    "Logging": {
      "LogLevel": {
        "Default": "Debug",
        "System": "Information",
        "Microsoft": "Information"
      }
    }
  }
  ```

1. Run the sample app. Open a browser to the app at `http://localhost:5000`.
2. The output contains key-value pairs for the configuration based on the app's environment. The log level for the key `Logging:LogLevel:Default` is `Debug` when running the app in the Development environment.
3. Run the sample app again in the Production environment:
   a. Open the *Properties/launchSettings.json* file.
   b. In the `ConfigurationSample` profile, change the value of the `ASPNETCORE_ENVIRONMENT` environment variable to `Production`.
   c. Save the file and run the app with `dotnet run` in a command shell.
4. The settings in the *appsettings.Development.json* no longer override the settings in *appsettings.json*. The log level for the key `Logging:LogLevel:Default` is `Warning`.

XML Configuration Provider

The XmlConfigurationProvider loads configuration from XML file key-value pairs at runtime.

To activate XML file configuration, call the AddXmlFile extension method on an instance of ConfigurationBuilder.

Overloads permit specifying:

- Whether the file is optional.

- Whether the configuration is reloaded if the file changes.
- The IFileProvider used to access the file.

The root node of the configuration file is ignored when the configuration key-value pairs are created. Don't specify a Document Type Definition (DTD) or namespace in the file.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.AddXmlFile(
        "config.xml", optional: true, reloadOnChange: true);
})
```

XML configuration files can use distinct element names for repeating sections:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section0>
    <key0>value</key0>
    <key1>value</key1>
  </section0>
  <section1>
    <key0>value</key0>
    <key1>value</key1>
  </section1>
</configuration>
```

The previous configuration file loads the following keys with `value`:

- section0:key0
- section0:key1
- section1:key0
- section1:key1

Repeating elements that use the same element name work if the `name` attribute is used to distinguish the elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="section0">
    <key name="key0">value</key>
    <key name="key1">value</key>
  </section>
  <section name="section1">
    <key name="key0">value</key>
    <key name="key1">value</key>
  </section>
</configuration>
```

The previous configuration file loads the following keys with `value`:

- section:section0:key:key0
- section:section0:key:key1
- section:section1:key:key0
- section:section1:key:key1

Attributes can be used to supply values:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
  </section>
</configuration>
```

The previous configuration file loads the following keys with `value`:

- key:attribute
- section:key:attribute

Key-per-file Configuration Provider

The KeyPerFileConfigurationProvider uses a directory's files as configuration key-value pairs. The key is the file name. The value contains the file's contents. The Key-per-file Configuration Provider is used in Docker hosting scenarios.

To activate key-per-file configuration, call the AddKeyPerFile extension method on an instance of ConfigurationBuilder. The `directoryPath` to the files must be an absolute path.

Overloads permit specifying:

- An `Action<KeyPerFileConfigurationSource>` delegate that configures the source.
- Whether the directory is optional and the path to the directory.

The double-underscore (`__`) is used as a configuration key delimiter in file names. For example, the file name `Logging__LogLevel__System` produces the configuration key `Logging:LogLevel:System`.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    var path = Path.Combine(
        Directory.GetCurrentDirectory(), "path/to/files");
    config.AddKeyPerFile(directoryPath: path, optional: true);
})
```

Memory Configuration Provider

The MemoryConfigurationProvider uses an in-memory collection as configuration key-value pairs.

To activate in-memory collection configuration, call the AddInMemoryCollection extension method on an instance of ConfigurationBuilder.

The configuration provider can be initialized with an `IEnumerable<KeyValuePair<String,String>>`.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration.

In the following example, a configuration dictionary is created:

```
public static readonly Dictionary<string, string> _dict =
    new Dictionary<string, string>
    {
        {"MemoryCollectionKey1", "value1"},
        {"MemoryCollectionKey2", "value2"}
    };
```

The dictionary is used with a call to `AddInMemoryCollection` to provide the configuration:

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    config.AddInMemoryCollection(_dict);
})
```

GetValue

`ConfigurationBinder.GetValue<T>` extracts a single value from configuration with a specified key and converts it to the specified noncollection type. An overload accepts a default value.

The following example:

- Extracts the string value from configuration with the key `NumberKey`. If `NumberKey` isn't found in the configuration keys, the default value of `99` is used.
- Types the value as an `int`.
- Stores the value in the `NumberConfig` property for use by the page.

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        _config = config;
    }

    public int NumberConfig { get; private set; }

    public void OnGet()
    {
        NumberConfig = _config.GetValue<int>("NumberKey", 99);
    }
}
```

GetSection, GetChildren, and Exists

For the examples that follow, consider the following JSON file. Four keys are found across two sections, one of which includes a pair of subsections:

```
{
  "section0": {
    "key0": "value",
    "key1": "value"
  },
  "section1": {
    "key0": "value",
    "key1": "value"
  },
  "section2": {
    "subsection0" : {
      "key0": "value",
      "key1": "value"
    },
    "subsection1" : {
      "key0": "value",
      "key1": "value"
    }
  }
}
```

When the file is read into configuration, the following unique hierarchical keys are created to hold the configuration values:

- section0:key0
- section0:key1
- section1:key0
- section1:key1
- section2:subsection0:key0
- section2:subsection0:key1
- section2:subsection1:key0
- section2:subsection1:key1

GetSection

IConfiguration.GetSection extracts a configuration subsection with the specified subsection key.

To return an IConfigurationSection containing only the key-value pairs in `section1`, call `GetSection` and supply the section name:

```
var configSection = _config.GetSection("section1");
```

The `configSection` doesn't have a value, only a key and a path.

Similarly, to obtain the values for keys in `section2:subsection0`, call `GetSection` and supply the section path:

```
var configSection = _config.GetSection("section2:subsection0");
```

`GetSection` never returns `null`. If a matching section isn't found, an empty `IConfigurationSection` is returned.

When `GetSection` returns a matching section, Value isn't populated. A Key and Path are returned when the section exists.

GetChildren

A call to IConfiguration.GetChildren on `section2` obtains an

`IEnumerable<IConfigurationSection>` that includes:

- `subsection0`
- `subsection1`

```
var configSection = _config.GetSection("section2");

var children = configSection.GetChildren();
```

Exists

Use ConfigurationExtensions.Exists to determine if a configuration section exists:

```
var sectionExists = _config.GetSection("section2:subsection2").Exists();
```

Given the example data, `sectionExists` is `false` because there isn't a `section2:subsection2` section in the configuration data.

Bind to an object graph

Bind is capable of binding an entire POCO object graph. As with binding a simple object, only public read/write properties are bound.

The sample contains a `TvShow` model whose object graph includes `Metadata` and `Actors` classes (*Models/TvShow.cs*):

```
public class TvShow
{
    public Metadata Metadata { get; set; }
    public Actors Actors { get; set; }
    public string Legal { get; set; }
}

public class Metadata
{
    public string Series { get; set; }
    public string Title { get; set; }
    public DateTime AirDate { get; set; }
    public int Episodes { get; set; }
}

public class Actors
{
    public string Names { get; set; }
}
```

The sample app has a *tvshow.xml* file containing the configuration data:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <tvshow>
    <metadata>
      <series>Dr. Who</series>
      <title>The Sun Makers</title>
      <airdate>11/26/1977</airdate>
      <episodes>4</episodes>
    </metadata>
    <actors>
      <names>Tom Baker, Louise Jameson, John Leeson</names>
    </actors>
    <legal>(c)1977 BBC https://www.bbc.co.uk/programmes/b006q2x0</legal>
  </tvshow>
</configuration>
```

Configuration is bound to the entire `TvShow` object graph with the `Bind` method. The bound instance is assigned to a property for rendering:

```
var tvShow = new TvShow();
_config.GetSection("tvshow").Bind(tvShow);
TvShow = tvShow;
```

`ConfigurationBinder.Get<T>` binds and returns the specified type. `Get<T>` is more convenient than using `Bind`. The following code shows how to use `Get<T>` with the preceding example:

```
TvShow = _config.GetSection("tvshow").Get<TvShow>();
```

Bind an array to a class

The sample app demonstrates the concepts explained in this section.

The Bind supports binding arrays to objects using array indices in configuration keys. Any array format that exposes a numeric key segment (`:0:` , `:1:` , ... `:{n}:`) is capable of array binding to a POCO class array.

> **NOTE**
> Binding is provided by convention. Custom configuration providers aren't required to implement array binding.

In-memory array processing

Consider the configuration keys and values shown in the following table.

KEY	VALUE
array:entries:0	value0
array:entries:1	value1
array:entries:2	value2
array:entries:4	value4

KEY	VALUE
array:entries:5	value5

These keys and values are loaded in the sample app using the Memory Configuration Provider:

```
public class Program
{
    public static Dictionary<string, string> arrayDict =
        new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddInMemoryCollection(arrayDict);
                config.AddJsonFile(
                    "json_array.json", optional: false, reloadOnChange: false);
                config.AddJsonFile(
                    "starship.json", optional: false, reloadOnChange: false);
                config.AddXmlFile(
                    "tvshow.xml", optional: false, reloadOnChange: false);
                config.AddEFConfiguration(
                    options => options.UseInMemoryDatabase("InMemoryDb"));
                config.AddCommandLine(args);
            })
            .UseStartup<Startup>();
}
```

The array skips a value for index #3. The configuration binder isn't capable of binding null values or creating null entries in bound objects, which becomes clear in a moment when the result of binding this array to an object is demonstrated.

In the sample app, a POCO class is available to hold the bound configuration data:

```
public class ArrayExample
{
    public string[] Entries { get; set; }
}
```

The configuration data is bound to the object:

```
var arrayExample = new ArrayExample();
_config.GetSection("array").Bind(arrayExample);
```

`ConfigurationBinder.Get<T>` syntax can also be used, which results in more compact code:

```
ArrayExample = _config.GetSection("array").Get<ArrayExample>();
```

The bound object, an instance of `ArrayExample`, receives the array data from configuration.

`ARRAYEXAMPLE.ENTRIES` INDEX	`ARRAYEXAMPLE.ENTRIES` VALUE
0	value0
1	value1
2	value2
3	value4
4	value5

Index #3 in the bound object holds the configuration data for the `array:4` configuration key and its value of `value4`. When configuration data containing an array is bound, the array indices in the configuration keys are merely used to iterate the configuration data when creating the object. A null value can't be retained in configuration data, and a null-valued entry isn't created in a bound object when an array in configuration keys skip one or more indices.

The missing configuration item for index #3 can be supplied before binding to the `ArrayExample` instance by any configuration provider that produces the correct key-value pair in configuration. If the sample included an additional JSON Configuration Provider with the missing key-value pair, the `ArrayExample.Entries` matches the complete configuration array:

missing_value.json:

```
{
  "array:entries:3": "value3"
}
```

In `ConfigureAppConfiguration`:

```
config.AddJsonFile(
    "missing_value.json", optional: false, reloadOnChange: false);
```

The key-value pair shown in the table is loaded into configuration.

KEY	VALUE
array:entries:3	value3

If the `ArrayExample` class instance is bound after the JSON Configuration Provider includes the entry for index #3, the `ArrayExample.Entries` array includes the value.

`ARRAYEXAMPLE.ENTRIES` INDEX	`ARRAYEXAMPLE.ENTRIES` VALUE
0	value0
1	value1

ARRAYEXAMPLE.ENTRIES INDEX	ARRAYEXAMPLE.ENTRIES VALUE
2	value2
3	value3
4	value4
5	value5

JSON array processing

If a JSON file contains an array, configuration keys are created for the array elements with a zero-based section index. In the following configuration file, `subsection` is an array:

```
{
  "json_array": {
    "key": "valueA",
    "subsection": [
      "valueB",
      "valueC",
      "valueD"
    ]
  }
}
```

The JSON Configuration Provider reads the configuration data into the following key-value pairs:

KEY	VALUE
json_array:key	valueA
json_array:subsection:0	valueB
json_array:subsection:1	valueC
json_array:subsection:2	valueD

In the sample app, the following POCO class is available to bind the configuration key-value pairs:

```
public class JsonArrayExample
{
    public string Key { get; set; }
    public string[] Subsection { get; set; }
}
```

After binding, `JsonArrayExample.Key` holds the value `valueA`. The subsection values are stored in the POCO array property, `Subsection`.

JSONARRAYEXAMPLE.SUBSECTION INDEX	JSONARRAYEXAMPLE.SUBSECTION VALUE
0	valueB

JSONARRAYEXAMPLE.SUBSECTION INDEX	JSONARRAYEXAMPLE.SUBSECTION VALUE
1	valueC
2	valueD

Custom configuration provider

The sample app demonstrates how to create a basic configuration provider that reads configuration key-value pairs from a database using Entity Framework (EF).

The provider has the following characteristics:

- The EF in-memory database is used for demonstration purposes. To use a database that requires a connection string, implement a secondary `ConfigurationBuilder` to supply the connection string from another configuration provider.
- The provider reads a database table into configuration at startup. The provider doesn't query the database on a per-key basis.
- Reload-on-change isn't implemented, so updating the database after the app starts has no effect on the app's configuration.

Define an `EFConfigurationValue` entity for storing configuration values in the database.

Models/EFConfigurationValue.cs:

```
public class EFConfigurationValue
{
    public string Id { get; set; }
    public string Value { get; set; }
}
```

Add an `EFConfigurationContext` to store and access the configured values.

EFConfigurationProvider/EFConfigurationContext.cs:

```
public class EFConfigurationContext : DbContext
{
    public EFConfigurationContext(DbContextOptions options) : base(options)
    {
    }

    public DbSet<EFConfigurationValue> Values { get; set; }
}
```

Create a class that implements IConfigurationSource.

EFConfigurationProvider/EFConfigurationSource.cs:

```
public class EFConfigurationSource : IConfigurationSource
{
    private readonly Action<DbContextOptionsBuilder> _optionsAction;

    public EFConfigurationSource(Action<DbContextOptionsBuilder> optionsAction)
    {
        _optionsAction = optionsAction;
    }

    public IConfigurationProvider Build(IConfigurationBuilder builder)
    {
        return new EFConfigurationProvider(_optionsAction);
    }
}
```

Create the custom configuration provider by inheriting from ConfigurationProvider. The configuration provider initializes the database when it's empty.

EFConfigurationProvider/EFConfigurationProvider.cs:

```
public class EFConfigurationProvider : ConfigurationProvider
{
    public EFConfigurationProvider(Action<DbContextOptionsBuilder> optionsAction)
    {
        OptionsAction = optionsAction;
    }

    Action<DbContextOptionsBuilder> OptionsAction { get; }

    // Load config data from EF DB.
    public override void Load()
    {
        var builder = new DbContextOptionsBuilder<EFConfigurationContext>();

        OptionsAction(builder);

        using (var dbContext = new EFConfigurationContext(builder.Options))
        {
            dbContext.Database.EnsureCreated();

            Data = !dbContext.Values.Any()
                ? CreateAndSaveDefaultValues(dbContext)
                : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
        }
    }

    private static IDictionary<string, string> CreateAndSaveDefaultValues(
        EFConfigurationContext dbContext)
    {
        // Quotes (c)2005 Universal Pictures: Serenity
        // https://www.uphe.com/movies/serenity
        var configValues =
            new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase)
            {
                { "quote1", "I aim to misbehave." },
                { "quote2", "I swallowed a bug." },
                { "quote3", "You can't stop the signal, Mal." }
            };

        dbContext.Values.AddRange(configValues
            .Select(kvp => new EFConfigurationValue
                {
                    Id = kvp.Key,
                    Value = kvp.Value
                })
            .ToArray());

        dbContext.SaveChanges();

        return configValues;
    }
}
```

An `AddEFConfiguration` extension method permits adding the configuration source to a `ConfigurationBuilder`.

Extensions/EntityFrameworkExtensions.cs:

```
public static class EntityFrameworkExtensions
{
    public static IConfigurationBuilder AddEFConfiguration(
        this IConfigurationBuilder builder,
        Action<DbContextOptionsBuilder> optionsAction)
    {
        return builder.Add(new EFConfigurationSource(optionsAction));
    }
}
```

The following code shows how to use the custom `EFConfigurationProvider` in *Program.cs*:

```
public class Program
{
    public static Dictionary<string, string> arrayDict =
        new Dictionary<string, string>
        {
            {"array:entries:0", "value0"},
            {"array:entries:1", "value1"},
            {"array:entries:2", "value2"},
            {"array:entries:4", "value4"},
            {"array:entries:5", "value5"}
        };

    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddInMemoryCollection(arrayDict);
                config.AddJsonFile(
                    "json_array.json", optional: false, reloadOnChange: false);
                config.AddJsonFile(
                    "starship.json", optional: false, reloadOnChange: false);
                config.AddXmlFile(
                    "tvshow.xml", optional: false, reloadOnChange: false);
                config.AddEFConfiguration(
                    options => options.UseInMemoryDatabase("InMemoryDb"));
                config.AddCommandLine(args);
            })
            .UseStartup<Startup>();
}
```

Access configuration during startup

Inject `IConfiguration` into the `Startup` constructor to access configuration values in `Startup.ConfigureServices`. To access configuration in `Startup.Configure`, either inject `IConfiguration` directly into the method or use the instance from the constructor:

```
public class Startup
{
    private readonly IConfiguration _config;

    public Startup(IConfiguration config)
    {
        _config = config;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        var value = _config["key"];
    }

    public void Configure(IApplicationBuilder app, IConfiguration config)
    {
        var value = config["key"];
    }
}
```

For an example of accessing configuration using startup convenience methods, see App startup: Convenience methods.

Access configuration in a Razor Pages page or MVC view

To access configuration settings in a Razor Pages page or an MVC view, add a using directive (C# reference: using directive) for the Microsoft.Extensions.Configuration namespace and inject IConfiguration into the page or view.

In a Razor Pages page:

```
@page
@model IndexModel
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index Page</title>
</head>
<body>
    <h1>Access configuration in a Razor Pages page</h1>
    <p>Configuration value for 'key': @Configuration["key"]</p>
</body>
</html>
```

In an MVC view:

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Index View</title>
</head>
<body>
    <h1>Access configuration in an MVC view</h1>
    <p>Configuration value for 'key': @Configuration["key"]</p>
</body>
</html>
```

Add configuration from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Startup` class. For more information, see Use hosting startup assemblies in ASP.NET Core.

Additional resources

- Options pattern in ASP.NET Core

Options pattern in ASP.NET Core

9/22/2020 • 34 minutes to read • Edit Online

By Kirk Larkin and Rick Anderson.

The options pattern uses classes to provide strongly typed access to groups of related settings. When configuration settings are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- The Interface Segregation Principle (ISP) or Encapsulation: Scenarios (classes) that depend on configuration settings depend only on the configuration settings that they use.
- Separation of Concerns: Settings for different parts of the app aren't dependent or coupled to one another.

Options also provide a mechanism to validate configuration data. For more information, see the Options validation section.

View or download sample code (how to download)

Bind hierarchical configuration

The preferred way to read related configuration values is using the options pattern. For example, to read the following configuration values:

```
"Position": {
  "Title": "Editor",
  "Name": "Joe Smith"
}
```

Create the following `PositionOptions` class:

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; }
    public string Name { get; set; }
}
```

An options class:

- Must be non-abstract with a public parameterless constructor.
- All public read-write properties of the type are bound.
- Fields are ***not*** bound. In the preceding code, `Position` is not bound. The `Position` property is used so the string `"Position"` doesn't need to be hard coded in the app when binding the class to a configuration provider.

The following code:

- Calls ConfigurationBinder.Bind to bind the `PositionOptions` class to the `Position` section.
- Displays the `Position` configuration data.

```
public class Test22Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test22Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var positionOptions = new PositionOptions();
        Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

`ConfigurationBinder.Get<T>` binds and returns the specified type. `ConfigurationBinder.Get<T>` may be more convenient than using `ConfigurationBinder.Bind`. The following code shows how to use `ConfigurationBinder.Get<T>` with the `PositionOptions` class:

```
public class Test21Model : PageModel
{
    private readonly IConfiguration Configuration;
    public PositionOptions positionOptions { get; private set; }

    public Test21Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        positionOptions = Configuration.GetSection(PositionOptions.Position)
                                                    .Get<PositionOptions>();

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

An alternative approach when using the ***options pattern*** is to bind the `Position` section and add it to the dependency injection service container. In the following code, `PositionOptions` is added to the service container with Configure and bound to configuration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<PositionOptions>(Configuration.GetSection(
                                        PositionOptions.Position));
    services.AddRazorPages();
}
```

Using the preceding code, the following code reads the position options:

```
public class Test2Model : PageModel
{
    private readonly PositionOptions _options;

    public Test2Model(IOptions<PositionOptions> options)
    {
        _options = options.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Title: {_options.Title} \n" +
                       $"Name: {_options.Name}");
    }
}
```

In the preceding code, changes to the JSON configuration file after the app has started are ***not*** read. To read changes after the app has started, use IOptionsSnapshot.

Options interfaces

IOptions<TOptions>:

- Does ***not*** support:
  - Reading of configuration data after the app has started.
  - Named options
- Is registered as a Singleton and can be injected into any service lifetime.

IOptionsSnapshot<TOptions>:

- Is useful in scenarios where options should be recomputed on every request. For more information, see Use IOptionsSnapshot to read updated data.
- Is registered as Scoped and therefore cannot be injected into a Singleton service.
- Supports named options

IOptionsMonitor<TOptions>:

- Is used to retrieve options and manage options notifications for `TOptions` instances.
- Is registered as a Singleton and can be injected into any service lifetime.
- Supports:
  - Change notifications
  - Named options
  - Reloadable configuration
  - Selective options invalidation (IOptionsMonitorCache<TOptions>)

Post-configuration scenarios enable setting or changing options after all IConfigureOptions<TOptions> configuration occurs.

IOptionsFactory<TOptions> is responsible for creating new options instances. It has a single Create method. The default implementation takes all registered IConfigureOptions<TOptions> and IPostConfigureOptions<TOptions> and runs all the configurations first, followed by the post-configuration. It distinguishes between IConfigureNamedOptions<TOptions> and IConfigureOptions<TOptions> and only calls the appropriate interface.

IOptionsMonitorCache<TOptions> is used by IOptionsMonitor<TOptions> to cache `TOptions` instances. The IOptionsMonitorCache<TOptions> invalidates options instances in the monitor so that the value is recomputed

(TryRemove). Values can be manually introduced with TryAdd. The Clear method is used when all named instances should be recreated on demand.

Use IOptionsSnapshot to read updated data

Using IOptionsSnapshot<TOptions>, options are computed once per request when accessed and cached for the lifetime of the request. Changes to the configuration are read after the app starts when using configuration providers that support reading updated configuration values.

The difference between `IOptionsMonitor` and `IOptionsSnapshot` is that:

- `IOptionsMonitor` is a singleton service that retrieves current option values at any time, which is especially useful in singleton dependencies.
- `IOptionsSnapshot` is a scoped service and provides a snapshot of the options at the time the `IOptionsSnapshot<T>` object is constructed. Options snapshots are designed for use with transient and scoped dependencies.

The following code uses IOptionsSnapshot<TOptions>.

```
public class TestSnapModel : PageModel
{
    private readonly MyOptions _snapshotOptions;

    public TestSnapModel(IOptionsSnapshot<MyOptions> snapshotOptionsAccessor)
    {
        _snapshotOptions = snapshotOptionsAccessor.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_snapshotOptions.Option1} \n" +
                       $"Option2: {_snapshotOptions.Option2}");
    }
}
```

The following code registers a configuration instance which `MyOptions` binds against:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MyOptions>(Configuration.GetSection("MyOptions"));

    services.AddRazorPages();
}
```

In the preceding code, changes to the JSON configuration file after the app has started are read.

IOptionsMonitor

The following code registers a configuration instance which `MyOptions` binds against.

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MyOptions>(Configuration.GetSection("MyOptions"));

    services.AddRazorPages();
}
```

The following example uses IOptionsMonitor<TOptions>:

```
public class TestMonitorModel : PageModel
{
    private readonly IOptionsMonitor<MyOptions> _optionsDelegate;

    public TestMonitorModel(IOptionsMonitor<MyOptions> optionsDelegate )
    {
        _optionsDelegate = optionsDelegate;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_optionsDelegate.CurrentValue.Option1} \n" +
                       $"Option2: {_optionsDelegate.CurrentValue.Option2}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

Named options support using IConfigureNamedOptions

Named options:

- Are useful when multiple configuration sections bind to the same properties.
- Are case sensitive.

Consider the following *appsettings.json* file:

```
{
  "TopItem": {
    "Month": {
      "Name": "Green Widget",
      "Model": "GW46"
    },
    "Year": {
      "Name": "Orange Gadget",
      "Model": "OG35"
    }
  }
}
```

Rather than creating two classes to bind `TopItem:Month` and `TopItem:Year`, the following class is used for each section:

```
public class TopItemSettings
{
    public const string Month = "Month";
    public const string Year = "Year";

    public string Name { get; set; }
    public string Model { get; set; }
}
```

The following code configures the named options:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<TopItemSettings>(TopItemSettings.Month,
                                        Configuration.GetSection("TopItem:Month"));
    services.Configure<TopItemSettings>(TopItemSettings.Year,
                                         Configuration.GetSection("TopItem:Year"));

    services.AddRazorPages();
}
```

The following code displays the named options:

```
public class TestNOModel : PageModel
{
    private readonly TopItemSettings _monthTopItem;
    private readonly TopItemSettings _yearTopItem;

    public TestNOModel(IOptionsSnapshot<TopItemSettings> namedOptionsAccessor)
    {
        _monthTopItem = namedOptionsAccessor.Get(TopItemSettings.Month);
        _yearTopItem = namedOptionsAccessor.Get(TopItemSettings.Year);
    }

    public ContentResult OnGet()
    {
        return Content($"Month:Name {_monthTopItem.Name} \n" +
                       $"Month:Model {_monthTopItem.Model} \n\n" +
                       $"Year:Name {_yearTopItem.Name} \n" +
                       $"Year:Model {_yearTopItem.Model} \n"   );
    }
}
```

All options are named instances. IConfigureOptions<TOptions> instances are treated as targeting the `Options.DefaultName` instance, which is `string.Empty`. IConfigureNamedOptions<TOptions> also implements IConfigureOptions<TOptions>. The default implementation of the IOptionsFactory<TOptions> has logic to use each appropriately. The `null` named option is used to target all of the named instances instead of a specific named instance. ConfigureAll and PostConfigureAll use this convention.

OptionsBuilder API

OptionsBuilder<TOptions> is used to configure `TOptions` instances. `OptionsBuilder` streamlines creating named options as it's only a single parameter to the initial `AddOptions<TOptions>(string optionsName)` call instead of appearing in all of the subsequent calls. Options validation and the `ConfigureOptions` overloads that accept service dependencies are only available via `OptionsBuilder`.

`OptionsBuilder` is used in the Options validation section.

Use DI services to configure options

Services can be accessed from dependency injection while configuring options in two ways:

- Pass a configuration delegate to Configure on OptionsBuilder<TOptions>. `OptionsBuilder<TOptions>` provides overloads of Configure that allow use of up to five services to configure options:

  ```
  services.AddOptions<MyOptions>("optionalName")
      .Configure<Service1, Service2, Service3, Service4, Service5>(
          (o, s, s2, s3, s4, s5) =>
              o.Property = DoSomethingWith(s, s2, s3, s4, s5));
  ```

- Create a type that implements IConfigureOptions<TOptions> or IConfigureNamedOptions<TOptions> and register the type as a service.

We recommend passing a configuration delegate to Configure, since creating a service is more complex. Creating a type is equivalent to what the framework does when calling Configure. Calling Configure registers a transient generic IConfigureNamedOptions<TOptions>, which has a constructor that accepts the generic service types specified.

Options validation

Options validation enables option values to be validated.

Consider the following *appsettings.json* file:

```
{
  "MyConfig": {
    "Key1": "My Key One",
    "Key2": 10,
    "Key3": 32
  }
}
```

The following class binds to the `"MyConfig"` configuration section and applies a couple of `DataAnnotations` rules:

```
public class MyConfigOptions
{
    public const string MyConfig = "MyConfig";

    [RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$")]
    public string Key1 { get; set; }
    [Range(0, 1000,
        ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public int Key2 { get; set; }
    public int Key3 { get; set; }
}
```

The following code:

- Calls AddOptions to get an OptionsBuilder<TOptions> that binds to the `MyConfigOptions` class.
- Calls ValidateDataAnnotations to enable validation using `DataAnnotations`.

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddOptions<MyConfigOptions>()
            .Bind(Configuration.GetSection(MyConfigOptions.MyConfig))
            .ValidateDataAnnotations();

        services.AddControllersWithViews();
    }
```

The `ValidateDataAnnotations` extension method is defined in the Microsoft.Extensions.Options.DataAnnotations NuGet package. For web apps that use the `Microsoft.NET.Sdk.Web` SDK, this package is referenced implicitly from the shared framework.

The following code displays the configuration values or the validation errors:

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IOptions<MyConfigOptions> _config;

    public HomeController(IOptions<MyConfigOptions> config,
                          ILogger<HomeController> logger)
    {
        _config = config;
        _logger = logger;

        try
        {
            var configValue = _config.Value;

        }
        catch (OptionsValidationException ex)
        {
            foreach (var failure in ex.Failures)
            {
                _logger.LogError(failure);
            }
        }
    }

    public ContentResult Index()
    {
        string msg;
        try
        {
             msg = $"Key1: {_config.Value.Key1} \n" +
                   $"Key2: {_config.Value.Key2} \n" +
                   $"Key3: {_config.Value.Key3}";
        }
        catch (OptionsValidationException optValEx)
        {
            return Content(optValEx.Message);
        }
        return Content(msg);
    }
```

The following code applies a more complex validation rule using a delegate:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddOptions<MyConfigOptions>()
        .Bind(Configuration.GetSection(MyConfigOptions.MyConfig))
        .ValidateDataAnnotations()
        .Validate(config =>
        {
            if (config.Key2 != 0)
            {
                return config.Key3 > config.Key2;
            }

            return true;
        }, "Key3 must be > than Key2.");   // Failure message.

    services.AddControllersWithViews();
}
```

IValidateOptions for complex validation

The following class implements IValidateOptions<TOptions>:

```
public class MyConfigValidation : IValidateOptions<MyConfigOptions>
{
    public MyConfigOptions _config { get; private set; }

    public  MyConfigValidation(IConfiguration config)
    {
        _config = config.GetSection(MyConfigOptions.MyConfig)
            .Get<MyConfigOptions>();
    }

    public ValidateOptionsResult Validate(string name, MyConfigOptions options)
    {
        string vor=null;
        var rx = new Regex(@"^[a-zA-Z''-'\s]{1,40}$");
        var match = rx.Match(options.Key1);

        if (string.IsNullOrEmpty(match.Value))
        {
            vor = $"{options.Key1} doesn't match RegEx \n";
        }

        if ( options.Key2 < 0 || options.Key2 > 1000)
        {
            vor = $"{options.Key2} doesn't match Range 0 - 1000 \n";
        }

        if (_config.Key2 != default)
        {
            if(_config.Key3 <= _config.Key2)
            {
                vor +=  "Key3 must be > than Key2.";
            }
        }

        if (vor != null)
        {
            return ValidateOptionsResult.Fail(vor);
        }

        return ValidateOptionsResult.Success;
    }
}
```

`IValidateOptions` enables moving the validation code out of `StartUp` and into a class.

Using the preceding code, validation is enabled in `Startup.ConfigureServices` with the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MyConfigOptions>(Configuration.GetSection(
                                        MyConfigOptions.MyConfig));
    services.TryAddEnumerable(ServiceDescriptor.Singleton<IValidateOptions
                            <MyConfigOptions>, MyConfigValidation>());
    services.AddControllersWithViews();
}
```

Options post-configuration

Set post-configuration with IPostConfigureOptions<TOptions>. Post-configuration runs after all IConfigureOptions<TOptions> configuration occurs:

```
services.PostConfigure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

PostConfigure is available to post-configure named options:

```
services.PostConfigure<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Use PostConfigureAll to post-configure all configuration instances:

```
services.PostConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Accessing options during startup

IOptions<TOptions> and IOptionsMonitor<TOptions> can be used in `Startup.Configure`, since services are built before the `Configure` method executes.

```
public void Configure(IApplicationBuilder app,
    IOptionsMonitor<MyOptions> optionsAccessor)
{
    var option1 = optionsAccessor.CurrentValue.Option1;
}
```

Don't use IOptions<TOptions> or IOptionsMonitor<TOptions> in `Startup.ConfigureServices`. An inconsistent options state may exist due to the ordering of service registrations.

Options.ConfigurationExtensions NuGet package

The Microsoft.Extensions.Options.ConfigurationExtensions package is implicitly referenced in ASP.NET Core apps.

The options pattern uses classes to represent groups of related settings. When configuration settings are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- The Interface Segregation Principle (ISP) or Encapsulation: Scenarios (classes) that depend on configuration settings depend only on the configuration settings that they use.
- Separation of Concerns: Settings for different parts of the app aren't dependent or coupled to one another.

Options also provide a mechanism to validate configuration data. For more information, see the Options validation section.

View or download sample code (how to download)

Prerequisites

Reference the Microsoft.AspNetCore.App metapackage or add a package reference to the Microsoft.Extensions.Options.ConfigurationExtensions package.

Options interfaces

IOptionsMonitor<TOptions> is used to retrieve options and manage options notifications for `TOptions` instances. IOptionsMonitor<TOptions> supports the following scenarios:

- Change notifications
- Named options
- Reloadable configuration
- Selective options invalidation (IOptionsMonitorCache<TOptions>)

Post-configuration scenarios allow you to set or change options after all IConfigureOptions<TOptions> configuration occurs.

IOptionsFactory<TOptions> is responsible for creating new options instances. It has a single Create method. The default implementation takes all registered IConfigureOptions<TOptions> and IPostConfigureOptions<TOptions> and runs all the configurations first, followed by the post-configuration. It distinguishes between IConfigureNamedOptions<TOptions> and IConfigureOptions<TOptions> and only calls the appropriate interface.

IOptionsMonitorCache<TOptions> is used by IOptionsMonitor<TOptions> to cache `TOptions` instances. The IOptionsMonitorCache<TOptions> invalidates options instances in the monitor so that the value is recomputed (TryRemove). Values can be manually introduced with TryAdd. The Clear method is used when all named instances should be recreated on demand.

IOptionsSnapshot<TOptions> is useful in scenarios where options should be recomputed on every request. For more information, see the Reload configuration data with IOptionsSnapshot section.

IOptions<TOptions> can be used to support options. However, IOptions<TOptions> doesn't support the preceding scenarios of IOptionsMonitor<TOptions>. You may continue to use IOptions<TOptions> in existing frameworks and libraries that already use the IOptions<TOptions> interface and don't require the scenarios provided by IOptionsMonitor<TOptions>.

General options configuration

General options configuration is demonstrated as Example 1 in the sample app.

An options class must be non-abstract with a public parameterless constructor. The following class, `MyOptions`, has two properties, `Option1` and `Option2`. Setting default values is optional, but the class constructor in the

following example sets the default value of `Option1`. `Option2` has a default value set by initializing the property directly (*Models/MyOptions.cs*):

```
public class MyOptions
{
    public MyOptions()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

The `MyOptions` class is added to the service container with Configure and bound to configuration:

```
// Example #1: General configuration
// Register the Configuration instance which MyOptions binds against.
services.Configure<MyOptions>(Configuration);
```

The following page model uses constructor dependency injection with IOptionsMonitor<TOptions> to access the settings (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #1: Simple options
var option1 = _options.Option1;
var option2 = _options.Option2;
SimpleOptions = $"option1 = {option1}, option2 = {option2}";
```

The sample's *appsettings.json* file specifies values for `option1` and `option2`:

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

When the app is run, the page model's `OnGet` method returns a string showing the option class values:

```
option1 = value1_from_json, option2 = -1
```

NOTE

When using a custom ConfigurationBuilder to load options configuration from a settings file, confirm that the base path is set correctly:

```
var configBuilder = new ConfigurationBuilder()
   .SetBasePath(Directory.GetCurrentDirectory())
   .AddJsonFile("appsettings.json", optional: true);
var config = configBuilder.Build();

services.Configure<MyOptions>(config);
```

Explicitly setting the base path isn't required when loading options configuration from the settings file via CreateDefaultBuilder.

Configure simple options with a delegate

Configuring simple options with a delegate is demonstrated as Example 2 in the sample app.

Use a delegate to set options values. The sample app uses the `MyOptionsWithDelegateConfig` class (*Models/MyOptionsWithDelegateConfig.cs*):

```
public class MyOptionsWithDelegateConfig
{
    public MyOptionsWithDelegateConfig()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

In the following code, a second IConfigureOptions<TOptions> service is added to the service container. It uses a delegate to configure the binding with `MyOptionsWithDelegateConfig`:

```
// Example #2: Options bound and configured by a delegate
services.Configure<MyOptionsWithDelegateConfig>(myOptions =>
{
    myOptions.Option1 = "value1_configured_by_delegate";
    myOptions.Option2 = 500;
});
```

Index.cshtml.cs:

```
private readonly MyOptionsWithDelegateConfig _optionsWithDelegateConfig;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #2: Options configured by delegate
var delegate_config_option1 = _optionsWithDelegateConfig.Option1;
var delegate_config_option2 = _optionsWithDelegateConfig.Option2;
SimpleOptionsWithDelegateConfig =
    $"delegate_option1 = {delegate_config_option1}, " +
    $"delegate_option2 = {delegate_config_option2}";
```

You can add multiple configuration providers. Configuration providers are available from NuGet packages and are applied in the order that they're registered. For more information, see Configuration in ASP.NET Core.

Each call to Configure adds an IConfigureOptions<TOptions> service to the service container. In the preceding example, the values of `Option1` and `Option2` are both specified in *appsettings.json*, but the values of `Option1` and `Option2` are overridden by the configured delegate.

When more than one configuration service is enabled, the last configuration source specified *wins* and sets the configuration value. When the app is run, the page model's `OnGet` method returns a string showing the option class values:

```
delegate_option1 = value1_configured_by_delegate, delegate_option2 = 500
```

Suboptions configuration

Suboptions configuration is demonstrated as Example 3 in the sample app.

Apps should create options classes that pertain to specific scenario groups (classes) in the app. Parts of the app that require configuration values should only have access to the configuration values that they use.

When binding options to configuration, each property in the options type is bound to a configuration key of the form `property[:sub-property:]`. For example, the `MyOptions.Option1` property is bound to the key `Option1`

, which is read from the `option1` property in *appsettings.json*.

In the following code, a third IConfigureOptions<TOptions> service is added to the service container. It binds `MySubOptions` to the section `subsection` of the *appsettings.json* file:

```
// Example #3: Suboptions
// Bind options using a sub-section of the appsettings.json file.
services.Configure<MySubOptions>(Configuration.GetSection("subsection"));
```

The `GetSection` method requires the Microsoft.Extensions.Configuration namespace.

The sample's *appsettings.json* file defines a `subsection` member with keys for `suboption1` and `suboption2`:

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

The `MySubOptions` class defines properties, `SubOption1` and `SubOption2`, to hold the options values (*Models/MySubOptions.cs*):

```
public class MySubOptions
{
    public MySubOptions()
    {
        // Set default values.
        SubOption1 = "value1_from_ctor";
        SubOption2 = 5;
    }

    public string SubOption1 { get; set; }
    public int SubOption2 { get; set; }
}
```

The page model's `OnGet` method returns a string with the options values (*Pages/Index.cshtml.cs*):

```
private readonly MySubOptions _subOptions;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #3: Suboptions
var subOption1 = _subOptions.SubOption1;
var subOption2 = _subOptions.SubOption2;
SubOptions = $"subOption1 = {subOption1}, subOption2 = {subOption2}";
```

When the app is run, the `OnGet` method returns a string showing the suboption class values:

```
subOption1 = subvalue1_from_json, subOption2 = 200
```

Options injection

Options injection is demonstrated as Example 4 in the sample app.

Inject IOptionsMonitor<TOptions> into:

- A Razor page or MVC view with the `@inject` Razor directive.
- A page or view model.

The following example from the sample app injects IOptionsMonitor<TOptions> into a page model (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #4: Bind options directly to the page
MyOptions = _options;
```

The sample app shows how to inject `IOptionsMonitor<MyOptions>` with an `@inject` directive:

```
@page
@model IndexModel
@using Microsoft.Extensions.Options
@inject IOptionsMonitor<MyOptions> OptionsAccessor
@{
    ViewData["Title"] = "Options Sample";
}

<h1>@ViewData["Title"]</h1>
```

When the app is run, the options values are shown in the rendered page:

Example #4: Model and injected options

Options provided by the model

Options provided by the model: `@Model.MyOptions.Option1` and `@Model.MyOptions.Option2`

Option1: value1_from_json

Option2: -1

Options injected into the page

Options injected into the page: `@inject IOptions<MyOptions> OptionsAccessor` with `@OptionsAccessor.Value.Option1` and `@OptionsAccessor.Value.Option2`

Option1: value1_from_json

Option2: -1

Reload configuration data with IOptionsSnapshot

Reloading configuration data with IOptionsSnapshot<TOptions> is demonstrated in Example 5 in the sample app.

Using IOptionsSnapshot<TOptions>, options are computed once per request when accessed and cached for the lifetime of the request.

The difference between `IOptionsMonitor` and `IOptionsSnapshot` is that:

- `IOptionsMonitor` is a singleton service that retrieves current option values at any time, which is especially useful in singleton dependencies.
- `IOptionsSnapshot` is a scoped service and provides a snapshot of the options at the time the `IOptionsSnapshot<T>` object is constructed. Options snapshots are designed for use with transient and scoped dependencies.

The following example demonstrates how a new IOptionsSnapshot<TOptions> is created after *appsettings.json* changes (*Pages/Index.cshtml.cs*). Multiple requests to the server return constant values provided by the *appsettings.json* file until the file is changed and configuration reloads.

```
private readonly MyOptions _snapshotOptions;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #5: Snapshot options
var snapshotOption1 = _snapshotOptions.Option1;
var snapshotOption2 = _snapshotOptions.Option2;
SnapshotOptions =
    $"snapshot option1 = {snapshotOption1}, " +
    $"snapshot option2 = {snapshotOption2}";
```

The following image shows the initial `option1` and `option2` values loaded from the *appsettings.json* file:

```
snapshot option1 = value1_from_json, snapshot option2 = -1
```

Change the values in the *appsettings.json* file to `value1_from_json UPDATED` and `200`. Save the *appsettings.json* file. Refresh the browser to see that the options values are updated:

```
snapshot option1 = value1_from_json UPDATED, snapshot option2 = 200
```

Named options support with IConfigureNamedOptions

Named options support with IConfigureNamedOptions<TOptions> is demonstrated as Example 6 in the sample app.

Named options support allows the app to distinguish between named options configurations. In the sample app, named options are declared with OptionsServiceCollectionExtensions.Configure, which calls the ConfigureNamedOptions<TOptions>.Configure extension method. Named options are case sensitive.

```
// Example #6: Named options (named_options_1)
// Register the ConfigurationBuilder instance which MyOptions binds against.
// Specify that the options loaded from configuration are named
// "named_options_1".
services.Configure<MyOptions>("named_options_1", Configuration);

// Example #6: Named options (named_options_2)
// Specify that the options loaded from the MyOptions class are named
// "named_options_2".
// Use a delegate to configure option values.
services.Configure<MyOptions>("named_options_2", myOptions =>
{
    myOptions.Option1 = "named_options_2_value1_from_action";
});
```

The sample app accesses the named options with Get (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _named_options_1;
private readonly MyOptions _named_options_2;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #6: Named options
var named_options_1 =
    $"named_options_1: option1 = {_named_options_1.Option1}, " +
    $"option2 = {_named_options_1.Option2}";
var named_options_2 =
    $"named_options_2: option1 = {_named_options_2.Option1}, " +
    $"option2 = {_named_options_2.Option2}";
NamedOptions = $"{named_options_1} {named_options_2}";
```

Running the sample app, the named options are returned:

```
named_options_1: option1 = value1_from_json, option2 = -1
named_options_2: option1 = named_options_2_value1_from_action, option2 = 5
```

`named_options_1` values are provided from configuration, which are loaded from the *appsettings.json* file. `named_options_2` values are provided by:

- The `named_options_2` delegate in `ConfigureServices` for `Option1`.
- The default value for `Option2` provided by the `MyOptions` class.

Configure all options with the ConfigureAll method

Configure all options instances with the ConfigureAll method. The following code configures `Option1` for all configuration instances with a common value. Add the following code manually to the `Startup.ConfigureServices` method:

```
services.ConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "ConfigureAll replacement value";
});
```

Running the sample app after adding the code produces the following result:

```
named_options_1: option1 = ConfigureAll replacement value, option2 = -1
named_options_2: option1 = ConfigureAll replacement value, option2 = 5
```

NOTE

All options are named instances. Existing IConfigureOptions<TOptions> instances are treated as targeting the `Options.DefaultName` instance, which is `string.Empty`. IConfigureNamedOptions<TOptions> also implements IConfigureOptions<TOptions>. The default implementation of the IOptionsFactory<TOptions> has logic to use each appropriately. The `null` named option is used to target all of the named instances instead of a specific named instance (ConfigureAll and PostConfigureAll use this convention).

OptionsBuilder API

OptionsBuilder<TOptions> is used to configure `TOptions` instances. `OptionsBuilder` streamlines creating named options as it's only a single parameter to the initial `AddOptions<TOptions>(string optionsName)` call instead of appearing in all of the subsequent calls. Options validation and the `ConfigureOptions` overloads that accept service dependencies are only available via `OptionsBuilder`.

```
// Options.DefaultName = "" is used.
services.AddOptions<MyOptions>().Configure(o => o.Property = "default");

services.AddOptions<MyOptions>("optionalName")
    .Configure(o => o.Property = "named");
```

Use DI services to configure options

You can access other services from dependency injection while configuring options in two ways:

- Pass a configuration delegate to Configure on OptionsBuilder<TOptions>. OptionsBuilder<TOptions> provides overloads of Configure that allow you to use up to five services to configure options:

  ```
  services.AddOptions<MyOptions>("optionalName")
      .Configure<Service1, Service2, Service3, Service4, Service5>(
          (o, s, s2, s3, s4, s5) =>
              o.Property = DoSomethingWith(s, s2, s3, s4, s5));
  ```

- Create your own type that implements IConfigureOptions<TOptions> or IConfigureNamedOptions<TOptions> and register the type as a service.

We recommend passing a configuration delegate to Configure, since creating a service is more complex. Creating your own type is equivalent to what the framework does for you when you use Configure. Calling Configure registers a transient generic IConfigureNamedOptions<TOptions>, which has a constructor that accepts the generic service types specified.

Options validation

Options validation allows you to validate options when options are configured. Call `Validate` with a validation method that returns `true` if options are valid and `false` if they aren't valid:

```
// Registration
services.AddOptions<MyOptions>("optionalOptionsName")
    .Configure(o => { }) // Configure the options
    .Validate(o => YourValidationShouldReturnTrueIfValid(o),
        "custom error");

// Consumption
var monitor = services.BuildServiceProvider()
    .GetService<IOptionsMonitor<MyOptions>>();

try
{
    var options = monitor.Get("optionalOptionsName");
}
catch (OptionsValidationException e)
{
   // e.OptionsName returns "optionalOptionsName"
   // e.OptionsType returns typeof(MyOptions)
   // e.Failures returns a list of errors, which would contain
   //     "custom error"
}
```

The preceding example sets the named options instance to `optionalOptionsName`. The default options instance is `Options.DefaultName`.

Validation runs when the options instance is created. An options instance is guaranteed to pass validation the first time it's accessed.

> **IMPORTANT**
>
> Options validation doesn't guard against options modifications after the options instance is created. For example, `IOptionsSnapshot` options are created and validated once per request when the options are first accessed. The `IOptionsSnapshot` options aren't validated again on subsequent access attempts *for the same request*.

The `Validate` method accepts a `Func<TOptions, bool>`. To fully customize validation, implement `IValidateOptions<TOptions>`, which allows:

- Validation of multiple options types:
  `class ValidateTwo : IValidateOptions<Option1>, IValidationOptions<Option2>`
- Validation that depends on another option type:
  `public DependsOnAnotherOptionValidator(IOptionsMonitor<AnotherOption> options)`

`IValidateOptions` validates:

- A specific named options instance.
- All options when `name` is `null`.

Return a `ValidateOptionsResult` from your implementation of the interface:

```
public interface IValidateOptions<TOptions> where TOptions : class
{
    ValidateOptionsResult Validate(string name, TOptions options);
}
```

Data Annotation-based validation is available from the Microsoft.Extensions.Options.DataAnnotations package by calling the ValidateDataAnnotations method on `OptionsBuilder<TOptions>`.
`Microsoft.Extensions.Options.DataAnnotations` is included in the Microsoft.AspNetCore.App metapackage.

```
using Microsoft.Extensions.DependencyInjection;

private class AnnotatedOptions
{
    [Required]
    public string Required { get; set; }

    [StringLength(5, ErrorMessage = "Too long.")]
    public string StringLength { get; set; }

    [Range(-5, 5, ErrorMessage = "Out of range.")]
    public int IntRange { get; set; }
}

[Fact]
public void CanValidateDataAnnotations()
{
    var services = new ServiceCollection();
    services.AddOptions<AnnotatedOptions>()
        .Configure(o =>
        {
            o.StringLength = "111111";
            o.IntRange = 10;
            o.Custom = "nowhere";
        })
        .ValidateDataAnnotations();

    var sp = services.BuildServiceProvider();

    var error = Assert.Throws<OptionsValidationException>(() =>
        sp.GetRequiredService<IOptionsMonitor<AnnotatedOptions>>().CurrentValue);
    ValidateFailure<AnnotatedOptions>(error, Options.DefaultName, 1,
        "DataAnnotation validation failed for members Required " +
            "with the error 'The Required field is required.'.",
        "DataAnnotation validation failed for members StringLength " +
            "with the error 'Too long.'.",
        "DataAnnotation validation failed for members IntRange " +
            "with the error 'Out of range.'.");
}
```

Eager validation (fail fast at startup) is under consideration for a future release.

Options post-configuration

Set post-configuration with IPostConfigureOptions<TOptions>. Post-configuration runs after all IConfigureOptions<TOptions> configuration occurs:

```
services.PostConfigure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

PostConfigure is available to post-configure named options:

```
services.PostConfigure<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Use PostConfigureAll to post-configure all configuration instances:

```
services.PostConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Accessing options during startup

IOptions<TOptions> and IOptionsMonitor<TOptions> can be used in `Startup.Configure`, since services are built before the `Configure` method executes.

```
public void Configure(IApplicationBuilder app, IOptionsMonitor<MyOptions> optionsAccessor)
{
    var option1 = optionsAccessor.CurrentValue.Option1;
}
```

Don't use IOptions<TOptions> or IOptionsMonitor<TOptions> in `Startup.ConfigureServices`. An inconsistent options state may exist due to the ordering of service registrations.

The options pattern uses classes to represent groups of related settings. When configuration settings are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- The Interface Segregation Principle (ISP) or Encapsulation: Scenarios (classes) that depend on configuration settings depend only on the configuration settings that they use.
- Separation of Concerns: Settings for different parts of the app aren't dependent or coupled to one another.

Options also provide a mechanism to validate configuration data. For more information, see the Options validation section.

View or download sample code (how to download)

Prerequisites

Reference the Microsoft.AspNetCore.App metapackage or add a package reference to the Microsoft.Extensions.Options.ConfigurationExtensions package.

Options interfaces

IOptionsMonitor<TOptions> is used to retrieve options and manage options notifications for `TOptions` instances. IOptionsMonitor<TOptions> supports the following scenarios:

- Change notifications
- Named options
- Reloadable configuration
- Selective options invalidation (IOptionsMonitorCache<TOptions>)

Post-configuration scenarios allow you to set or change options after all IConfigureOptions<TOptions> configuration occurs.

IOptionsFactory<TOptions> is responsible for creating new options instances. It has a single Create method. The default implementation takes all registered IConfigureOptions<TOptions> and IPostConfigureOptions<TOptions> and runs all the configurations first, followed by the post-configuration. It distinguishes between IConfigureNamedOptions<TOptions> and IConfigureOptions<TOptions> and only calls the appropriate interface.

IOptionsMonitorCache<TOptions> is used by IOptionsMonitor<TOptions> to cache `TOptions` instances. The

IOptionsMonitorCache<TOptions> invalidates options instances in the monitor so that the value is recomputed (TryRemove). Values can be manually introduced with TryAdd. The Clear method is used when all named instances should be recreated on demand.

IOptionsSnapshot<TOptions> is useful in scenarios where options should be recomputed on every request. For more information, see the Reload configuration data with IOptionsSnapshot section.

IOptions<TOptions> can be used to support options. However, IOptions<TOptions> doesn't support the preceding scenarios of IOptionsMonitor<TOptions>. You may continue to use IOptions<TOptions> in existing frameworks and libraries that already use the IOptions<TOptions> interface and don't require the scenarios provided by IOptionsMonitor<TOptions>.

General options configuration

General options configuration is demonstrated as Example 1 in the sample app.

An options class must be non-abstract with a public parameterless constructor. The following class, `MyOptions`, has two properties, `Option1` and `Option2`. Setting default values is optional, but the class constructor in the following example sets the default value of `Option1`. `Option2` has a default value set by initializing the property directly (*Models/MyOptions.cs*):

```
public class MyOptions
{
    public MyOptions()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

The `MyOptions` class is added to the service container with Configure and bound to configuration:

```
// Example #1: General configuration
// Register the Configuration instance which MyOptions binds against.
services.Configure<MyOptions>(Configuration);
```

The following page model uses constructor dependency injection with IOptionsMonitor<TOptions> to access the settings (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #1: Simple options
var option1 = _options.Option1;
var option2 = _options.Option2;
SimpleOptions = $"option1 = {option1}, option2 = {option2}";
```

The sample's *appsettings.json* file specifies values for `option1` and `option2`:

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

When the app is run, the page model's `OnGet` method returns a string showing the option class values:

```
option1 = value1_from_json, option2 = -1
```

NOTE

When using a custom ConfigurationBuilder to load options configuration from a settings file, confirm that the base path is set correctly:

```
var configBuilder = new ConfigurationBuilder()
   .SetBasePath(Directory.GetCurrentDirectory())
   .AddJsonFile("appsettings.json", optional: true);
var config = configBuilder.Build();

services.Configure<MyOptions>(config);
```

Explicitly setting the base path isn't required when loading options configuration from the settings file via CreateDefaultBuilder.

Configure simple options with a delegate

Configuring simple options with a delegate is demonstrated as Example 2 in the sample app.

Use a delegate to set options values. The sample app uses the `MyOptionsWithDelegateConfig` class (*Models/MyOptionsWithDelegateConfig.cs*):

```
public class MyOptionsWithDelegateConfig
{
    public MyOptionsWithDelegateConfig()
    {
        // Set default value.
        Option1 = "value1_from_ctor";
    }

    public string Option1 { get; set; }
    public int Option2 { get; set; } = 5;
}
```

In the following code, a second IConfigureOptions<TOptions> service is added to the service container. It uses a delegate to configure the binding with `MyOptionsWithDelegateConfig`:

```
// Example #2: Options bound and configured by a delegate
services.Configure<MyOptionsWithDelegateConfig>(myOptions =>
{
    myOptions.Option1 = "value1_configured_by_delegate";
    myOptions.Option2 = 500;
});
```

Index.cshtml.cs:

```
private readonly MyOptionsWithDelegateConfig _optionsWithDelegateConfig;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #2: Options configured by delegate
var delegate_config_option1 = _optionsWithDelegateConfig.Option1;
var delegate_config_option2 = _optionsWithDelegateConfig.Option2;
SimpleOptionsWithDelegateConfig =
    $"delegate_option1 = {delegate_config_option1}, " +
    $"delegate_option2 = {delegate_config_option2}";
```

You can add multiple configuration providers. Configuration providers are available from NuGet packages and are applied in the order that they're registered. For more information, see Configuration in ASP.NET Core.

Each call to Configure adds an IConfigureOptions<TOptions> service to the service container. In the preceding example, the values of `Option1` and `Option2` are both specified in *appsettings.json*, but the values of `Option1`

and `Option2` are overridden by the configured delegate.

When more than one configuration service is enabled, the last configuration source specified *wins* and sets the configuration value. When the app is run, the page model's `OnGet` method returns a string showing the option class values:

```
delegate_option1 = value1_configured_by_delegate, delegate_option2 = 500
```

Suboptions configuration

Suboptions configuration is demonstrated as Example 3 in the sample app.

Apps should create options classes that pertain to specific scenario groups (classes) in the app. Parts of the app that require configuration values should only have access to the configuration values that they use.

When binding options to configuration, each property in the options type is bound to a configuration key of the form `property[:sub-property:]`. For example, the `MyOptions.Option1` property is bound to the key `Option1`, which is read from the `option1` property in *appsettings.json*.

In the following code, a third IConfigureOptions<TOptions> service is added to the service container. It binds `MySubOptions` to the section `subsection` of the *appsettings.json* file:

```
// Example #3: Suboptions
// Bind options using a sub-section of the appsettings.json file.
services.Configure<MySubOptions>(Configuration.GetSection("subsection"));
```

The `GetSection` method requires the Microsoft.Extensions.Configuration namespace.

The sample's *appsettings.json* file defines a `subsection` member with keys for `suboption1` and `suboption2`:

```
{
  "option1": "value1_from_json",
  "option2": -1,
  "subsection": {
    "suboption1": "subvalue1_from_json",
    "suboption2": 200
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

The `MySubOptions` class defines properties, `SubOption1` and `SubOption2`, to hold the options values (*Models/MySubOptions.cs*):

```
public class MySubOptions
{
    public MySubOptions()
    {
        // Set default values.
        SubOption1 = "value1_from_ctor";
        SubOption2 = 5;
    }

    public string SubOption1 { get; set; }
    public int SubOption2 { get; set; }
}
```

The page model's `OnGet` method returns a string with the options values (*Pages/Index.cshtml.cs*):

```
private readonly MySubOptions _subOptions;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #3: Suboptions
var subOption1 = _subOptions.SubOption1;
var subOption2 = _subOptions.SubOption2;
SubOptions = $"subOption1 = {subOption1}, subOption2 = {subOption2}";
```

When the app is run, the `OnGet` method returns a string showing the suboption class values:

```
subOption1 = subvalue1_from_json, subOption2 = 200
```

Options provided by a view model or with direct view injection

Options provided by a view model or with direct view injection is demonstrated as Example 4 in the sample app.

Options can be supplied in a view model or by injecting IOptionsMonitor<TOptions> directly into a view (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _options;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #4: Bind options directly to the page
MyOptions = _options;
```

The sample app shows how to inject `IOptionsMonitor<MyOptions>` with an `@inject` directive:

```
@page
@model IndexModel
@using Microsoft.Extensions.Options
@inject IOptionsMonitor<MyOptions> OptionsAccessor
@{
    ViewData["Title"] = "Options Sample";
}

<h1>@ViewData["Title"]</h1>
```

When the app is run, the options values are shown in the rendered page:

Example #4: Model and injected options

Options provided by the model

Options provided by the model: `@Model.MyOptions.Option1` and `@Model.MyOptions.Option2`

Option1: value1_from_json

Option2: -1

Options injected into the page

Options injected into the page: `@inject IOptions<MyOptions> OptionsAccessor` with `@OptionsAccessor.Value.Option1` and `@OptionsAccessor.Value.Option2`

Option1: value1_from_json

Option2: -1

Reload configuration data with IOptionsSnapshot

Reloading configuration data with IOptionsSnapshot<TOptions> is demonstrated in Example 5 in the sample app.

IOptionsSnapshot<TOptions> supports reloading options with minimal processing overhead.

Options are computed once per request when accessed and cached for the lifetime of the request.

The following example demonstrates how a new IOptionsSnapshot<TOptions> is created after *appsettings.json*

changes (*Pages/Index.cshtml.cs*). Multiple requests to the server return constant values provided by the *appsettings.json* file until the file is changed and configuration reloads.

```
private readonly MyOptions _snapshotOptions;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #5: Snapshot options
var snapshotOption1 = _snapshotOptions.Option1;
var snapshotOption2 = _snapshotOptions.Option2;
SnapshotOptions =
    $"snapshot option1 = {snapshotOption1}, " +
    $"snapshot option2 = {snapshotOption2}";
```

The following image shows the initial `option1` and `option2` values loaded from the *appsettings.json* file:

```
snapshot option1 = value1_from_json, snapshot option2 = -1
```

Change the values in the *appsettings.json* file to `value1_from_json UPDATED` and `200`. Save the *appsettings.json* file. Refresh the browser to see that the options values are updated:

```
snapshot option1 = value1_from_json UPDATED, snapshot option2 = 200
```

Named options support with IConfigureNamedOptions

Named options support with IConfigureNamedOptions<TOptions> is demonstrated as Example 6 in the sample app.

Named options support allows the app to distinguish between named options configurations. In the sample app, named options are declared with OptionsServiceCollectionExtensions.Configure, which calls the ConfigureNamedOptions<TOptions>.Configure extension method. Named options are case sensitive.

```
// Example #6: Named options (named_options_1)
// Register the ConfigurationBuilder instance which MyOptions binds against.
// Specify that the options loaded from configuration are named
// "named_options_1".
services.Configure<MyOptions>("named_options_1", Configuration);

// Example #6: Named options (named_options_2)
// Specify that the options loaded from the MyOptions class are named
// "named_options_2".
// Use a delegate to configure option values.
services.Configure<MyOptions>("named_options_2", myOptions =>
{
    myOptions.Option1 = "named_options_2_value1_from_action";
});
```

The sample app accesses the named options with Get (*Pages/Index.cshtml.cs*):

```
private readonly MyOptions _named_options_1;
private readonly MyOptions _named_options_2;
```

```
public IndexModel(
    IOptionsMonitor<MyOptions> optionsAccessor,
    IOptionsMonitor<MyOptionsWithDelegateConfig> optionsAccessorWithDelegateConfig,
    IOptionsMonitor<MySubOptions> subOptionsAccessor,
    IOptionsSnapshot<MyOptions> snapshotOptionsAccessor,
    IOptionsSnapshot<MyOptions> namedOptionsAccessor)
{
    _options = optionsAccessor.CurrentValue;
    _optionsWithDelegateConfig = optionsAccessorWithDelegateConfig.CurrentValue;
    _subOptions = subOptionsAccessor.CurrentValue;
    _snapshotOptions = snapshotOptionsAccessor.Value;
    _named_options_1 = namedOptionsAccessor.Get("named_options_1");
    _named_options_2 = namedOptionsAccessor.Get("named_options_2");
}
```

```
// Example #6: Named options
var named_options_1 =
    $"named_options_1: option1 = {_named_options_1.Option1}, " +
    $"option2 = {_named_options_1.Option2}";
var named_options_2 =
    $"named_options_2: option1 = {_named_options_2.Option1}, " +
    $"option2 = {_named_options_2.Option2}";
NamedOptions = $"{named_options_1} {named_options_2}";
```

Running the sample app, the named options are returned:

```
named_options_1: option1 = value1_from_json, option2 = -1
named_options_2: option1 = named_options_2_value1_from_action, option2 = 5
```

`named_options_1` values are provided from configuration, which are loaded from the *appsettings.json* file.
`named_options_2` values are provided by:

- The `named_options_2` delegate in `ConfigureServices` for `Option1`.
- The default value for `Option2` provided by the `MyOptions` class.

Configure all options with the ConfigureAll method

Configure all options instances with the ConfigureAll method. The following code configures `Option1` for all configuration instances with a common value. Add the following code manually to the `Startup.ConfigureServices` method:

```
services.ConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "ConfigureAll replacement value";
});
```

Running the sample app after adding the code produces the following result:

```
named_options_1: option1 = ConfigureAll replacement value, option2 = -1
named_options_2: option1 = ConfigureAll replacement value, option2 = 5
```

> **NOTE**
>
> All options are named instances. Existing IConfigureOptions<TOptions> instances are treated as targeting the `Options.DefaultName` instance, which is `string.Empty`. IConfigureNamedOptions<TOptions> also implements IConfigureOptions<TOptions>. The default implementation of the IOptionsFactory<TOptions> has logic to use each appropriately. The `null` named option is used to target all of the named instances instead of a specific named instance (ConfigureAll and PostConfigureAll use this convention).

OptionsBuilder API

OptionsBuilder<TOptions> is used to configure `TOptions` instances. `OptionsBuilder` streamlines creating named options as it's only a single parameter to the initial `AddOptions<TOptions>(string optionsName)` call instead of appearing in all of the subsequent calls. Options validation and the `ConfigureOptions` overloads that accept service dependencies are only available via `OptionsBuilder`.

```
// Options.DefaultName = "" is used.
services.AddOptions<MyOptions>().Configure(o => o.Property = "default");

services.AddOptions<MyOptions>("optionalName")
    .Configure(o => o.Property = "named");
```

Use DI services to configure options

You can access other services from dependency injection while configuring options in two ways:

- Pass a configuration delegate to Configure on OptionsBuilder<TOptions>. OptionsBuilder<TOptions> provides overloads of Configure that allow you to use up to five services to configure options:

  ```
  services.AddOptions<MyOptions>("optionalName")
      .Configure<Service1, Service2, Service3, Service4, Service5>(
          (o, s, s2, s3, s4, s5) =>
              o.Property = DoSomethingWith(s, s2, s3, s4, s5));
  ```

- Create your own type that implements IConfigureOptions<TOptions> or IConfigureNamedOptions<TOptions> and register the type as a service.

We recommend passing a configuration delegate to Configure, since creating a service is more complex. Creating your own type is equivalent to what the framework does for you when you use Configure. Calling Configure registers a transient generic IConfigureNamedOptions<TOptions>, which has a constructor that

accepts the generic service types specified.

Options post-configuration

Set post-configuration with IPostConfigureOptions<TOptions>. Post-configuration runs after all IConfigureOptions<TOptions> configuration occurs:

```
services.PostConfigure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

PostConfigure is available to post-configure named options:

```
services.PostConfigure<MyOptions>("named_options_1", myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Use PostConfigureAll to post-configure all configuration instances:

```
services.PostConfigureAll<MyOptions>(myOptions =>
{
    myOptions.Option1 = "post_configured_option1_value";
});
```

Accessing options during startup

IOptions<TOptions> and IOptionsMonitor<TOptions> can be used in `Startup.Configure`, since services are built before the `Configure` method executes.

```
public void Configure(IApplicationBuilder app, IOptionsMonitor<MyOptions> optionsAccessor)
{
    var option1 = optionsAccessor.CurrentValue.Option1;
}
```

Don't use IOptions<TOptions> or IOptionsMonitor<TOptions> in `Startup.ConfigureServices`. An inconsistent options state may exist due to the ordering of service registrations.

Additional resources

- Configuration in ASP.NET Core

Use multiple environments in ASP.NET Core

9/22/2020 • 21 minutes to read • Edit Online

By Rick Anderson and Kirk Larkin

ASP.NET Core configures app behavior based on the runtime environment using an environment variable.

View or download sample code (how to download)

Environments

To determine the runtime environment, ASP.NET Core reads from the following environment variables:

1. DOTNET_ENVIRONMENT
2. `ASPNETCORE_ENVIRONMENT` when ConfigureWebHostDefaults is called. The default ASP.NET Core web app templates call `ConfigureWebHostDefaults`. The `ASPNETCORE_ENVIRONMENT` value overrides `DOTNET_ENVIRONMENT`.

`IHostEnvironment.EnvironmentName` can be set to any value, but the following values are provided by the framework:

- Development : The launchSettings.json file sets `ASPNETCORE_ENVIRONMENT` to `Development` on the local machine.
- Staging
- Production : The default if `DOTNET_ENVIRONMENT` and `ASPNETCORE_ENVIRONMENT` have not been set.

The following code:

- Calls UseDeveloperExceptionPage when `ASPNETCORE_ENVIRONMENT` is set to `Development`.
- Calls UseExceptionHandler when the value of `ASPNETCORE_ENVIRONMENT` is set to `Staging`, `Production`, or `Staging_2`.
- Injects IWebHostEnvironment into `Startup.Configure`. This approach is useful when the app only requires adjusting `Startup.Configure` for a few environments with minimal code differences per environment.
- Is similar to the code generated by the ASP.NET Core templates.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

The Environment Tag Helper uses the value of IHostEnvironment.EnvironmentName to include or exclude markup in the element:

```
<environment include="Development">
    <div>The effective tag is: &lt;environment include="Development"&gt;</div>
</environment>
<environment exclude="Development">
    <div>The effective tag is: &lt;environment exclude="Development"&gt;</div>
</environment>
<environment include="Staging,Development,Staging_2">
    <div>
        The effective tag is:
        &lt;environment include="Staging,Development,Staging_2"&gt;
    </div>
</environment>
```

The About page from the sample code includes the preceding markup and displays the value of `IWebHostEnvironment.EnvironmentName`.

On Windows and macOS, environment variables and values aren't case-sensitive. Linux environment variables and values are **case-sensitive** by default.

Create EnvironmentsSample

The sample code used in this document is based on a Razor Pages project named *EnvironmentsSample*.

The following code creates and runs a web app named *EnvironmentsSample*:

```
dotnet new webapp -o EnvironmentsSample
cd EnvironmentsSample
dotnet run --verbosity normal
```

When the app runs, it displays some of the following output:

```
Using launch settings from c:\tmp\EnvironmentsSample\Properties\launchSettings.json
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: c:\tmp\EnvironmentsSample
```

Development and launchSettings.json

The development environment can enable features that shouldn't be exposed in production. For example, the ASP.NET Core templates enable the Developer Exception Page in the development environment.

The environment for local machine development can be set in the *Properties\launchSettings.json* file of the project. Environment values set in *launchSettings.json* override values set in the system environment.

The *launchSettings.json* file:

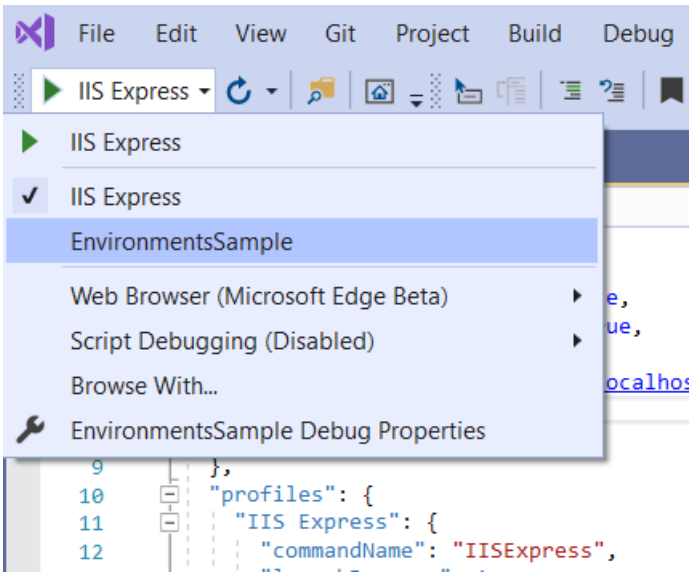
- Is only used on the local development machine.
- Is not deployed.
- contains profile settings.

The following JSON shows the *launchSettings.json* file for an ASP.NET Core web projected named *EnvironmentsSample* created with Visual Studio or `dotnet new`:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:64645",
      "sslPort": 44366
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "EnvironmentsSample": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The preceding markup contains two profiles:

- `IIS Express`: The default profile used when launching the app from Visual Studio. The `"commandName"` key has the value `"IISExpress"`, therefore, IISExpress is the web server. You can set the launch profile to the project or any other profile included. For example, in the image below, selecting the project name launches the Kestrel web server.

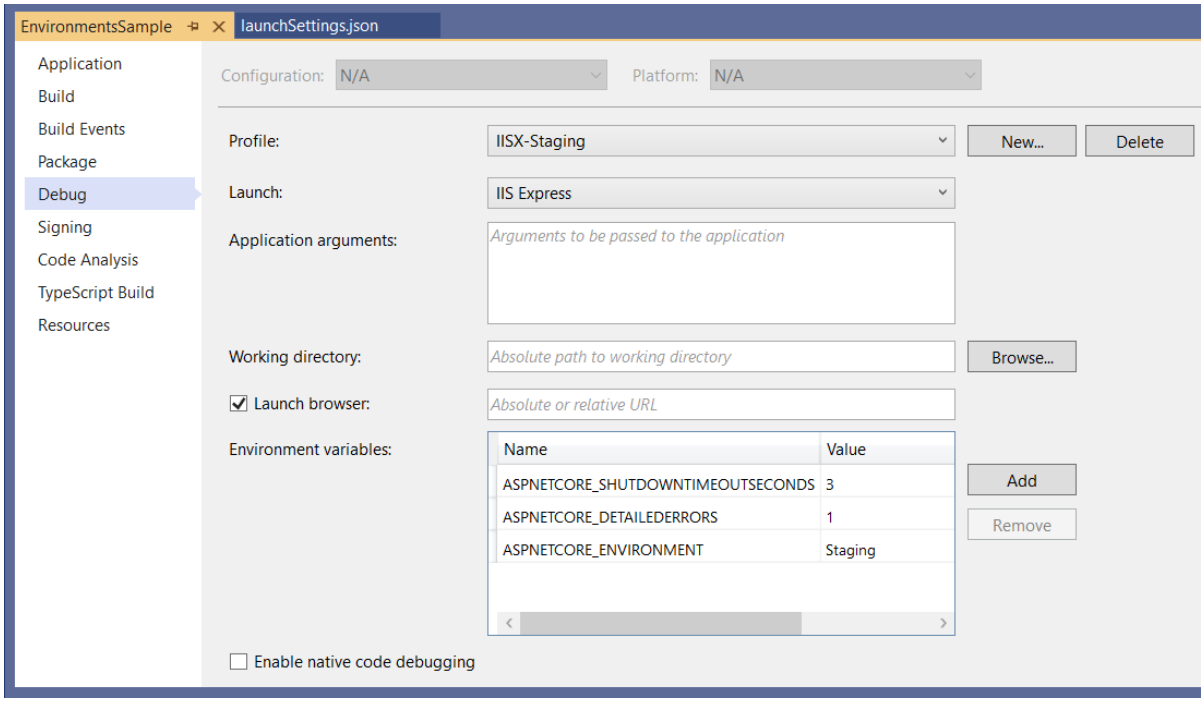


- `EnvironmentsSample`: The profile name is the project name. This profile is used by default when launching the app with `dotnet run`. The `"commandName"` key has the value `"Project"`, therefore, the Kestrel web server is launched.

The value of `commandName` can specify the web server to launch. `commandName` can be any one of the following:

- `IISExpress` : Launches IIS Express.
- `IIS` : No web server launched. IIS is expected to be available.
- `Project` : Launches Kestrel.

The Visual Studio project properties **Debug** tab provides a GUI to edit the *launchSettings.json* file. Changes made to project profiles may not take effect until the web server is restarted. Kestrel must be restarted before it can detect changes made to its environment.



The following *launchSettings.json* file contains multiple profiles:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:64645",
      "sslPort": 44366
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IISX-Production": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    },
    "IISX-Staging": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
      }
    },
    "EnvironmentsSample": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "KestrelStaging": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging"
      }
    }
  }
}
```

Profiles can be selected:

- From the Visual Studio UI.
- Using the `dotnet run` command in a command shell with the `--launch-profile` option set to the profile's name. *This approach only supports Kestrel profiles.*

  ```
  dotnet run --launch-profile "SampleApp"
  ```

> **WARNING**
>
> *launchSettings.json* shouldn't store secrets. The Secret Manager tool can be used to store secrets for local development.

When using Visual Studio Code, environment variables can be set in the *.vscode/launch.json* file. The following example sets several Host configuration values environment variables:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            // Configuration ommitted for brevity.
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development",
                "ASPNETCORE_URLS": "https://localhost:5001",
                "ASPNETCORE_DETAILEDERRORS": "1",
                "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
            },
            // Configuration ommitted for brevity.
```

The *.vscode/launch.json* file is only used by Visual Studio Code.

Production

The production environment should be configured to maximize security, performance, and application robustness. Some common settings that differ from development include:

- Caching.
- Client-side resources are bundled, minified, and potentially served from a CDN.
- Diagnostic error pages disabled.
- Friendly error pages enabled.
- Production logging and monitoring enabled. For example, using Application Insights.

Set the environment

It's often useful to set a specific environment for testing with an environment variable or platform setting. If the environment isn't set, it defaults to `Production`, which disables most debugging features. The method for setting the environment depends on the operating system.

When the host is built, the last environment setting read by the app determines the app's environment. The app's environment can't be changed while the app is running.

The About page from the sample code displays the value of `IWebHostEnvironment.EnvironmentName`.

Azure App Service

To set the environment in Azure App Service, perform the following steps:

1. Select the app from the **App Services** blade.
2. In the **Settings** group, select the **Configuration** blade.
3. In the **Application settings** tab, select **New application setting**.
4. In the **Add/Edit application setting** window, provide `ASPNETCORE_ENVIRONMENT` for the **Name**. For **Value**, provide the environment (for example, `Staging`).
5. Select the **Deployment slot setting** check box if you wish the environment setting to remain with the

current slot when deployment slots are swapped. For more information, see Set up staging environments in Azure App Service in the Azure documentation.

6. Select **OK** to close the **Add/Edit application setting** window.
7. Select **Save** at the top of the **Configuration** blade.

Azure App Service automatically restarts the app after an app setting is added, changed, or deleted in the Azure portal.

Windows

Environment values in *launchSettings.json* override values set in the system environment.

To set the `ASPNETCORE_ENVIRONMENT` for the current session when the app is started using dotnet run, the following commands are used:

Command prompt

```
set ASPNETCORE_ENVIRONMENT=Staging
dotnet run --no-launch-profile
```

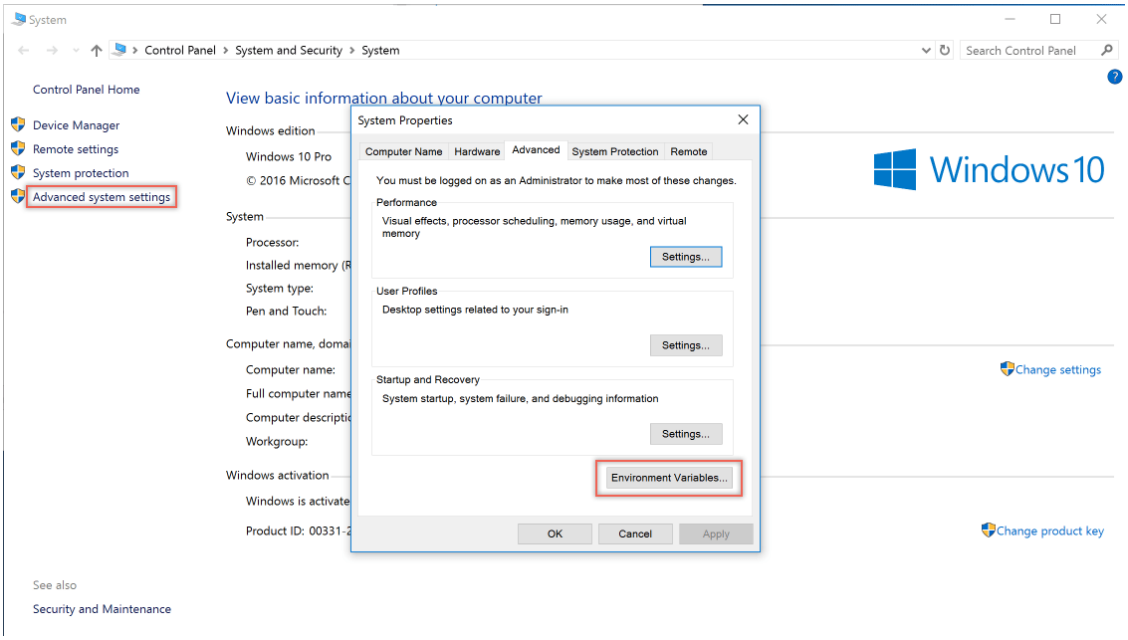
PowerShell

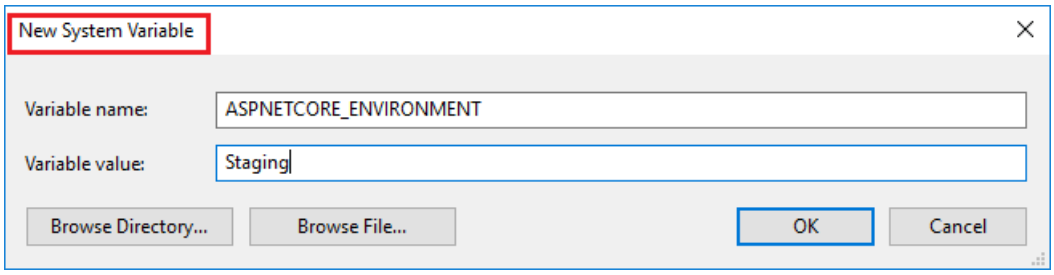
```
$Env:ASPNETCORE_ENVIRONMENT = "Staging"
dotnet run --no-launch-profile
```

The preceding command sets `ASPNETCORE_ENVIRONMENT` only for processes launched from that command window.

To set the value globally in Windows, use either of the following approaches:

- Open the **Control Panel** > **System** > **Advanced system settings** and add or edit the `ASPNETCORE_ENVIRONMENT` value:





- Open an administrative command prompt and use the `setx` command or open an administrative PowerShell command prompt and use `[Environment]::SetEnvironmentVariable`:

  Command prompt

  ```
  setx ASPNETCORE_ENVIRONMENT Staging /M
  ```

  The `/M` switch indicates to set the environment variable at the system level. If the `/M` switch isn't used, the environment variable is set for the user account.

  PowerShell

  ```
  [Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT", "Staging", "Machine")
  ```

  The `Machine` option value indicates to set the environment variable at the system level. If the option value is changed to `User`, the environment variable is set for the user account.

When the `ASPNETCORE_ENVIRONMENT` environment variable is set globally, it takes effect for `dotnet run` in any command window opened after the value is set. Environment values in *launchSettings.json* override values set in the system environment.

web.config

To set the `ASPNETCORE_ENVIRONMENT` environment variable with *web.config*, see the *Setting environment variables* section of ASP.NET Core Module.

Project file or publish profile

For Windows IIS deployments: Include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

Per IIS Application Pool

To set the `ASPNETCORE_ENVIRONMENT` environment variable for an app running in an isolated Application Pool (supported on IIS 10.0 or later), see the *AppCmd.exe command* section of the Environment Variables <environmentVariables> topic. When the `ASPNETCORE_ENVIRONMENT` environment variable is set for an app pool, its value overrides a setting at the system level.

When hosting an app in IIS and adding or changing the `ASPNETCORE_ENVIRONMENT` environment variable, use any one of the following approaches to have the new value picked up by apps:

- Execute `net stop was /y` followed by `net start w3svc` from a command prompt.
- Restart the server.

macOS

Setting the current environment for macOS can be performed in-line when running the app:

```
ASPNETCORE_ENVIRONMENT=Staging dotnet run
```

Alternatively, set the environment with `export` prior to running the app:

```
export ASPNETCORE_ENVIRONMENT=Staging
```

Machine-level environment variables are set in the *.bashrc* or *.bash_profile* file. Edit the file using any text editor. Add the following statement:

```
export ASPNETCORE_ENVIRONMENT=Staging
```

Linux

For Linux distributions, use the `export` command at a command prompt for session-based variable settings and *bash_profile* file for machine-level environment settings.

Set the environment in code

Call UseEnvironment when building the host. See .NET Generic Host.

Configuration by environment

To load configuration by environment, see Configuration in ASP.NET Core.

Environment-based Startup class and methods

Inject IWebHostEnvironment into the Startup class

Inject IWebHostEnvironment into the `Startup` constructor. This approach is useful when the app requires configuring `Startup` for only a few environments with minimal code differences per environment.

In the following example:

- The environment is held in the `_env` field.
- `_env` is used in `ConfigureServices` and `Configure` to apply startup configuration based on the app's environment.

```
public class Startup
{
    public Startup(IConfiguration configuration, IWebHostEnvironment env)
    {
        Configuration = configuration;
        _env = env;
    }

    public IConfiguration Configuration { get; }
    private readonly IWebHostEnvironment _env;

    public void ConfigureServices(IServiceCollection services)
    {
        if (_env.IsDevelopment())
        {
            Console.WriteLine(_env.EnvironmentName);
        }
        else if (_env.IsStaging())
        {
            Console.WriteLine(_env.EnvironmentName);
        }
        else
        {
            Console.WriteLine("Not dev or staging");
        }

        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {
        if (_env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

Startup class conventions

When an ASP.NET Core app starts, the Startup class bootstraps the app. The app can define multiple `Startup` classes for different environments. The appropriate `Startup` class is selected at runtime. The class whose name suffix matches the current environment is prioritized. If a matching `Startup{EnvironmentName}` class isn't found, the `Startup` class is used. This approach is useful when the app requires configuring startup for several environments with many code differences per environment. Typical apps will not need this approach.

To implement environment-based `Startup` classes, create a `Startup{EnvironmentName}` classes and a fallback

`Startup` class:

```
public class StartupDevelopment
{
    public StartupDevelopment(IConfiguration configuration)
    {
        Configuration = configuration;
        Console.WriteLine(MethodBase.GetCurrentMethod().DeclaringType.Name);
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseDeveloperExceptionPage();

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}

public class StartupProduction
{
    public StartupProduction(IConfiguration configuration)
    {
        Configuration = configuration;
        Console.WriteLine(MethodBase.GetCurrentMethod().DeclaringType.Name);
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {

        app.UseExceptionHandler("/Error");
        app.UseHsts();

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
```

```
}

public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
        Console.WriteLine(MethodBase.GetCurrentMethod().DeclaringType.Name);
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

Use the UseStartup(IWebHostBuilder, String) overload that accepts an assembly name:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        var assemblyName = typeof(Startup).GetTypeInfo().Assembly.FullName;

        return   Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup(assemblyName);
            });
    }
}
```

Startup method conventions

Configure and ConfigureServices support environment-specific versions of the form `Configure<EnvironmentName>` and `Configure<EnvironmentName>Services`. If a matching `Configure<EnvironmentName>Services` or `Configure<EnvironmentName>` method isn't found, the `ConfigureServices` or `Configure` method is used, respectively. This approach is useful when the app requires configuring startup for several environments with many code differences per environment:

```
public class Startup
{
    private void StartupConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void ConfigureDevelopmentServices(IServiceCollection services)
    {
        MyTrace.TraceMessage();
        StartupConfigureServices(services);
    }

    public void ConfigureStagingServices(IServiceCollection services)
    {
        MyTrace.TraceMessage();
        StartupConfigureServices(services);
    }

    public void ConfigureProductionServices(IServiceCollection services)
    {
        MyTrace.TraceMessage();
        StartupConfigureServices(services);
    }

    public void ConfigureServices(IServiceCollection services)
    {
        MyTrace.TraceMessage();
        StartupConfigureServices(services);
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        MyTrace.TraceMessage();

        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }

    public void ConfigureStaging(IApplicationBuilder app, IWebHostEnvironment env)
    {
```

```
            MyTrace.TraceMessage();

            app.UseExceptionHandler("/Error");
            app.UseHsts();

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
            });
        }
    }

    public static class MyTrace
    {
        public static void TraceMessage([CallerMemberName] string memberName = "")
        {
            Console.WriteLine($"Method: {memberName}");
        }
    }
```

Additional resources

- App startup in ASP.NET Core
- Configuration in ASP.NET Core
- ASP.NET Core Blazor environments

By Rick Anderson

ASP.NET Core configures app behavior based on the runtime environment using an environment variable.

View or download sample code (how to download)

Environments

ASP.NET Core reads the environment variable `ASPNETCORE_ENVIRONMENT` at app startup and stores the value in IHostingEnvironment.EnvironmentName. `ASPNETCORE_ENVIRONMENT` can be set to any value, but three values are provided by the framework:

- Development
- Staging
- Production (default)

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseMvc();
}
```

The preceding code:

- Calls UseDeveloperExceptionPage when `ASPNETCORE_ENVIRONMENT` is set to `Development`.
- Calls UseExceptionHandler when the value of `ASPNETCORE_ENVIRONMENT` is set one of the following:
  - `Staging`
  - `Production`
  - `Staging_2`

The Environment Tag Helper uses the value of `IHostingEnvironment.EnvironmentName` to include or exclude markup in the element:

```
<environment include="Development">
    <div>The effective tag is: &lt;environment include="Development"&gt;</div>
</environment>
<environment exclude="Development">
    <div>The effective tag is: &lt;environment exclude="Development"&gt;</div>
</environment>
<environment include="Staging,Development,Staging_2">
    <div>
        The effective tag is:
        &lt;environment include="Staging,Development,Staging_2"&gt;
    </div>
</environment>
```

On Windows and macOS, environment variables and values aren't case-sensitive. Linux environment variables and values are case-sensitive by default.

Development

The development environment can enable features that shouldn't be exposed in production. For example, the ASP.NET Core templates enable the Developer Exception Page in the development environment.

The environment for local machine development can be set in the *Properties\launchSettings.json* file of the project. Environment values set in *launchSettings.json* override values set in the system environment.

The following JSON shows three profiles from a *launchSettings.json* file:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:54339/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_My_Environment": "1",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_ENVIRONMENT": "Staging"
      }
    },
    "EnvironmentsSample": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging"
      },
      "applicationUrl": "http://localhost:54340/"
    },
    "Kestrel Staging": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_My_Environment": "1",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_ENVIRONMENT": "Staging"
      },
      "applicationUrl": "http://localhost:51997/"
    }
  }
}
```

NOTE

The `applicationUrl` property in *launchSettings.json* can specify a list of server URLs. Use a semicolon between the URLs in the list:

```
"EnvironmentsSample": {
   "commandName": "Project",
   "launchBrowser": true,
   "applicationUrl": "https://localhost:5001;http://localhost:5000",
   "environmentVariables": {
     "ASPNETCORE_ENVIRONMENT": "Development"
   }
}
```

When the app is launched with dotnet run, the first profile with `"commandName": "Project"` is used. The value of `commandName` specifies the web server to launch. `commandName` can be any one of the following:

- `IISExpress`
- `IIS`
- `Project` (which launches Kestrel)

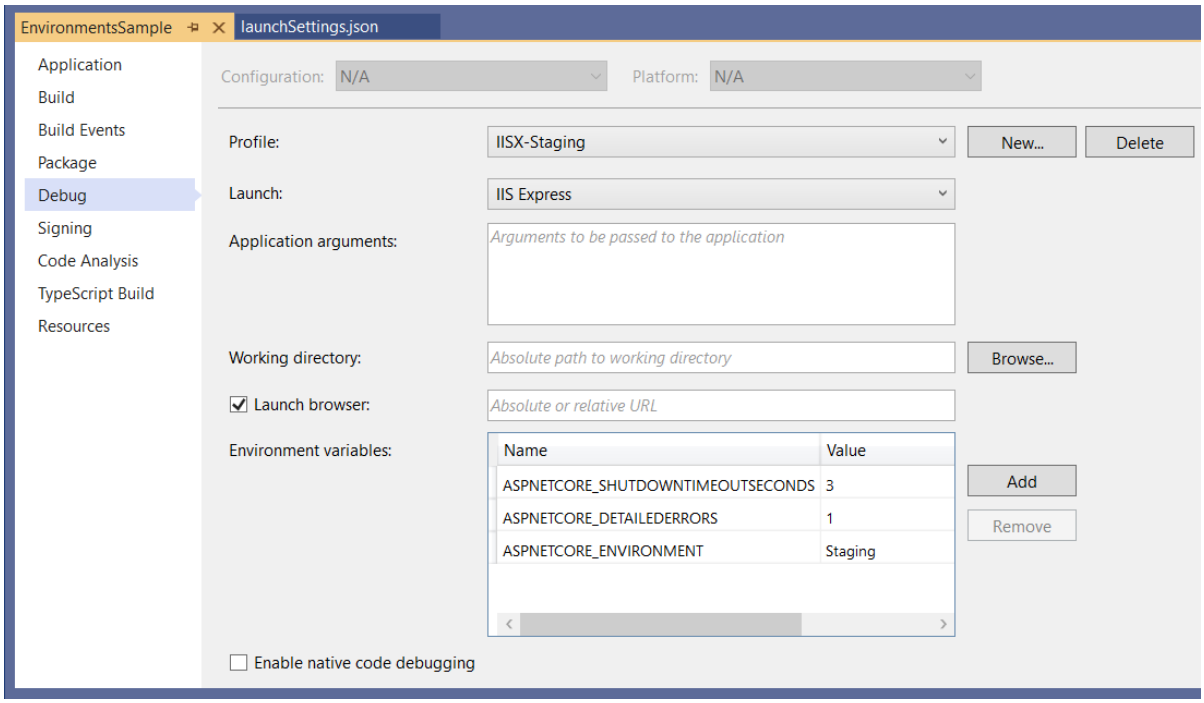
When an app is launched with dotnet run:

- *launchSettings.json* is read if available. `environmentVariables` settings in *launchSettings.json* override environment variables.
- The hosting environment is displayed.

The following output shows an app started with dotnet run:

```
PS C:\Websites\EnvironmentsSample> dotnet run
Using launch settings from C:\Websites\EnvironmentsSample\Properties\launchSettings.json...
Hosting environment: Staging
Content root path: C:\Websites\EnvironmentsSample
Now listening on: http://localhost:54340
Application started. Press Ctrl+C to shut down.
```

The Visual Studio project properties **Debug** tab provides a GUI to edit the *launchSettings.json* file:



Changes made to project profiles may not take effect until the web server is restarted. Kestrel must be restarted before it can detect changes made to its environment.

> **WARNING**
> *launchSettings.json* shouldn't store secrets. The Secret Manager tool can be used to store secrets for local development.

When using Visual Studio Code, environment variables can be set in the *.vscode/launch.json* file. The following example sets the environment to `Development`:

```
{
   "version": "0.2.0",
   "configurations": [
        {
            "name": ".NET Core Launch (web)",

            ... additional VS Code configuration settings ...

            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            }
        }
    ]
}
```

A *.vscode/launch.json* file in the project isn't read when starting the app with `dotnet run` in the same way as *Properties/launchSettings.json*. When launching an app in development that doesn't have a *launchSettings.json* file, either set the environment with an environment variable or a command-line argument to the `dotnet run` command.

Production

The production environment should be configured to maximize security, performance, and app robustness. Some common settings that differ from development include:

- Caching.
- Client-side resources are bundled, minified, and potentially served from a CDN.
- Diagnostic error pages disabled.
- Friendly error pages enabled.
- Production logging and monitoring enabled. For example, Application Insights.

Set the environment

It's often useful to set a specific environment for testing with an environment variable or platform setting. If the environment isn't set, it defaults to `Production`, which disables most debugging features. The method for setting the environment depends on the operating system.

When the host is built, the last environment setting read by the app determines the app's environment. The app's environment can't be changed while the app is running.

Environment variable or platform setting

Azure App Service

To set the environment in Azure App Service, perform the following steps:

1. Select the app from the **App Services** blade.
2. In the **Settings** group, select the **Configuration** blade.
3. In the **Application settings** tab, select **New application setting**.
4. In the **Add/Edit application setting** window, provide `ASPNETCORE_ENVIRONMENT` for the **Name**. For **Value**, provide the environment (for example, `Staging`).
5. Select the **Deployment slot setting** check box if you wish the environment setting to remain with the current slot when deployment slots are swapped. For more information, see Set up staging environments in Azure App Service in the Azure documentation.
6. Select **OK** to close the **Add/Edit application setting** window.
7. Select **Save** at the top of the **Configuration** blade.

Azure App Service automatically restarts the app after an app setting (environment variable) is added,

changed, or deleted in the Azure portal.

Windows

To set the `ASPNETCORE_ENVIRONMENT` for the current session when the app is started using dotnet run, the following commands are used:

Command prompt

```
set ASPNETCORE_ENVIRONMENT=Development
```

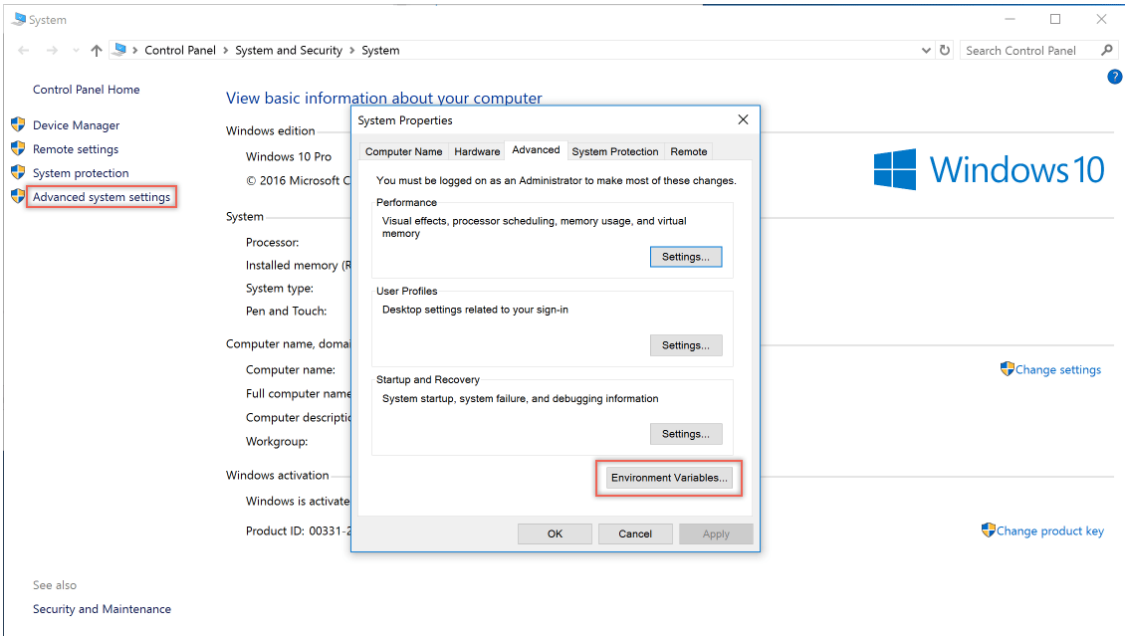
PowerShell

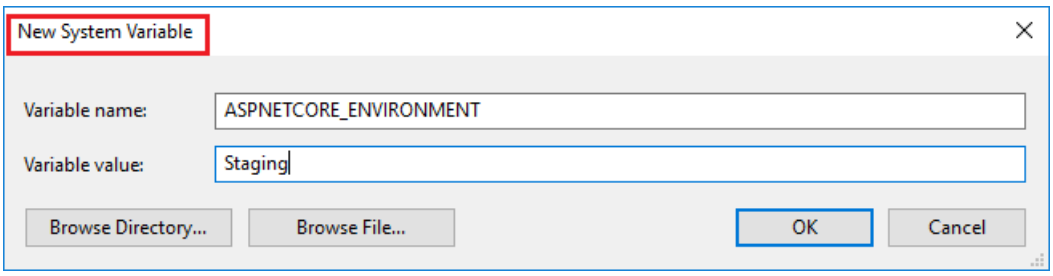
```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

These commands only take effect for the current window. When the window is closed, the `ASPNETCORE_ENVIRONMENT` setting reverts to the default setting or machine value.

To set the value globally in Windows, use either of the following approaches:

- Open the **Control Panel** > **System** > **Advanced system settings** and add or edit the `ASPNETCORE_ENVIRONMENT` value:





- Open an administrative command prompt and use the `setx` command or open an administrative PowerShell command prompt and use `[Environment]::SetEnvironmentVariable`:

  ### Command prompt

  ```
  setx ASPNETCORE_ENVIRONMENT Development /M
  ```

The `/M` switch indicates to set the environment variable at the system level. If the `/M` switch isn't used, the environment variable is set for the user account.

PowerShell

```
[Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT", "Development", "Machine")
```

The `Machine` option value indicates to set the environment variable at the system level. If the option value is changed to `User`, the environment variable is set for the user account.

When the `ASPNETCORE_ENVIRONMENT` environment variable is set globally, it takes effect for `dotnet run` in any command window opened after the value is set.

web.config

To set the `ASPNETCORE_ENVIRONMENT` environment variable with *web.config*, see the *Setting environment variables* section of ASP.NET Core Module.

Project file or publish profile

For Windows IIS deployments: Include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

Per IIS Application Pool

To set the `ASPNETCORE_ENVIRONMENT` environment variable for an app running in an isolated Application Pool (supported on IIS 10.0 or later), see the *AppCmd.exe command* section of the Environment Variables <environmentVariables> topic. When the `ASPNETCORE_ENVIRONMENT` environment variable is set for an app pool, its value overrides a setting at the system level.

> **IMPORTANT**
> When hosting an app in IIS and adding or changing the `ASPNETCORE_ENVIRONMENT` environment variable, use any one of the following approaches to have the new value picked up by apps:
>
> - Execute `net stop was /y` followed by `net start w3svc` from a command prompt.
> - Restart the server.

macOS

Setting the current environment for macOS can be performed in-line when running the app:

```
ASPNETCORE_ENVIRONMENT=Development dotnet run
```

Alternatively, set the environment with `export` prior to running the app:

```
export ASPNETCORE_ENVIRONMENT=Development
```

Machine-level environment variables are set in the *.bashrc* or *.bash_profile* file. Edit the file using any text editor. Add the following statement:

```
export ASPNETCORE_ENVIRONMENT=Development
```

Linux

For Linux distributions, use the `export` command at a command prompt for session-based variable settings and *bash_profile* file for machine-level environment settings.

Set the environment in code

Call UseEnvironment when building the host. See ASP.NET Core Web Host.

Configuration by environment

To load configuration by environment, we recommend:

- *appsettings* files (*appsettings.{Environment}.json*). See Configuration in ASP.NET Core.
- Environment variables (set on each system where the app is hosted). See ASP.NET Core Web Host and Safe storage of app secrets in development in ASP.NET Core.
- Secret Manager (in the Development environment only). See Safe storage of app secrets in development in ASP.NET Core.

Environment-based Startup class and methods

Inject IHostingEnvironment into Startup.Configure

Inject IHostingEnvironment into `Startup.Configure`. This approach is useful when the app only requires configuring `Startup.Configure` for only a few environments with minimal code differences per environment.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        // Development environment code
    }
    else
    {
        // Code for all other environments
    }
}
```

Inject IHostingEnvironment into the Startup class

Inject IHostingEnvironment into the `Startup` constructor and assign the service to a field for use throughout the `Startup` class. This approach is useful when the app requires configuring startup for only a few environments with minimal code differences per environment.

In the following example:

- The environment is held in the `_env` field.
- `_env` is used in `ConfigureServices` and `Configure` to apply startup configuration based on the app's environment.

```
public class Startup
{
    private readonly IHostingEnvironment _env;

    public Startup(IHostingEnvironment env)
    {
        _env = env;
    }

    public void ConfigureServices(IServiceCollection services)
    {
        if (_env.IsDevelopment())
        {
            // Development environment code
        }
        else if (_env.IsStaging())
        {
            // Staging environment code
        }
        else
        {
            // Code for all other environments
        }
    }

    public void Configure(IApplicationBuilder app)
    {
        if (_env.IsDevelopment())
        {
            // Development environment code
        }
        else
        {
            // Code for all other environments
        }
    }
}
```

Startup class conventions

When an ASP.NET Core app starts, the Startup class bootstraps the app. The app can define separate `Startup` classes for different environments (for example, `StartupDevelopment`). The appropriate `Startup` class is selected at runtime. The class whose name suffix matches the current environment is prioritized. If a matching `Startup{EnvironmentName}` class isn't found, the `Startup` class is used. This approach is useful when the app requires configuring startup for several environments with many code differences per environment.

To implement environment-based `Startup` classes, create a `Startup{EnvironmentName}` class for each environment in use and a fallback `Startup` class:

```
// Startup class to use in the Development environment
public class StartupDevelopment
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
    }
}

// Startup class to use in the Production environment
public class StartupProduction
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
    }
}

// Fallback Startup class
// Selected if the environment doesn't match a Startup{EnvironmentName} class
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
    }
}
```

Use the UseStartup(IWebHostBuilder, String) overload that accepts an assembly name:

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args)
{
    var assemblyName = typeof(Startup).GetTypeInfo().Assembly.FullName;

    return WebHost.CreateDefaultBuilder(args)
        .UseStartup(assemblyName);
}
```

Startup method conventions

Configure and ConfigureServices support environment-specific versions of the form `Configure<EnvironmentName>` and `Configure<EnvironmentName>Services`. If a matching `Configure<EnvironmentName>Services` or `Configure<EnvironmentName>` method isn't found, the `ConfigureServices` or `Configure` method is used, respectively. This approach is useful when the app requires configuring startup for several environments with many code differences per environment:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        StartupConfigureServices(services);
    }

    public void ConfigureStagingServices(IServiceCollection services)
    {
        StartupConfigureServices(services);
    }

    private void StartupConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
        {
            app.UseExceptionHandler("/Error");
        }

        app.UseStaticFiles();
        app.UseMvc();
    }

    public void ConfigureStaging(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (!env.IsStaging())
        {
            throw new Exception("Not staging.");
        }

        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseMvc();
    }
}
```

Additional resources

- App startup in ASP.NET Core
- Configuration in ASP.NET Core

Logging in .NET Core and ASP.NET Core

9/22/2020 • 57 minutes to read • Edit Online

By Kirk Larkin, Juergen Gutsch and Rick Anderson

.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. This article shows how to use the logging API with built-in providers.

Most of the code examples shown in this article are from ASP.NET Core apps. The logging-specific parts of these code snippets apply to any .NET Core app that uses the Generic Host. The ASP.NET Core web app templates use the Generic Host.

View or download sample code (how to download)

Logging providers

Logging providers store logs, except for the `Console` provider which displays logs. For example, the Azure Application Insights provider stores logs in Azure Application Insights. Multiple providers can be enabled.

The default ASP.NET Core web app templates:

- Use the Generic Host.
- Call CreateDefaultBuilder, which adds the following logging providers:
  - Console
  - Debug
  - EventSource
  - EventLog: Windows only

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The preceding code shows the `Program` class created with the ASP.NET Core web app templates. The next several sections provide samples based on the ASP.NET Core web app templates, which use the Generic Host. Non-host console apps are discussed later in this document.

To override the default set of logging providers added by `Host.CreateDefaultBuilder`, call `ClearProviders` and add the required logging providers. For example, the following code:

- Calls ClearProviders to remove all the ILoggerProvider instances from the builder.
- Adds the Console logging provider.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

For additional providers, see:

- Built-in logging providers
- Third-party logging providers.

Create logs

To create logs, use an ILogger<TCategoryName> object from dependency injection (DI).

The following example:

- Creates a logger, `ILogger<AboutModel>`, which uses a log *category* of the fully qualified name of the type `AboutModel`. The log category is a string that is associated with each log.
- Calls LogInformation to log at the `Information` level. The Log *level* indicates the severity of the logged event.

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
    public string Message { get; set; }

    public void OnGet()
    {
        Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
        _logger.LogInformation(Message);
    }
}
```

Levels and categories are explained in more detail later in this document.

For information on Blazor, see Create logs in Blazor and Blazor WebAssembly in this document.

Create logs in Main and Startup shows how to create logs in `Main` and `Startup`.

Configure logging

Logging configuration is commonly provided by the `Logging` section of *appsettings*.`{Environment}`*.json* files. The following *appsettings.Development.json* file is generated by the ASP.NET Core web app templates:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

In the preceding JSON:

- The `"Default"`, `"Microsoft"`, and `"Microsoft.Hosting.Lifetime"` categories are specified.
- The `"Microsoft"` category applies to all categories that start with `"Microsoft"`. For example, this setting applies to the `"Microsoft.AspNetCore.Routing.EndpointMiddleware"` category.
- The `"Microsoft"` category logs at log level `Warning` and higher.
- The `"Microsoft.Hosting.Lifetime"` category is more specific than the `"Microsoft"` category, so the `"Microsoft.Hosting.Lifetime"` category logs at log level "Information" and higher.
- A specific log provider is not specified, so `LogLevel` applies to all the enabled logging providers except for the Windows EventLog.

The `Logging` property can have LogLevel and log provider properties. The `LogLevel` specifies the minimum level to log for selected categories. In the preceding JSON, `Information` and `Warning` log levels are specified. `LogLevel` indicates the severity of the log and ranges from 0 to 6:

`Trace` = 0, `Debug` = 1, `Information` = 2, `Warning` = 3, `Error` = 4, `Critical` = 5, and `None` = 6.

When a `LogLevel` is specified, logging is enabled for messages at the specified level and higher. In the preceding JSON, the `Default` category is logged for `Information` and higher. For example, `Information`, `Warning`, `Error`, and `Critical` messages are logged. If no `LogLevel` is specified, logging defaults to the `Information` level. For more information, see Log levels.

A provider property can specify a `LogLevel` property. `LogLevel` under a provider specifies levels to log for that provider, and overrides the non-provider log settings. Consider the following *appsettings.json* file:

```
{
  "Logging": {
    "LogLevel": { // All providers, LogLevel applies to all the enabled providers.
      "Default": "Error", // Default logging, Error and higher.
      "Microsoft": "Warning" // All Microsoft* categories, Warning and higher.
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information", // Overrides preceding LogLevel:Default setting.
        "Microsoft.Hosting": "Trace" // Debug:Microsoft.Hosting category.
      }
    },
    "EventSource": { // EventSource provider
      "LogLevel": {
        "Default": "Warning" // All categories of EventSource provider.
      }
    }
  }
}
```

Settings in `Logging.{providername}.LogLevel` override settings in `Logging.LogLevel`. In the preceding JSON, the `Debug` provider's default log level is set to `Information`:

`Logging:Debug:LogLevel:Default:Information`

The preceding setting specifies the `Information` log level for every `Logging:Debug:` category except `Microsoft.Hosting`. When a specific category is listed, the specific category overrides the default category. In the preceding JSON, the `Logging:Debug:LogLevel` categories `"Microsoft.Hosting"` and `"Default"` override the settings in `Logging:LogLevel`

The minimum log level can be specified for any of:

- Specific providers: For example, `Logging:EventSource:LogLevel:Default:Information`
- Specific categories: For example, `Logging:LogLevel:Microsoft:Warning`
- All providers and all categories: `Logging:LogLevel:Default:Warning`

Any logs below the minimum level are ***not***:

- Passed to the provider.
- Logged or displayed.

To suppress all logs, specify LogLevel.None. `LogLevel.None` has a value of 6, which is higher than `LogLevel.Critical` (5).

If a provider supports log scopes, `IncludeScopes` indicates whether they're enabled. For more information, see log scopes

The following *appsettings.json* file contains all the providers enabled by default:

```
{
  "Logging": {
    "LogLevel": { // No provider, LogLevel applies to all the enabled providers.
      "Default": "Error",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Warning"
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information" // Overrides preceding LogLevel:Default setting.
      }
    },
    "Console": {
      "IncludeScopes": true,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "EventSource": {
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "EventLog": {
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "AzureAppServicesFile": {
      "IncludeScopes": true,
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "AzureAppServicesBlob": {
      "IncludeScopes": true,
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "ApplicationInsights": {
      "LogLevel": {
        "Default": "Information"
      }
    }
  }
}
```

In the preceding sample:

- The categories and levels are not suggested values. The sample is provided to show all the default providers.
- Settings in `Logging.{providername}.LogLevel` override settings in `Logging.LogLevel`. For example, the level in `Debug.LogLevel.Default` overrides the level in `LogLevel.Default`.
- Each default provider *alias* is used. Each provider defines an *alias* that can be used in configuration in place of the fully qualified type name. The built-in providers aliases are:
  - Console
  - Debug
  - EventSource
  - EventLog

- AzureAppServicesFile
- AzureAppServicesBlob
- ApplicationInsights

Set log level by command line, environment variables, and other configuration

Log level can be set by any of the configuration providers.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

The following commands:

- Set the environment key `Logging:LogLevel:Microsoft` to a value of `Information` on Windows.
- Test the settings when using an app created with the ASP.NET Core web application templates. The `dotnet run` command must be run in the project directory after using `set`.

```
set Logging__LogLevel__Microsoft=Information
dotnet run
```

The preceding environment setting:

- Is only set in processes launched from the command window they were set in.
- Isn't read by browsers launched with Visual Studio.

The following setx command also sets the environment key and value on Windows. Unlike `set`, `setx` settings are persisted. The `/M` switch sets the variable in the system environment. If `/M` isn't used, a user environment variable is set.

```
setx Logging__LogLevel__Microsoft=Information /M
```

On Azure App Service, select **New application setting** on the **Settings** > **Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

For more information, see Azure Apps: Override app configuration using the Azure Portal.

For more information on setting ASP.NET Core configuration values using environment variables, see environment variables. For information on using other configuration sources, including the command line, Azure Key Vault, Azure App Configuration, other file formats, and more, see Configuration in ASP.NET Core.

How filtering rules are applied

When an ILogger<TCategoryName> object is created, the ILoggerFactory object selects a single rule per provider to apply to that logger. All messages written by an `ILogger` instance are filtered based on the selected rules. The most specific rule for each provider and category pair is selected from the available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If no match is found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If no match is found, select all rules that don't specify a category.
- If multiple rules are selected, take the **last** one.
- If no rules are selected, use `MinimumLevel`.

Logging output from dotnet run and Visual Studio

Logs created with the default logging providers are displayed:

- In Visual Studio
  - In the Debug output window when debugging.
  - In the ASP.NET Core Web Server window.
- In the console window when the app is run with `dotnet run`.

Logs that begin with "Microsoft" categories are from ASP.NET Core framework code. ASP.NET Core and application code use the same logging API and providers.

Log category

When an `ILogger` object is created, a *category* is specified. That category is included with each log message created by that instance of `ILogger`. The category string is arbitrary, but the convention is to use the class name. For example, in a controller the name might be `"TodoApi.Controllers.TodoController"`. The ASP.NET Core web apps use `ILogger<T>` to automatically get an `ILogger` instance that uses the fully qualified type name of `T` as the category:

```
public class PrivacyModel : PageModel
{
    private readonly ILogger<PrivacyModel> _logger;

    public PrivacyModel(ILogger<PrivacyModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.PrivacyModel called.");
    }
}
```

To explicitly specify the category, call `ILoggerFactory.CreateLogger`:

```
public class ContactModel : PageModel
{
    private readonly ILogger _logger;

    public ContactModel(ILoggerFactory logger)
    {
        _logger = logger.CreateLogger("MyCategory");
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.ContactModel called.");
    }
```

Calling `CreateLogger` with a fixed name can be useful when used in multiple methods so the events can be

organized by category.

`ILogger<T>` is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

The following table lists the LogLevel values, the convenience `Log{LogLevel}` extension method, and the suggested usage:

LOGLEVEL	VALUE	METHOD	DESCRIPTION
Trace	0	LogTrace	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should ***not*** be enabled in production.
Debug	1	LogDebug	For debugging and development. Use with caution in production due to the high volume.
Information	2	LogInformation	Tracks the general flow of the app. May have long-term value.
Warning	3	LogWarning	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	LogError	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.
Critical	5	LogCritical	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category should not write any messages.

In the previous table, the `LogLevel` is listed from lowest to highest severity.

The Log method's first parameter, LogLevel, indicates the severity of the log. Rather than calling `Log(LogLevel, ...)`, most developers call the Log{LogLevel} extension methods. The `Log{LogLevel}` extension methods call the Log method and specify the LogLevel. For example, the following two logging calls are functionally equivalent and produce the same log:

```
[HttpGet]
public IActionResult Test1(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);

    _logger.Log(LogLevel.Information, MyLogEvents.TestItem, routeInfo);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    return ControllerContext.MyDisplayRouteInfo();
}
```

`MyLogEvents.TestItem` is the event ID. `MyLogEvents` is part of the sample app and is displayed in the Log event ID section.

MyDisplayRouteInfo and ToCtxString are provided by the Rick.Docs.Samples.RouteInfo NuGet package. The methods display `Controller` route information.

The following code creates `Information` and `Warning` logs:

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

In the preceding code, the first `Log{LogLevel}` parameter, `MyLogEvents.GetItem`, is the Log event ID. The second parameter is a message template with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in the message template section later in this document.

Call the appropriate `Log{LogLevel}` method to control how much log output is written to a particular storage medium. For example:

- In production:
  - Logging at the `Trace` or `Information` levels produces a high-volume of detailed log messages. To control costs and not exceed data storage limits, log `Trace` and `Information` level messages to a high-volume, low-cost data store. Consider limiting `Trace` and `Information` to specific categories.
  - Logging at `Warning` through `Critical` levels should produce few log messages.
    - Costs and storage limits usually aren't a concern.
    - Few logs allow more flexibility in data store choices.
- In development:
  - Set to `Warning`.
  - Add `Trace` or `Information` messages when troubleshooting. To limit output, set `Trace` or `Information` only for the categories under investigation.

ASP.NET Core writes logs for framework events. For example, consider the log output for:

- A Razor Pages app created with the ASP.NET Core templates.

- Logging set to `Logging:Console:LogLevel:Microsoft:Information`
- Navigation to the Privacy page:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 GET https://localhost:5001/Privacy
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint '/Privacy'
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
      Route matched with {page = "/Privacy"}. Executing page /Privacy
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
      Executing handler method DefaultRP.Pages.PrivacyModel.OnGet - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
      Executed handler method OnGet, returned result .
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[103]
      Executing an implicit handler method - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[104]
      Executed an implicit handler method, returned result Microsoft.AspNetCore.Mvc.RazorPages.PageResult.
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
      Executed page /Privacy in 74.5188ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint '/Privacy'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 149.3023ms 200 text/html; charset=utf-8
```

The following JSON sets `Logging:Console:LogLevel:Microsoft:Information`:

```
{
  "Logging": {      // Default, all providers.
    "LogLevel": {
      "Microsoft": "Warning"
    },
    "Console": { // Console provider.
      "LogLevel": {
        "Microsoft": "Information"
      }
    }
  }
}
```

Log event ID

Each log can specify an *event ID*. The sample app uses the `MyLogEvents` class to define event IDs:

```
public class MyLogEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems     = 1001;
    public const int GetItem       = 1002;
    public const int InsertItem    = 1003;
    public const int UpdateItem    = 1004;
    public const int DeleteItem    = 1005;

    public const int TestItem      = 3000;

    public const int GetItemNotFound    = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

An event ID associates a set of events. For example, all logs related to displaying a list of items on a page might be 1001.

The logging provider may store the event ID in an ID field, in the logging message, or not at all. The Debug provider doesn't show event IDs. The console provider shows event IDs in brackets after the category:

```
info: TodoApi.Controllers.TodoItemsController[1002]
      Getting item 1
warn: TodoApi.Controllers.TodoItemsController[4000]
      Get(1) NOT FOUND
```

Some logging providers store the event ID in a field, which allows for filtering on the ID.

Log message template

Each log API uses a message template. The message template can contain placeholders for which arguments are provided. Use names for the placeholders, not numbers.

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

The order of placeholders, not their names, determines which parameters are used to provide their values. In the following code, the parameter names are out of sequence in the message template:

```
string p1 = "param1";
string p2 = "param2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

The preceding code creates a log message with the parameter values in sequence:

```
Parameter values: param1, param2
```

This approach allows logging providers to implement semantic or structured logging. The arguments themselves are passed to the logging system, not just the formatted message template. This enables logging providers to store the parameter values as fields. For example, consider the following logger method:

```
_logger.LogInformation("Getting item {Id} at {RequestTime}", id, DateTime.Now);
```

For example, when logging to Azure Table Storage:

- Each Azure Table entity can have `ID` and `RequestTime` properties.
- Tables with properties simplify queries on logged data. For example, a query can find all logs within a particular `RequestTime` range without having to parse the time out of the text message.

Log exceptions

The logger methods have overloads that take an exception parameter:

```
[HttpGet("{id}")]
public IActionResult TestExp(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    try
    {
        if (id == 3)
        {
            throw new Exception("Test exception");
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, ex, "TestExp({Id})", id);
        return NotFound();
    }

    return ControllerContext.MyDisplayRouteInfo();
}
```

MyDisplayRouteInfo and ToCtxString are provided by the Rick.Docs.Samples.RouteInfo NuGet package. The methods display `Controller` route information.

Exception logging is provider-specific.

Default log level

If the default log level is not set, the default log level value is `Information`.

For example, consider the following web app:

- Created with the ASP.NET web app templates.
- *appsettings.json* and *appsettings.Development.json* deleted or renamed.

With the preceding setup, navigating to the privacy or home page produces many `Trace`, `Debug`, and `Information` messages with `Microsoft` in the category name.

The following code sets the default log level when the default log level is not set in configuration:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning))
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Generally, log levels should be specified in configuration and not code.

Filter function

A filter function is invoked for all providers and categories that don't have rules assigned to them by configuration or code:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
            {
                logging.AddFilter((provider, category, logLevel) =>
                {
                    if (provider.Contains("ConsoleLoggerProvider")
                        && category.Contains("Controller")
                        && logLevel >= LogLevel.Information)
                    {
                        return true;
                    }
                    else if (provider.Contains("ConsoleLoggerProvider")
                        && category.Contains("Microsoft")
                        && logLevel >= LogLevel.Information)
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                });
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The preceding code displays console logs when the category contains `Controller` or `Microsoft` and the log level is `Information` or higher.

Generally, log levels should be specified in configuration and not code.

ASP.NET Core and EF Core categories

The following table contains some categories used by ASP.NET Core and Entity Framework Core, with notes about the logs:

CATEGORY	NOTES
Microsoft.AspNetCore	General ASP.NET Core diagnostics.
Microsoft.AspNetCore.DataProtection	Which keys were considered, found, and used.
Microsoft.AspNetCore.HostFiltering	Hosts allowed.
Microsoft.AspNetCore.Hosting	How long HTTP requests took to complete and what time they started. Which hosting startup assemblies were loaded.
Microsoft.AspNetCore.Mvc	MVC and Razor diagnostics. Model binding, filter execution, view compilation, action selection.
Microsoft.AspNetCore.Routing	Route matching information.
Microsoft.AspNetCore.Server	Connection start, stop, and keep alive responses. HTTPS certificate information.
Microsoft.AspNetCore.StaticFiles	Files served.
Microsoft.EntityFrameworkCore	General Entity Framework Core diagnostics. Database activity and configuration, change detection, migrations.

To view more categories in the console window, set **appsettings.Development.json** to the following:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Trace",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Log scopes

A *scope* can group a set of logical operations. This grouping can be used to attach the same data to each log that's created as part of a set. For example, every log created as part of processing a transaction can include the transaction ID.

A scope:

- Is an IDisposable type that's returned by the BeginScope method.
- Lasts until it's disposed.

The following providers support scopes:

- `Console`
- AzureAppServicesFile and AzureAppServicesBlob

Use a scope by wrapping logger calls in a `using` block:

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    TodoItem todoItem;

    using (_logger.BeginScope("using block message"))
    {
        _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

        todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            _logger.LogWarning(MyLogEvents.GetItemNotFound,
                "Get({Id}) NOT FOUND", id);
            return NotFound();
        }
    }

    return ItemToDTO(todoItem);
}
```

The following JSON enables scopes for the console provider:

```
{
  "Logging": {
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "IncludeScopes": true, // Required to use Scopes.
      "LogLevel": {
        "Microsoft": "Warning",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

The following code enables scopes for the console provider:

```
public class Scopes
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging((hostingContext, logging) =>
            {
                logging.ClearProviders();
                logging.AddConsole(options => options.IncludeScopes = true);
                logging.AddDebug();
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Generally, logging should be specified in configuration and not code.

Built-in logging providers

ASP.NET Core includes the following logging providers as part of the shared framework:

- Console
- Debug
- EventSource
- EventLog

The following logging providers are shipped by Microsoft, but not as part of the shared framework. They must be installed as additional nuget.

- AzureAppServicesFile and AzureAppServicesBlob
- ApplicationInsights

For information on `stdout` and debug logging with the ASP.NET Core Module, see Troubleshoot ASP.NET Core on Azure App Service and IIS and ASP.NET Core Module.

Console

The `Console` provider logs output to the console. For more information on viewing `Console` logs in development, see Logging output from dotnet run and Visual Studio.

Debug

The `Debug` provider writes log output by using the System.Diagnostics.Debug class. Calls to `System.Diagnostics.Debug.WriteLine` write to the `Debug` provider.

On Linux, the `Debug` provider log location is distribution-dependent and may be one of the following:

- */var/log/message*
- */var/log/syslog*

Event Source

The `EventSource` provider writes to a cross-platform event source with the name `Microsoft-Extensions-Logging`. On Windows, the provider uses ETW.

dotnet trace tooling

The dotnet-trace tool is a cross-platform CLI global tool that enables the collection of .NET Core traces of a running process. The tool collects Microsoft.Extensions.Logging.EventSource provider data using a LoggingEventSource.

See dotnet-trace for installation instructions.

Use the dotnet trace tooling to collect a trace from an app:

1. Run the app with the `dotnet run` command.
2. Determine the process identifier (PID) of the .NET Core app:
   - On Windows, use one of the following approaches:
     - Task Manager (Ctrl+Alt+Del)
     - tasklist command
     - Get-Process Powershell command
   - On Linux, use the pidof command.

   Find the PID for the process that has the same name as the app's assembly.
3. Execute the `dotnet trace` command.

   General command syntax:

```
dotnet trace collect -p {PID}
    --providers Microsoft-Extensions-Logging:{Keyword}:{Provider Level}
        :FilterSpecs=\"
            {Logger Category 1}:{Category Level 1};
            {Logger Category 2}:{Category Level 2};
            ...
            {Logger Category N}:{Category Level N}\"
```

   When using a PowerShell command shell, enclose the `--providers` value in single quotes (`'`):

```
dotnet trace collect -p {PID}
    --providers 'Microsoft-Extensions-Logging:{Keyword}:{Provider Level}
        :FilterSpecs=\"
            {Logger Category 1}:{Category Level 1};
            {Logger Category 2}:{Category Level 2};
            ...
            {Logger Category N}:{Category Level N}\"'
```

   On non-Windows platforms, add the `-f speedscope` option to change the format of the output trace file to `speedscope`.

   The following table defines the Keyword:

KEYWORD	DESCRIPTION
1	Log meta events about the `LoggingEventSource`. Doesn't log events from `ILogger`.
2	Turns on the `Message` event when `ILogger.Log()` is called. Provides information in a programmatic (not formatted) way.
4	Turns on the `FormatMessage` event when `ILogger.Log()` is called. Provides the formatted string version of the information.

KEYWORD	DESCRIPTION
8	Turns on the `MessageJson` event when `ILogger.Log()` is called. Provides a JSON representation of the arguments.

The following table lists the provider levels:

PROVIDER LEVEL	DESCRIPTION
0	`LogAlways`
1	`Critical`
2	`Error`
3	`Warning`
4	`Informational`
5	`Verbose`

The parsing for a category level can be either a string or a number:

CATEGORY NAMED VALUE	NUMERIC VALUE
`Trace`	0
`Debug`	1
`Information`	2
`Warning`	3
`Error`	4
`Critical`	5

The provider level and category level:

- Are in reverse order.
- The string constants aren't all identical.

If no `FilterSpecs` are specified then the `EventSourceLogger` implementation attempts to convert the provider level to a category level and applies it to all categories.

PROVIDER LEVEL	CATEGORY LEVEL
`Verbose` (5)	`Debug` (1)
`Informational` (4)	`Information` (2)
`Warning` (3)	`Warning` (3)

PROVIDER LEVEL	CATEGORY LEVEL
`Error` (2)	`Error` (4)
`Critical` (1)	`Critical` (5)

If `FilterSpecs` are provided, any category that is included in the list uses the category level encoded there, all other categories are filtered out.

The following examples assume:

- An app is running and calling `logger.LogDebug("12345")`.
- The process ID (PID) has been set via `set PID=12345`, where `12345` is the actual PID.

Consider the following command:

```
dotnet trace collect -p %PID% --providers Microsoft-Extensions-Logging:4:5
```

The preceding command:

- Captures debug messages.
- Doesn't apply a `FilterSpecs`.
- Specifies level 5 which maps category Debug.

Consider the following command:

```
dotnet trace collect -p %PID%  --providers Microsoft-Extensions-Logging:4:5:\"FilterSpecs=*:5\"
```

The preceding command:

- Doesn't capture debug messages because the category level 5 is `Critical`.
- Provides a `FilterSpecs`.

The following command captures debug messages because category level 1 specifies `Debug`.

```
dotnet trace collect -p %PID%  --providers Microsoft-Extensions-Logging:4:5:\"FilterSpecs=*:1\"
```

The following command captures debug messages because category specifies `Debug`.

```
dotnet trace collect -p %PID%  --providers Microsoft-Extensions-Logging:4:5:\"FilterSpecs=*:Debug\"
```

`FilterSpecs` entries for `{Logger Category}` and `{Category Level}` represent additional log filtering conditions. Separate `FilterSpecs` entries with the `;` semicolon character.

Example using a Windows command shell:

```
dotnet trace collect -p %PID% --providers Microsoft-Extensions-
Logging:4:2:FilterSpecs=\"Microsoft.AspNetCore.Hosting*:4\"
```

The preceding command activates:

- The Event Source logger to produce formatted strings (`4`) for errors (`2`).
- `Microsoft.AspNetCore.Hosting` logging at the `Informational` logging level (`4`).

4. Stop the dotnet trace tooling by pressing the Enter key or Ctrl+C.

The trace is saved with the name *trace.nettrace* in the folder where the `dotnet trace` command is executed.

5. Open the trace with Perfview. Open the *trace.nettrace* file and explore the trace events.

If the app doesn't build the host with `CreateDefaultBuilder`, add the Event Source provider to the app's logging configuration.

For more information, see:

- Trace for performance analysis utility (dotnet-trace) (.NET Core documentation)
- Trace for performance analysis utility (dotnet-trace) (dotnet/diagnostics GitHub repository documentation)
- LoggingEventSource Class (.NET API Browser)
- EventLevel
- LoggingEventSource reference source (3.0): To obtain reference source for a different version, change the branch to `release/{Version}`, where `{Version}` is the version of ASP.NET Core desired.
- Perfview: Useful for viewing Event Source traces.

Perfview

Use the PerfView utility to collect and view logs. There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET Core.

To configure PerfView for collecting events logged by this provider, add the string `*Microsoft-Extensions-Logging` to the **Additional Providers** list. Don't miss the `*` at the start of the string.

Windows EventLog

The `EventLog` provider sends log output to the Windows Event Log. Unlike the other providers, the `EventLog` provider does ***not*** inherit the default non-provider settings. If `EventLog` log settings aren't specified, they default to LogLevel.Warning.

To log events lower than LogLevel.Warning, explicitly set the log level. The following example sets the Event Log default log level to LogLevel.Information:

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

AddEventLog overloads can pass in EventLogSettings. If `null` or not specified, the following default settings are used:

- `LogName`: "Application"
- `SourceName`: ".NET Runtime"
- `MachineName`: The local machine name is used.

The following code changes the `SourceName` from the default value of `".NET Runtime"` to `MyLogs`:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
            {
                logging.AddEventLog(eventLogSettings =>
                {
                    eventLogSettings.SourceName = "MyLogs";
                });
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Azure App Service

The Microsoft.Extensions.Logging.AzureAppServices provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account.

The provider package isn't included in the shared framework. To use the provider, add the provider package to the project.

To configure provider settings, use AzureFileLoggerOptions and AzureBlobLoggerOptions, as shown in the following example:

```
public class Scopes
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureLogging(logging => logging.AddAzureWebAppDiagnostics())
                .ConfigureServices(serviceCollection => serviceCollection
                    .Configure<AzureFileLoggerOptions>(options =>
                    {
                        options.FileName = "azure-diagnostics-";
                        options.FileSizeLimit = 50 * 1024;
                        options.RetainedFileCountLimit = 5;
                    })
                    .Configure<AzureBlobLoggerOptions>(options =>
                    {
                        options.BlobName = "log.txt";
                    }))
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

When deployed to Azure App Service, the app uses the settings in the App Service logs section of the **App Service**

page of the Azure portal. When the following settings are updated, the changes take effect immediately without requiring a restart or redeployment of the app.

- **Application Logging (Filesystem)**
- **Application Logging (Blob)**

The default location for log files is in the *D:\home\LogFiles\Application* folder, and the default file name is *diagnostics-yyyymmdd.txt*. The default file size limit is 10 MB, and the default maximum number of files retained is 2. The default blob name is *{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt*.

This provider only logs when the project runs in the Azure environment.

Azure log streaming

Azure log streaming supports viewing log activity in real time from:

- The app server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the **App Service logs** page from the app's portal page.
- Set **Application Logging (Filesystem)** to **On**.
- Choose the log **Level**. This setting only applies to Azure log streaming.

Navigate to the **Log Stream** page to view logs. The logged messages are logged with the `ILogger` interface.

Azure Application Insights

The Microsoft.Extensions.Logging.ApplicationInsights provider package writes logs to Azure Application Insights. Application Insights is a service that monitors a web app and provides tools for querying and analyzing the telemetry data. If you use this provider, you can query and analyze your logs by using the Application Insights tools.

The logging provider is included as a dependency of Microsoft.ApplicationInsights.AspNetCore, which is the package that provides all available telemetry for ASP.NET Core. If you use this package, you don't have to install the provider package.

The Microsoft.ApplicationInsights.Web package is for ASP.NET 4.x, not ASP.NET Core.

For more information, see the following resources:

- Application Insights overview
- Application Insights for ASP.NET Core applications - Start here if you want to implement the full range of Application Insights telemetry along with logging.
- ApplicationInsightsLoggerProvider for .NET Core ILogger logs - Start here if you want to implement the logging provider without the rest of Application Insights telemetry.
- Application Insights logging adapters.
- Install, configure, and initialize the Application Insights SDK - Interactive tutorial on the Microsoft Learn site.

Third-party logging providers

Third-party logging frameworks that work with ASP.NET Core:

- elmah.io (GitHub repo)
- Gelf (GitHub repo)
- JSNLog (GitHub repo)
- KissLog.net (GitHub repo)

- Log4Net (GitHub repo)
- Loggr (GitHub repo)
- NLog (GitHub repo)
- PLogger (GitHub repo)
- Sentry (GitHub repo)
- Serilog (GitHub repo)
- Stackdriver (Github repo)

Some third-party frameworks can perform semantic logging, also known as structured logging.

Using a third-party framework is similar to using one of the built-in providers:

1. Add a NuGet package to your project.
2. Call an `ILoggerFactory` extension method provided by the logging framework.

For more information, see each provider's documentation. Third-party logging providers aren't supported by Microsoft.

Non-host console app

For an example of how to use the Generic Host in a non-web console app, see the *Program.cs* file of the Background Tasks sample app (Background tasks with hosted services in ASP.NET Core).

Logging code for apps without Generic Host differs in the way providers are added and loggers are created.

Logging providers

In a non-host console app, call the provider's `Add{provider name}` extension method while creating a `LoggerFactory`:

```
class Program
{
    static void Main(string[] args)
    {
        using var loggerFactory = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter("Microsoft", LogLevel.Warning)
                .AddFilter("System", LogLevel.Warning)
                .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
                .AddConsole()
                .AddEventLog();
        });
        ILogger logger = loggerFactory.CreateLogger<Program>();
        logger.LogInformation("Example log message");
    }
}
```

Create logs

To create logs, use an ILogger<TCategoryName> object. Use the `LoggerFactory` to create an `ILogger`.

The following example creates a logger with `LoggingConsoleApp.Program` as the category.

```
class Program
{
    static void Main(string[] args)
    {
        using var loggerFactory = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter("Microsoft", LogLevel.Warning)
                .AddFilter("System", LogLevel.Warning)
                .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
                .AddConsole()
                .AddEventLog();
        });
        ILogger logger = loggerFactory.CreateLogger<Program>();
        logger.LogInformation("Example log message");
    }
}
```

In the following example, the logger is used to create logs with `Information` as the level. The Log *level* indicates the severity of the logged event.

```
class Program
{
    static void Main(string[] args)
    {
        using var loggerFactory = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter("Microsoft", LogLevel.Warning)
                .AddFilter("System", LogLevel.Warning)
                .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
                .AddConsole()
                .AddEventLog();
        });
        ILogger logger = loggerFactory.CreateLogger<Program>();
        logger.LogInformation("Example log message");
    }
}
```

Levels and categories are explained in more detail in this document.

Log during host construction

Logging during host construction isn't directly supported. However, a separate logger can be used. In the following example, a Serilog logger is used to log in `CreateHostBuilder`. `AddSerilog` uses the static configuration specified in `Log.Logger`:

```
using System;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        var builtConfig = new ConfigurationBuilder()
            .AddJsonFile("appsettings.json")
            .AddCommandLine(args)
            .Build();

        Log.Logger = new LoggerConfiguration()
            .WriteTo.Console()
            .WriteTo.File(builtConfig["Logging:FilePath"])
            .CreateLogger();

        try
        {
            return Host.CreateDefaultBuilder(args)
                .ConfigureServices((context, services) =>
                {
                    services.AddRazorPages();
                })
                .ConfigureAppConfiguration((hostingContext, config) =>
                {
                    config.AddConfiguration(builtConfig);
                })
                .ConfigureLogging(logging =>
                {
                    logging.AddSerilog();
                })
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Host builder error");

            throw;
        }
        finally
        {
            Log.CloseAndFlush();
        }
    }
}
```

Configure a service that depends on ILogger

Constructor injection of a logger into `Startup` works in earlier versions of ASP.NET Core because a separate DI container is created for the Web Host. For information about why only one container is created for the Generic Host, see the breaking change announcement.

To configure a service that depends on `ILogger<T>`, use constructor injection or provide a factory method. The factory method approach is recommended only if there is no other option. For example, consider a service that needs an `ILogger<T>` instance provided by DI:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddRazorPages();

    services.AddSingleton<IMyService>((container) =>
    {
        var logger = container.GetRequiredService<ILogger<MyService>>();
        return new MyService() { Logger = logger };
    });
}
```

The preceding highlighted code is a Func that runs the first time the DI container needs to construct an instance of `MyService`. You can access any of the registered services in this way.

Create logs in Main

The following code logs in `Main` by getting an `ILogger` instance from DI after building the host:

```
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Host created.");

    host.Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

Create logs in Startup

The following code writes logs in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
                      ILogger<Startup> logger)
{
    if (env.IsDevelopment())
    {
        logger.LogInformation("In Development.");
        app.UseDeveloperExceptionPage();
    }
    else
    {
        logger.LogInformation("Not Development.");
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapRazorPages();
    });
}
```

Writing logs before completion of the DI container setup in the `Startup.ConfigureServices` method is not supported:

- Logger injection into the `Startup` constructor is not supported.
- Logger injection into the `Startup.ConfigureServices` method signature is not supported

The reason for this restriction is that logging depends on DI and on configuration, which in turns depends on DI. The DI container isn't set up until `ConfigureServices` finishes.

For information on configuring a service that depends on `ILogger<T>` or why constructor injection of a logger into `Startup` worked in earlier versions, see Configure a service that depends on ILogger

No asynchronous logger methods

Logging should be so fast that it isn't worth the performance cost of asynchronous code. If a logging data store is slow, don't write to it directly. Consider writing the log messages to a fast store initially, then moving them to the slow store later. For example, when logging to SQL Server, don't do so directly in a `Log` method, since the `Log` methods are synchronous. Instead, synchronously add log messages to an in-memory queue and have a background worker pull the messages out of the queue to do the asynchronous work of pushing data to SQL Server. For more information, see this GitHub issue.

Change log levels in a running app

The Logging API doesn't include a scenario to change log levels while an app is running. However, some configuration providers are capable of reloading configuration, which takes immediate effect on logging configuration. For example, the File Configuration Provider, reloads logging configuration by default. If configuration is changed in code while an app is running, the app can call IConfigurationRoot.Reload to update the app's logging configuration.

ILogger and ILoggerFactory

The ILogger<TCategoryName> and ILoggerFactory interfaces and implementations are included in the .NET Core SDK. They are also available in the following NuGet packages:

- The interfaces are in Microsoft.Extensions.Logging.Abstractions.
- The default implementations are in Microsoft.Extensions.Logging.

Apply log filter rules in code

The preferred approach for setting log filter rules is by using Configuration.

The following example shows how to register filter rules in code:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
               logging.AddFilter("System", LogLevel.Debug)
                  .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)
                  .AddFilter<ConsoleLoggerProvider>("Microsoft", LogLevel.Trace))
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

`logging.AddFilter("System", LogLevel.Debug)` specifies the `System` category and log level `Debug`. The filter is applied to all providers because a specific provider was not configured.

`AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)` specifies:

- The `Debug` logging provider.
- Log level `Information` and higher.
- All categories starting with `"Microsoft"`.

Create a custom logger

To add a custom logger, add an ILoggerProvider with ILoggerFactory:

```
public void Configure(
    IApplicationBuilder app,
    IWebHostEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory.AddProvider(new CustomLoggerProvider(new CustomLoggerConfiguration()));
```

The `ILoggerProvider` creates one or more `ILogger` instances. The `ILogger` instances are used by the framework to log the information.

Sample custom logger configuration

The sample:

- Is designed to be a very basic sample that sets the color of the log console by event ID and log level. Loggers generally don't change by event ID and are not specific to log level.

- Creates different color console entries per log level and event ID using the following configuration type:

```
public class ColorConsoleLoggerConfiguration
{
    public LogLevel LogLevel { get; set; } = LogLevel.Warning;
    public int EventId { get; set; } = 0;
    public ConsoleColor Color { get; set; } = ConsoleColor.Yellow;
}
```

The preceding code sets the default level to `Warning` and the color to `Yellow`. If the `EventId` is set to 0, we will log all events.

Create the custom logger

The `ILogger` implementation category name is typically the logging source. For example, the type where the logger is created:

```
public class ColorConsoleLogger : ILogger
{
    private readonly string _name;
    private readonly ColorConsoleLoggerConfiguration _config;

    public ColorConsoleLogger(string name, ColorConsoleLoggerConfiguration config)
    {
        _name = name;
        _config = config;
    }

    public IDisposable BeginScope<TState>(TState state)
    {
        return null;
    }

    public bool IsEnabled(LogLevel logLevel)
    {
        return logLevel == _config.LogLevel;
    }

    public void Log<TState>(LogLevel logLevel, EventId eventId, TState state,
                        Exception exception, Func<TState, Exception, string> formatter)
    {
        if (!IsEnabled(logLevel))
        {
            return;
        }

        if (_config.EventId == 0 || _config.EventId == eventId.Id)
        {
            var color = Console.ForegroundColor;
            Console.ForegroundColor = _config.Color;
            Console.WriteLine($"{logLevel} - {eventId.Id} " +
                              $"- {_name} - {formatter(state, exception)}");
            Console.ForegroundColor = color;
        }
    }
}
```

The preceding code:

- Creates a logger instance per category name.
- Checks `logLevel == _config.LogLevel` in `IsEnabled`, so each `logLevel` has a unique logger. Generally, loggers should also be enabled for all higher log levels:

```
public bool IsEnabled(LogLevel logLevel)
{
    return logLevel >= _config.LogLevel;
}
```

Create the custom LoggerProvider

The `LoggerProvider` is the class that creates the logger instances. Maybe it is not needed to create a logger instance per category, but this makes sense for some Loggers, like NLog or log4net. Doing this you are also able to choose different logging output targets per category if needed:

```
public class ColorConsoleLoggerProvider : ILoggerProvider
{
    private readonly ColorConsoleLoggerConfiguration _config;
    private readonly ConcurrentDictionary<string, ColorConsoleLogger> _loggers = new
ConcurrentDictionary<string, ColorConsoleLogger>();

    public ColorConsoleLoggerProvider(ColorConsoleLoggerConfiguration config)
    {
        _config = config;
    }

    public ILogger CreateLogger(string categoryName)
    {
        return _loggers.GetOrAdd(categoryName, name => new ColorConsoleLogger(name, _config));
    }

    public void Dispose()
    {
        _loggers.Clear();
    }
}
```

In the preceding code, CreateLogger creates a single instance of the `ColorConsoleLogger` per category name and stores it in the `ConcurrentDictionary<TKey,TValue>`;

Usage and registration of the custom logger

Register the logger in the `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
                      ILoggerFactory loggerFactory)
{
    // Default registration.
    loggerFactory.AddProvider(new ColorConsoleLoggerProvider(
                              new ColorConsoleLoggerConfiguration
    {
        LogLevel = LogLevel.Error,
        Color = ConsoleColor.Red
    }));

    // Custom registration with default values.
    loggerFactory.AddColorConsoleLogger();

    // Custom registration with a new configuration instance.
    loggerFactory.AddColorConsoleLogger(new ColorConsoleLoggerConfiguration
    {
        LogLevel = LogLevel.Debug,
        Color = ConsoleColor.Gray
    });

    // Custom registration with a configuration object.
    loggerFactory.AddColorConsoleLogger(c =>
    {
        c.LogLevel = LogLevel.Information;
        c.Color = ConsoleColor.Blue;
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

For the preceding code, provide at least one extension method for the `ILoggerFactory`:

```
public static class ColorConsoleLoggerExtensions
{
    public static ILoggerFactory AddColorConsoleLogger(
                                    this ILoggerFactory loggerFactory,
                                    ColorConsoleLoggerConfiguration config)
    {
        loggerFactory.AddProvider(new ColorConsoleLoggerProvider(config));
        return loggerFactory;
    }
    public static ILoggerFactory AddColorConsoleLogger(
                                    this ILoggerFactory loggerFactory)
    {
        var config = new ColorConsoleLoggerConfiguration();
        return loggerFactory.AddColorConsoleLogger(config);
    }
    public static ILoggerFactory AddColorConsoleLogger(
                                  this ILoggerFactory loggerFactory,
                                  Action<ColorConsoleLoggerConfiguration> configure)
    {
        var config = new ColorConsoleLoggerConfiguration();
        configure(config);
        return loggerFactory.AddColorConsoleLogger(config);
    }
}
```

Additional resources

- High-performance logging with LoggerMessage in ASP.NET Core
- Logging bugs should be created in the github.com/dotnet/runtime/ repo.
- ASP.NET Core Blazor logging

By Tom Dykstra and Steve Smith

.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. This article shows how to use the logging API with built-in providers.

View or download sample code (how to download)

Add providers

A logging provider displays or stores logs. For example, the Console provider displays logs on the console, and the Azure Application Insights provider stores them in Azure Application Insights. Logs can be sent to multiple destinations by adding multiple providers.

To add a provider, call the provider's `Add{provider name}` extension method in *Program.cs*:

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                      optional: true, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            // Requires `using Microsoft.Extensions.Logging;`
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
            logging.AddEventSourceLogger();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

The preceding code requires references to `Microsoft.Extensions.Logging` and `Microsoft.Extensions.Configuration`.

The default project template calls CreateDefaultBuilder, which adds the following logging providers:

- Console
- Debug
- EventSource (starting in ASP.NET Core 2.2)

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

If you use `CreateDefaultBuilder`, you can replace the default providers with your own choices. Call ClearProviders, and add the providers you want.

```
public static void Main(string[] args)
{
    var host = CreateWebHostBuilder(args).Build();

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        });
```

Learn more about built-in logging providers and third-party logging providers later in the article.

Create logs

To create logs, use an ILogger<TCategoryName> object. In a web app or hosted service, get an `ILogger` from dependency injection (DI). In non-host console apps, use the `LoggerFactory` to create an `ILogger`.

The following ASP.NET Core example creates a logger with `TodoApiSample.Pages.AboutModel` as the category. The log *category* is a string that is associated with each log. The `ILogger<T>` instance provided by DI creates logs that have the fully qualified name of type `T` as the category.

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
```

In the following ASP.NET Core and console app examples, the logger is used to create logs with `Information` as the level. The Log *level* indicates the severity of the logged event.

```
public void OnGet()
{
    Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
    _logger.LogInformation("Message displayed: {Message}", Message);
}
```

Levels and categories are explained in more detail later in this article.

Create logs in Startup

To write logs in the `Startup` class, include an `ILogger` parameter in the constructor signature:

```
public class Startup
{
    private readonly ILogger _logger;

    public Startup(IConfiguration configuration, ILogger<Startup> logger)
    {
        Configuration = configuration;
        _logger = logger;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

        // Add our repository type
        services.AddSingleton<ITodoRepository, TodoRepository>();
        _logger.LogInformation("Added TodoRepository to services");
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            _logger.LogInformation("In Development environment");
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseMvc();
    }
}
```

Create logs in the Program class

To write logs in the `Program` class, get an `ILogger` instance from DI:

```
public static void Main(string[] args)
{
    var host = CreateWebHostBuilder(args).Build();

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        });
```

Logging during host construction isn't directly supported. However, a separate logger can be used. In the following example, a Serilog logger is used to log in `CreateWebHostBuilder`. `AddSerilog` uses the static configuration specified in `Log.Logger`:

```
using System;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var builtConfig = new ConfigurationBuilder()
            .AddJsonFile("appsettings.json")
            .AddCommandLine(args)
            .Build();

        Log.Logger = new LoggerConfiguration()
            .WriteTo.Console()
            .WriteTo.File(builtConfig["Logging:FilePath"])
            .CreateLogger();

        try
        {
            return WebHost.CreateDefaultBuilder(args)
                .ConfigureServices((context, services) =>
                {
                    services.AddMvc();
                })
                .ConfigureAppConfiguration((hostingContext, config) =>
                {
                    config.AddConfiguration(builtConfig);
                })
                .ConfigureLogging(logging =>
                {
                    logging.AddSerilog();
                })
                .UseStartup<Startup>();
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Host builder error");

            throw;
        }
        finally
        {
            Log.CloseAndFlush();
        }
    }
}
```

No asynchronous logger methods

Logging should be so fast that it isn't worth the performance cost of asynchronous code. If your logging data store is slow, don't write to it directly. Consider writing the log messages to a fast store initially, then move them to the slow store later. For example, if you're logging to SQL Server, you don't want to do that directly in a `Log` method, since the `Log` methods are synchronous. Instead, synchronously add log messages to an in-memory queue and have a background worker pull the messages out of the queue to do the asynchronous work of pushing data to SQL Server. For more information, see this GitHub issue.

Configuration

Logging provider configuration is provided by one or more configuration providers:

- File formats (INI, JSON, and XML).
- Command-line arguments.
- Environment variables.
- In-memory .NET objects.
- The unencrypted Secret Manager storage.
- An encrypted user store, such as Azure Key Vault.
- Custom providers (installed or created).

For example, logging configuration is commonly provided by the `Logging` section of app settings files. The following example shows the contents of a typical *appsettings.Development.json* file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    },
    "Console":
    {
      "IncludeScopes": true
    }
  }
}
```

The `Logging` property can have `LogLevel` and log provider properties (Console is shown).

The `LogLevel` property under `Logging` specifies the minimum level to log for selected categories. In the example, `System` and `Microsoft` categories log at `Information` level, and all others log at `Debug` level.

Other properties under `Logging` specify logging providers. The example is for the Console provider. If a provider supports log scopes, `IncludeScopes` indicates whether they're enabled. A provider property (such as `Console` in the example) may also specify a `LogLevel` property. `LogLevel` under a provider specifies levels to log for that provider.

If levels are specified in `Logging.{providername}.LogLevel`, they override anything set in `Logging.LogLevel`. For example, consider the following JSON:

```
{
  "Logging": {      // Default, all providers.
    "LogLevel": {
      "Microsoft": "Warning"
    },
    "Console": { // Console provider.
      "LogLevel": {
        "Microsoft": "Information"
      }
    }
  }
}
```

In the preceding JSON, the `Console` provider settings overrides the preceding (default) log level.

The Logging API doesn't include a scenario to change log levels while an app is running. However, some configuration providers are capable of reloading configuration, which takes immediate effect on logging

configuration. For example, the File Configuration Provider, which is added by `CreateDefaultBuilder` to read settings files, reloads logging configuration by default. If configuration is changed in code while an app is running, the app can call IConfigurationRoot.Reload to update the app's logging configuration.

For information on implementing configuration providers, see Configuration in ASP.NET Core.

Sample logging output

With the sample code shown in the preceding section, logs appear in the console when the app is run from the command line. Here's an example of console output:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/0
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 42.9286ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 148.889ms 404
```

The preceding logs were generated by making an HTTP Get request to the sample app at `http://localhost:5000/api/todo/0`.

Here's an example of the same logs as they appear in the Debug window when you run the sample app in Visual Studio:

```
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request starting HTTP/1.1 GET
http://localhost:53104/api/todo/0
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executing action method
TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) - ModelState is Valid
TodoApi.Controllers.TodoController:Information: Getting item 0
TodoApi.Controllers.TodoController:Warning: GetById(0) NOT FOUND
Microsoft.AspNetCore.Mvc.StatusCodeResult:Information: Executing HttpStatusCodeResult, setting HTTP status code
404
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executed action
TodoApi.Controllers.TodoController.GetById (TodoApi) in 152.5657ms
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request finished in 316.3195ms 404
```

The logs that are created by the `ILogger` calls shown in the preceding section begin with "TodoApi". The logs that begin with "Microsoft" categories are from ASP.NET Core framework code. ASP.NET Core and application code are using the same logging API and providers.

The remainder of this article explains some details and options for logging.

NuGet packages

The `ILogger` and `ILoggerFactory` interfaces are in Microsoft.Extensions.Logging.Abstractions, and default implementations for them are in Microsoft.Extensions.Logging.

Log category

When an `ILogger` object is created, a *category* is specified for it. That category is included with each log message

created by that instance of `ILogger`. The category may be any string, but the convention is to use the class name, such as "TodoApi.Controllers.TodoController".

Use `ILogger<T>` to get an `ILogger` instance that uses the fully qualified type name of `T` as the category:

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

To explicitly specify the category, call `ILoggerFactory.CreateLogger`:

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApiSample.Controllers.TodoController");
    }
```

`ILogger<T>` is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

Every log specifies a LogLevel value. The log level indicates the severity or importance. For example, you might write an `Information` log when a method ends normally and a `Warning` log when a method returns a *404 Not Found* status code.

The following code creates `Information` and `Warning` logs:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

In the preceding code, the `MyLogEvents.GetItem` and `MyLogEvents.GetItemNotFound` parameters are the Log event ID. The second parameter is a message template with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in the Log message template section in this article.

Log methods that include the level in the method name (for example, `LogInformation` and `LogWarning`) are extension methods for ILogger. These methods call a `Log` method that takes a `LogLevel` parameter. You can call

the `Log` method directly rather than one of these extension methods, but the syntax is relatively complicated. For more information, see ILogger and the logger extensions source code.

ASP.NET Core defines the following log levels, ordered here from lowest to highest severity.

- Trace = 0

  For information that's typically valuable only for debugging. These messages may contain sensitive application data and so shouldn't be enabled in a production environment. *Disabled by default.*

- Debug = 1

  For information that may be useful in development and debugging. Example: `Entering method Configure with flag set to true.` Enable `Debug` level logs in production only when troubleshooting, due to the high volume of logs.

- Information = 2

  For tracking the general flow of the app. These logs typically have some long-term value. Example: `Request received for path /api/todo`

- Warning = 3

  For abnormal or unexpected events in the app flow. These may include errors or other conditions that don't cause the app to stop but might need to be investigated. Handled exceptions are a common place to use the `Warning` log level. Example: `FileNotFoundException for file quotes.txt.`

- Error = 4

  For errors and exceptions that cannot be handled. These messages indicate a failure in the current activity or operation (such as the current HTTP request), not an app-wide failure. Example log message: `Cannot insert record due to duplicate key violation.`

- Critical = 5

  For failures that require immediate attention. Examples: data loss scenarios, out of disk space.

Use the log level to control how much log output is written to a particular storage medium or display window. For example:

- In production:
  - Logging at the `Trace` through `Information` levels produces a high-volume of detailed log messages. To control costs and not exceed data storage limits, log `Trace` through `Information` level messages to a high-volume, low-cost data store.
  - Logging at `Warning` through `Critical` levels typically produces fewer, smaller log messages. Therefore, costs and storage limits usually aren't a concern, which results in greater flexibility of data store choice.
- During development:
  - Log `Warning` through `Critical` messages to the console.
  - Add `Trace` through `Information` messages when troubleshooting.

The Log filtering section later in this article explains how to control which log levels a provider handles.

ASP.NET Core writes logs for framework events. The log examples earlier in this article excluded logs below `Information` level, so no `Debug` or `Trace` level logs were created. Here's an example of console logs produced by running the sample app configured to show `Debug` logs:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:62555/api/todo/0
dbug: Microsoft.AspNetCore.Routing.Tree.TreeRouter[1]
      Request successfully matched the route with name 'GetTodo' and template 'api/Todo/{id}'.
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Update (TodoApi)' with id '089d59b6-92ec-472d-b552-
cc613dfd625d' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Delete (TodoApi)' with id 'f3476abe-4bd9-4ad3-9261-
3ead09607366' did not match the constraint 'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action TodoApi.Controllers.TodoController.GetById (TodoApi)
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) -
ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action method TodoApi.Controllers.TodoController.GetById (TodoApi), returned result
Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 0.8788ms
dbug: Microsoft.AspNetCore.Server.Kestrel[9]
      Connection id "0HL6L7NEFF2QD" completed keep alive response.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2.7286ms 404
```

Log event ID

Each log can specify an *event ID*. The sample app does this by using a locally defined `LoggingEvents` class:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

An event ID associates a set of events. For example, all logs related to displaying a list of items on a page might be

1001.

The logging provider may store the event ID in an ID field, in the logging message, or not at all. The Debug provider doesn't show event IDs. The console provider shows event IDs in brackets after the category:

```
info: TodoApi.Controllers.TodoController[1002]
      Getting item invalidid
warn: TodoApi.Controllers.TodoController[4000]
      GetById(invalidid) NOT FOUND
```

Log message template

Each log specifies a message template. The message template can contain placeholders for which arguments are provided. Use names for the placeholders, not numbers.

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

The order of placeholders, not their names, determines which parameters are used to provide their values. In the following code, notice that the parameter names are out of sequence in the message template:

```
string p1 = "parm1";
string p2 = "parm2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

This code creates a log message with the parameter values in sequence:

```
Parameter values: parm1, parm2
```

The logging framework works this way so that logging providers can implement semantic logging, also known as structured logging. The arguments themselves are passed to the logging system, not just the formatted message template. This information enables logging providers to store the parameter values as fields. For example, suppose logger method calls look like this:

```
_logger.LogInformation("Getting item {Id} at {RequestTime}", id, DateTime.Now);
```

If you're sending the logs to Azure Table Storage, each Azure Table entity can have `ID` and `RequestTime` properties, which simplifies queries on log data. A query can find all logs within a particular `RequestTime` range without parsing the time out of the text message.

Logging exceptions

The logger methods have overloads that let you pass in an exception, as in the following example:

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({Id}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```

Different providers handle the exception information in different ways. Here's an example of Debug provider output from the code shown above.

```
TodoApiSample.Controllers.TodoController: Warning: GetById(55) NOT FOUND

System.Exception: Item not found exception.
   at TodoApiSample.Controllers.TodoController.GetById(String id) in
C:\TodoApiSample\Controllers\TodoController.cs:line 226
```

Log filtering

You can specify a minimum log level for a specific provider and category or for all providers or all categories. Any logs below the minimum level aren't passed to that provider, so they don't get displayed or stored.

To suppress all logs, specify `LogLevel.None` as the minimum log level. The integer value of `LogLevel.None` is 6, which is higher than `LogLevel.Critical` (5).

Create filter rules in configuration

The project template code calls `CreateDefaultBuilder` to set up logging for the Console, Debug, and EventSource (ASP.NET Core 2.2 or later) providers. The `CreateDefaultBuilder` method sets up logging to look for configuration in a `Logging` section, as explained earlier in this article.

The configuration data specifies minimum log levels by provider and category, as in the following example:

```
{
  "Logging": {
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "IncludeScopes": false,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

This JSON creates six filter rules: one for the Debug provider, four for the Console provider, and one for all providers. A single rule is chosen for each provider when an `ILogger` object is created.

Filter rules in code

The following example shows how to register filter rules in code:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("System", LogLevel.Debug)
               .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace));
```

The second `AddFilter` specifies the Debug provider by using its type name. The first `AddFilter` applies to all providers because it doesn't specify a provider type.

How filtering rules are applied

The configuration data and the `AddFilter` code shown in the preceding examples create the rules shown in the following table. The first six come from the configuration example and the last two come from the code example.

NUMBER	PROVIDER	CATEGORIES THAT BEGIN WITH ...	MINIMUM LOG LEVEL
1	Debug	All categories	Information
2	Console	Microsoft.AspNetCore.Mvc.Razor.Internal	Warning
3	Console	Microsoft.AspNetCore.Mvc.Razor.Razor	Debug
4	Console	Microsoft.AspNetCore.Mvc.Razor	Error
5	Console	All categories	Information
6	All providers	All categories	Debug
7	All providers	System	Debug
8	Debug	Microsoft	Trace

When an `ILogger` object is created, the `ILoggerFactory` object selects a single rule per provider to apply to that logger. All messages written by an `ILogger` instance are filtered based on the selected rules. The most specific rule possible for each provider and category pair is selected from the available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If no match is found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If no match is found, select all rules that don't specify a category.
- If multiple rules are selected, take the **last** one.
- If no rules are selected, use `MinimumLevel`.

With the preceding list of rules, suppose you create an `ILogger` object for category "Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine":

- For the Debug provider, rules 1, 6, and 8 apply. Rule 8 is most specific, so that's the one selected.
- For the Console provider, rules 3, 4, 5, and 6 apply. Rule 3 is most specific.

The resulting `ILogger` instance sends logs of `Trace` level and above to the Debug provider. Logs of `Debug` level and above are sent to the Console provider.

Provider aliases

Each provider defines an *alias* that can be used in configuration in place of the fully qualified type name. For the built-in providers, use the following aliases:

- Console
- Debug
- EventSource
- EventLog
- TraceSource
- AzureAppServicesFile
- AzureAppServicesBlob
- ApplicationInsights

Default minimum level

There's a minimum level setting that takes effect only if no rules from configuration or code apply for a given provider and category. The following example shows how to set the minimum level:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning));
```

If you don't explicitly set the minimum level, the default value is `Information`, which means that `Trace` and `Debug` logs are ignored.

Filter functions

A filter function is invoked for all providers and categories that don't have rules assigned to them by configuration or code. Code in the function has access to the provider type, category, and log level. For example:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logBuilder =>
    {
        logBuilder.AddFilter((provider, category, logLevel) =>
        {
            if (provider == "Microsoft.Extensions.Logging.Console.ConsoleLoggerProvider" &&
                category == "TodoApiSample.Controllers.TodoController")
            {
                return false;
            }
            return true;
        });
    });
```

System categories and levels

Here are some categories used by ASP.NET Core and Entity Framework Core, with notes about what logs to expect from them:

CATEGORY	NOTES
Microsoft.AspNetCore	General ASP.NET Core diagnostics.
Microsoft.AspNetCore.DataProtection	Which keys were considered, found, and used.

CATEGORY	NOTES
Microsoft.AspNetCore.HostFiltering	Hosts allowed.
Microsoft.AspNetCore.Hosting	How long HTTP requests took to complete and what time they started. Which hosting startup assemblies were loaded.
Microsoft.AspNetCore.Mvc	MVC and Razor diagnostics. Model binding, filter execution, view compilation, action selection.
Microsoft.AspNetCore.Routing	Route matching information.
Microsoft.AspNetCore.Server	Connection start, stop, and keep alive responses. HTTPS certificate information.
Microsoft.AspNetCore.StaticFiles	Files served.
Microsoft.EntityFrameworkCore	General Entity Framework Core diagnostics. Database activity and configuration, change detection, migrations.

Log scopes

A *scope* can group a set of logical operations. This grouping can be used to attach the same data to each log that's created as part of a set. For example, every log created as part of processing a transaction can include the transaction ID.

A scope is an `IDisposable` type that's returned by the BeginScope method and lasts until it's disposed. Use a scope by wrapping logger calls in a `using` block:

```
public IActionResult GetById(string id)
{
    TodoItem item;
    using (_logger.BeginScope("Message attached to logs created in the using block"))
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
        item = _todoRepository.Find(id);
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
            return NotFound();
        }
    }
    return new ObjectResult(item);
}
```

The following code enables scopes for the console provider:

Program.cs:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})
```

> **NOTE**
>
> Configuring the `IncludeScopes` console logger option is required to enable scope-based logging.
>
> For information on configuration, see the Configuration section.

Each log message includes the scoped information:

```
info: TodoApiSample.Controllers.TodoController[1002]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApiSample.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      Getting item 0
warn: TodoApiSample.Controllers.TodoController[4000]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 => TodoApiSample.Controllers.TodoController.GetById
(TodoApi) => Message attached to logs created in the using block
      GetById(0) NOT FOUND
```

Built-in logging providers

ASP.NET Core ships the following providers:

- Console
- Debug
- EventSource
- EventLog
- TraceSource
- AzureAppServicesFile
- AzureAppServicesBlob
- ApplicationInsights

For information on stdout and debug logging with the ASP.NET Core Module, see Troubleshoot ASP.NET Core on Azure App Service and IIS and ASP.NET Core Module.

Console provider

The Microsoft.Extensions.Logging.Console provider package sends log output to the console.

```
logging.AddConsole();
```

To see console logging output, open a command prompt in the project folder and run the following command:

```
dotnet run
```

Debug provider

The Microsoft.Extensions.Logging.Debug provider package writes log output by using the System.Diagnostics.Debug class (`Debug.WriteLine` method calls).

On Linux, this provider writes logs to */var/log/message*.

```
logging.AddDebug();
```

Event Source provider

The Microsoft.Extensions.Logging.EventSource provider package writes to an Event Source cross-platform with the

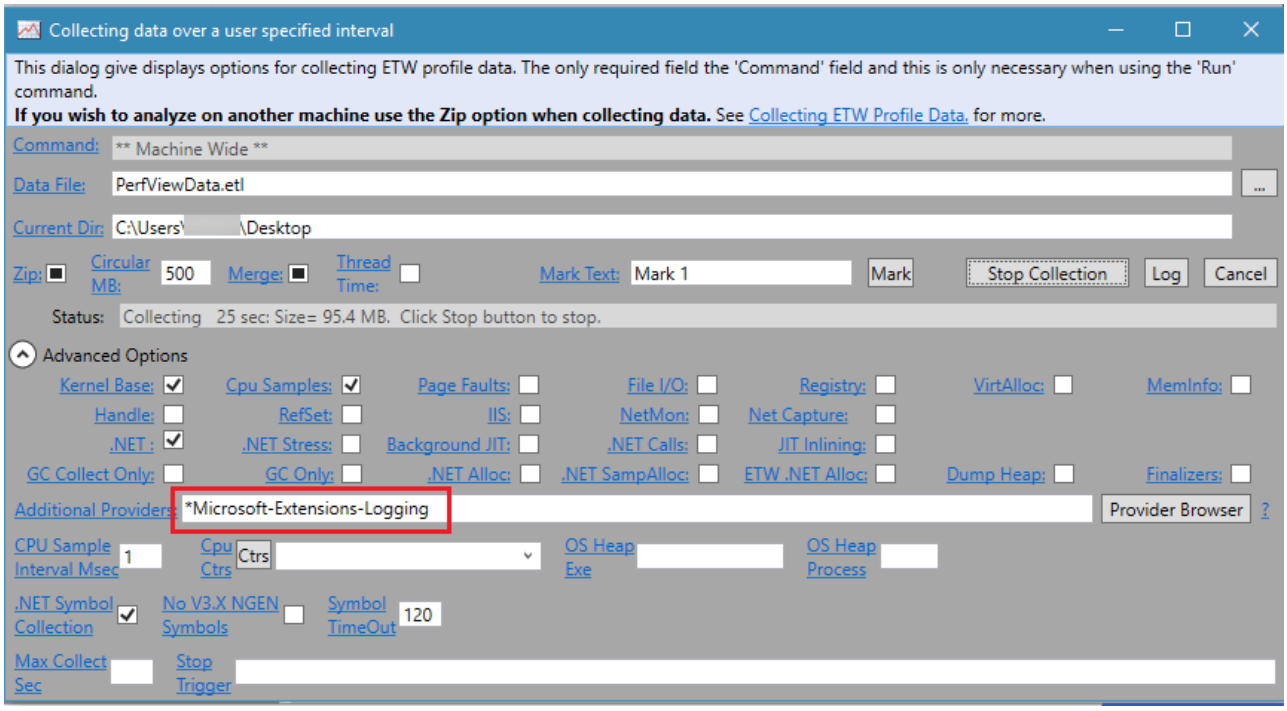
name `Microsoft-Extensions-Logging`. On Windows, the provider uses ETW.

```
logging.AddEventSourceLogger();
```

The Event Source provider is added automatically when `CreateDefaultBuilder` is called to build the host.

Use the PerfView utility to collect and view logs. There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET Core.

To configure PerfView for collecting events logged by this provider, add the string `*Microsoft-Extensions-Logging` to the **Additional Providers** list. (Don't miss the asterisk at the start of the string.)



Windows EventLog provider

The Microsoft.Extensions.Logging.EventLog provider package sends log output to the Windows Event Log.

```
logging.AddEventLog();
```

AddEventLog overloads let you pass in EventLogSettings. If `null` or not specified, the following default settings are used:

- `LogName`: "Application"
- `SourceName`: ".NET Runtime"
- `MachineName`: The local machine name is used.

Events are logged for Warning level and higher. The following example sets the Event Log default log level to LogLevel.Information:

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

TraceSource provider

The Microsoft.Extensions.Logging.TraceSource provider package uses the TraceSource libraries and providers.

```
logging.AddTraceSource(sourceSwitchName);
```

AddTraceSource overloads let you pass in a source switch and a trace listener.

To use this provider, an app has to run on the .NET Framework (rather than .NET Core). The provider can route messages to a variety of listeners, such as the TextWriterTraceListener used in the sample app.

Azure App Service provider

The Microsoft.Extensions.Logging.AzureAppServices provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account.

```
logging.AddAzureWebAppDiagnostics();
```

The provider package isn't included in the Microsoft.AspNetCore.App metapackage. When targeting .NET Framework or referencing the `Microsoft.AspNetCore.App` metapackage, add the provider package to the project.

An AddAzureWebAppDiagnostics overload lets you pass in AzureAppServicesDiagnosticsSettings. The settings object can override default settings, such as the logging output template, blob name, and file size limit. (*Output template* is a message template that's applied to all logs in addition to what's provided with an `ILogger` method call.)

When you deploy to an App Service app, the application honors the settings in the App Service logs section of the **App Service** page of the Azure portal. When the following settings are updated, the changes take effect immediately without requiring a restart or redeployment of the app.

- **Application Logging (Filesystem)**
- **Application Logging (Blob)**

The default location for log files is in the *D:\home\LogFiles\Application* folder, and the default file name is *diagnostics-yyyymmdd.txt*. The default file size limit is 10 MB, and the default maximum number of files retained is 2. The default blob name is *{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt*.

The provider only works when the project runs in the Azure environment. It has no effect when the project is run locally—it doesn't write to local files or local development storage for blobs.

Azure log streaming

Azure log streaming lets you view log activity in real time from:

- The app server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the **App Service logs** page from your app's portal page.
- Set **Application Logging (Filesystem)** to **On**.
- Choose the log **Level**. This setting only applies to Azure log streaming, not other logging providers in the app.

Navigate to the **Log Stream** page to view app messages. They're logged by the app through the `ILogger` interface.

Azure Application Insights trace logging

The Microsoft.Extensions.Logging.ApplicationInsights provider package writes logs to Azure Application Insights.

Application Insights is a service that monitors a web app and provides tools for querying and analyzing the telemetry data. If you use this provider, you can query and analyze your logs by using the Application Insights tools.

The provider package isn't included in the shared framework. To use the provider, add the provider package to the project. The logging provider is included as a dependency of Microsoft.ApplicationInsights.AspNetCore, which is the package that provides all available telemetry for ASP.NET Core. If you use this package, you don't have to install the provider package.

Don't use the Microsoft.ApplicationInsights.Web package—that's for ASP.NET 4.x.

For more information, see the following resources:

- Application Insights overview
- Application Insights for ASP.NET Core applications - Start here if you want to implement the full range of Application Insights telemetry along with logging.
- ApplicationInsightsLoggerProvider for .NET Core ILogger logs - Start here if you want to implement the logging provider without the rest of Application Insights telemetry.
- Application Insights logging adapters.
- Install, configure, and initialize the Application Insights SDK - Interactive tutorial on the Microsoft Learn site.

Third-party logging providers

Third-party logging frameworks that work with ASP.NET Core:

- elmah.io (GitHub repo)
- Gelf (GitHub repo)
- JSNLog (GitHub repo)
- KissLog.net (GitHub repo)
- Log4Net (GitHub repo)
- Loggr (GitHub repo)
- NLog (GitHub repo)
- Sentry (GitHub repo)
- Serilog (GitHub repo)
- Stackdriver (Github repo)

Some third-party frameworks can perform semantic logging, also known as structured logging.

Using a third-party framework is similar to using one of the built-in providers:

1. Add a NuGet package to your project.
2. Call an `ILoggerFactory` or `ILoggingBuilder` extension method provided by the logging framework.

For more information, see each provider's documentation. Third-party logging providers aren't supported by Microsoft.

Additional resources

- High-performance logging with LoggerMessage in ASP.NET Core

Routing in ASP.NET Core

9/22/2020 • 101 minutes to read • Edit Online

By Ryan Nowak, Kirk Larkin, and Rick Anderson

Routing is responsible for matching incoming HTTP requests and dispatching those requests to the app's executable endpoints. Endpoints are the app's units of executable request-handling code. Endpoints are defined in the app and configured when the app starts. The endpoint matching process can extract values from the request's URL and provide those values for request processing. Using endpoint information from the app, routing is also able to generate URLs that map to endpoints.

Apps can configure routing using:

- Controllers
- Razor Pages
- SignalR
- gRPC Services
- Endpoint-enabled middleware such as Health Checks.
- Delegates and lambdas registered with routing.

This document covers low-level details of ASP.NET Core routing. For information on configuring routing:

- For controllers, see Routing to controller actions in ASP.NET Core.
- For Razor Pages conventions, see Razor Pages route and app conventions in ASP.NET Core.

The endpoint routing system described in this document applies to ASP.NET Core 3.0 and later. For information on the previous routing system based on IRouter, select the ASP.NET Core 2.1 version using one of the following approaches:

- The version selector for a previous version.
- Select .

View or download sample code (how to download)

The download samples for this document are enabled by a specific `Startup` class. To run a specific sample, modify *Program.cs* to call the desired `Startup` class.

Routing basics

All ASP.NET Core templates include routing in the generated code. Routing is registered in the middleware pipeline in `Startup.Configure`.

The following code shows a basic example of routing:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

Routing uses a pair of middleware, registered by UseRouting and UseEndpoints:

- `UseRouting` adds route matching to the middleware pipeline. This middleware looks at the set of endpoints defined in the app, and selects the best match based on the request.
- `UseEndpoints` adds endpoint execution to the middleware pipeline. It runs the delegate associated with the selected endpoint.

The preceding example includes a single *route to code* endpoint using the MapGet method:

- When an HTTP `GET` request is sent to the root URL `/`:
  - The request delegate shown executes.
  - `Hello World!` is written to the HTTP response. By default, the root URL `/` is `https://localhost:5001/`.
- If the request method is not `GET` or the root URL is not `/`, no route matches and an HTTP 404 is returned.

Endpoint

The `MapGet` method is used to define an **endpoint**. An endpoint is something that can be:

- Selected, by matching the URL and HTTP method.
- Executed, by running the delegate.

Endpoints that can be matched and executed by the app are configured in `UseEndpoints`. For example, MapGet, MapPost, and similar methods connect request delegates to the routing system. Additional methods can be used to connect ASP.NET Core framework features to the routing system:

- MapRazorPages for Razor Pages
- MapControllers for controllers
- MapHub<THub> for SignalR
- MapGrpcService<TService> for gRPC

The following example shows routing with a more sophisticated route template:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/hello/{name:alpha}", async context =>
    {
        var name = context.Request.RouteValues["name"];
        await context.Response.WriteAsync($"Hello {name}!");
    });
});
```

The string `/hello/{name:alpha}` is a **route template**. It is used to configure how the endpoint is matched. In this case, the template matches:

- A URL like `/hello/Ryan`
- Any URL path that begins with `/hello/` followed by a sequence of alphabetic characters. `:alpha` applies a route constraint that matches only alphabetic characters. Route constraints are explained later in this document.

The second segment of the URL path, `{name:alpha}`:

- Is bound to the `name` parameter.
- Is captured and stored in HttpRequest.RouteValues.

The endpoint routing system described in this document is new as of ASP.NET Core 3.0. However, all versions of ASP.NET Core support the same set of route template features and route constraints.

The following example shows routing with health checks and authorization:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Matches request to an endpoint.
    app.UseRouting();

    // Endpoint aware middleware.
    // Middleware can use metadata from the matched endpoint.
    app.UseAuthentication();
    app.UseAuthorization();

    // Execute the matched endpoint.
    app.UseEndpoints(endpoints =>
    {
        // Configure the Health Check endpoint and require an authorized user.
        endpoints.MapHealthChecks("/healthz").RequireAuthorization();

        // Configure another endpoint, no authorization requirements.
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding example demonstrates how:

- The authorization middleware can be used with routing.
- Endpoints can be used to configure authorization behavior.

The MapHealthChecks call adds a health check endpoint. Chaining RequireAuthorization on to this call attaches an authorization policy to the endpoint.

Calling UseAuthentication and UseAuthorization adds the authentication and authorization middleware. These middleware are placed between UseRouting and `UseEndpoints` so that they can:

- See which endpoint was selected by `UseRouting`.
- Apply an authorization policy before UseEndpoints dispatches to the endpoint.

Endpoint metadata

In the preceding example, there are two endpoints, but only the health check endpoint has an authorization policy attached. If the request matches the health check endpoint, `/healthz`, an authorization check is performed. This demonstrates that endpoints can have extra data attached to them. This extra data is called endpoint **metadata**:

- The metadata can be processed by routing-aware middleware.
- The metadata can be of any .NET type.

Routing concepts

The routing system builds on top of the middleware pipeline by adding the powerful **endpoint** concept. Endpoints represent units of the app's functionality that are distinct from each other in terms of routing, authorization, and any number of ASP.NET Core's systems.

ASP.NET Core endpoint definition

An ASP.NET Core endpoint is:

- Executable: Has a RequestDelegate.
- Extensible: Has a Metadata collection.
- Selectable: Optionally, has routing information.
- Enumerable: The collection of endpoints can be listed by retrieving the EndpointDataSource from DI.

The following code shows how to retrieve and inspect the endpoint matching the current request:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.Use(next => context =>
    {
        var endpoint = context.GetEndpoint();
        if (endpoint is null)
        {
            return Task.CompletedTask;
        }

        Console.WriteLine($"Endpoint: {endpoint.DisplayName}");

        if (endpoint is RouteEndpoint routeEndpoint)
        {
            Console.WriteLine("Endpoint has route pattern: " +
                routeEndpoint.RoutePattern.RawText);
        }

        foreach (var metadata in endpoint.Metadata)
        {
            Console.WriteLine($"Endpoint has metadata: {metadata}");
        }

        return Task.CompletedTask;
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

The endpoint, if selected, can be retrieved from the `HttpContext`. Its properties can be inspected. Endpoint objects are immutable and cannot be modified after creation. The most common type of endpoint is a RouteEndpoint. `RouteEndpoint` includes information that allows it to be to selected by the routing system.

In the preceding code, app.Use configures an in-line middleware.

The following code shows that, depending on where `app.Use` is called in the pipeline, there may not be an endpoint:

```
// Location 1: before routing runs, endpoint is always null here
app.Use(next => context =>
{
    Console.WriteLine($"1. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");
    return next(context);
});

app.UseRouting();

// Location 2: after routing runs, endpoint will be non-null if routing found a match
app.Use(next => context =>
{
    Console.WriteLine($"2. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");
    return next(context);
});

app.UseEndpoints(endpoints =>
{
    // Location 3: runs when this endpoint matches
    endpoints.MapGet("/", context =>
    {
        Console.WriteLine(
            $"3. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");
        return Task.CompletedTask;
    }).WithDisplayName("Hello");
});

// Location 4: runs after UseEndpoints - will only run if there was no match
app.Use(next => context =>
{
    Console.WriteLine($"4. Endpoint: {context.GetEndpoint()?.DisplayName ?? "(null)"}");
    return next(context);
});
```

This preceding sample adds `Console.WriteLine` statements that display whether or not an endpoint has been selected. For clarity, the sample assigns a display name to the provided `/` endpoint.

Running this code with a URL of `/` displays:

```
1. Endpoint: (null)
2. Endpoint: Hello
3. Endpoint: Hello
```

Running this code with any other URL displays:

```
1. Endpoint: (null)
2. Endpoint: (null)
4. Endpoint: (null)
```

This output demonstrates that:

- The endpoint is always null before `UseRouting` is called.
- If a match is found, the endpoint is non-null between `UseRouting` and UseEndpoints.
- The `UseEndpoints` middleware is **terminal** when a match is found. Terminal middleware is defined later in this document.
- The middleware after `UseEndpoints` execute only when no match is found.

The `UseRouting` middleware uses the SetEndpoint method to attach the endpoint to the current context. It's possible to replace the `UseRouting` middleware with custom logic and still get the benefits of using

endpoints. Endpoints are a low-level primitive like middleware, and aren't coupled to the routing implementation. Most apps don't need to replace `UseRouting` with custom logic.

The `UseEndpoints` middleware is designed to be used in tandem with the `UseRouting` middleware. The core logic to execute an endpoint isn't complicated. Use GetEndpoint to retrieve the endpoint, and then invoke its RequestDelegate property.

The following code demonstrates how middleware can influence or react to routing:

```
public class IntegratedMiddlewareStartup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        // Location 1: Before routing runs. Can influence request before routing runs.
        app.UseHttpMethodOverride();

        app.UseRouting();

        // Location 2: After routing runs. Middleware can match based on metadata.
        app.Use(next => context =>
        {
            var endpoint = context.GetEndpoint();
            if (endpoint?.Metadata.GetMetadata<AuditPolicyAttribute>()?.NeedsAudit
                                                                        == true)
            {
                Console.WriteLine($"ACCESS TO SENSITIVE DATA AT: {DateTime.UtcNow}");
            }

            return next(context);
        });

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello world!");
            });

            // Using metadata to configure the audit policy.
            endpoints.MapGet("/sensitive", async context =>
            {
                await context.Response.WriteAsync("sensitive data");
            })
            .WithMetadata(new AuditPolicyAttribute(needsAudit: true));
        });

    }
}

public class AuditPolicyAttribute : Attribute
{
    public AuditPolicyAttribute(bool needsAudit)
    {
        NeedsAudit = needsAudit;
    }

    public bool NeedsAudit { get; }
}
```

The preceding example demonstrates two important concepts:

- Middleware can run before `UseRouting` to modify the data that routing operates upon.
  - Usually middleware that appears before routing modifies some property of the request, such as UseRewriter, UseHttpMethodOverride, or UsePathBase.
- Middleware can run between `UseRouting` and UseEndpoints to process the results of routing before the endpoint is executed.
  - Middleware that runs between `UseRouting` and `UseEndpoints`:
    - Usually inspects metadata to understand the endpoints.
    - Often makes security decisions, as done by `UseAuthorization` and `UseCors`.
  - The combination of middleware and metadata allows configuring policies per-endpoint.

The preceding code shows an example of a custom middleware that supports per-endpoint policies. The middleware writes an *audit log* of access to sensitive data to the console. The middleware can be configured to *audit* an endpoint with the `AuditPolicyAttribute` metadata. This sample demonstrates an *opt-in* pattern where only endpoints that are marked as sensitive are audited. It's possible to define this logic in reverse, auditing everything that isn't marked as safe, for example. The endpoint metadata system is flexible. This logic could be designed in whatever way suits the use case.

The preceding sample code is intended to demonstrate the basic concepts of endpoints. **The sample is not intended for production use**. A more complete version of an *audit log* middleware would:

- Log to a file or database.
- Include details such as the user, IP address, name of the sensitive endpoint, and more.

The audit policy metadata `AuditPolicyAttribute` is defined as an `Attribute` for easier use with class-based frameworks such as controllers and SignalR. When using *route to code*:

- Metadata is attached with a builder API.
- Class-based frameworks include all attributes on the corresponding method and class when creating endpoints.

The best practices for metadata types are to define them either as interfaces or attributes. Interfaces and attributes allow code reuse. The metadata system is flexible and doesn't impose any limitations.

Comparing a terminal middleware and routing

The following code sample contrasts using middleware with using routing:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Approach 1: Writing a terminal middleware.
    app.Use(next => async context =>
    {
        if (context.Request.Path == "/")
        {
            await context.Response.WriteAsync("Hello terminal middleware!");
            return;
        }

        await next(context);
    });

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        // Approach 2: Using routing.
        endpoints.MapGet("/Movie", async context =>
        {
            await context.Response.WriteAsync("Hello routing!");
        });
    });
}
```

The style of middleware shown with `Approach 1:` is **terminal middleware**. It's called terminal middleware because it does a matching operation:

- The matching operation in the preceding sample is `Path == "/"` for the middleware and `Path == "/Movie"` for routing.
- When a match is successful, it executes some functionality and returns, rather than invoking the `next` middleware.

It's called terminal middleware because it terminates the search, executes some functionality, and then returns.

Comparing a terminal middleware and routing:

- Both approaches allow terminating the processing pipeline:
  - Middleware terminates the pipeline by returning rather than invoking `next`.
  - Endpoints are always terminal.
- Terminal middleware allows positioning the middleware at an arbitrary place in the pipeline:
  - Endpoints execute at the position of UseEndpoints.
- Terminal middleware allows arbitrary code to determine when the middleware matches:
  - Custom route matching code can be verbose and difficult to write correctly.
  - Routing provides straightforward solutions for typical apps. Most apps don't require custom route matching code.
- Endpoints interface with middleware such as `UseAuthorization` and `UseCors`.
  - Using a terminal middleware with `UseAuthorization` or `UseCors` requires manual interfacing with the authorization system.

An endpoint defines both:

- A delegate to process requests.
- A collection of arbitrary metadata. The metadata is used to implement cross-cutting concerns based on policies and configuration attached to each endpoint.

Terminal middleware can be an effective tool, but can require:

- A significant amount of coding and testing.
- Manual integration with other systems to achieve the desired level of flexibility.

Consider integrating with routing before writing a terminal middleware.

Existing terminal middleware that integrates with Map or MapWhen can usually be turned into a routing aware endpoint. MapHealthChecks demonstrates the pattern for router-ware:

- Write an extension method on IEndpointRouteBuilder.
- Create a nested middleware pipeline using CreateApplicationBuilder.
- Attach the middleware to the new pipeline. In this case, UseHealthChecks.
- Build the middleware pipeline into a RequestDelegate.
- Call `Map` and provide the new middleware pipeline.
- Return the builder object provided by `Map` from the extension method.

The following code shows use of MapHealthChecks:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    // Matches request to an endpoint.
    app.UseRouting();

    // Endpoint aware middleware.
    // Middleware can use metadata from the matched endpoint.
    app.UseAuthentication();
    app.UseAuthorization();

    // Execute the matched endpoint.
    app.UseEndpoints(endpoints =>
    {
        // Configure the Health Check endpoint and require an authorized user.
        endpoints.MapHealthChecks("/healthz").RequireAuthorization();

        // Configure another endpoint, no authorization requirements.
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

The preceding sample shows why returning the builder object is important. Returning the builder object allows the app developer to configure policies such as authorization for the endpoint. In this example, the health checks middleware has no direct integration with the authorization system.

The metadata system was created in response to the problems encountered by extensibility authors using terminal middleware. It's problematic for each middleware to implement its own integration with the authorization system.

URL matching

- Is the process by which routing matches an incoming request to an endpoint.
- Is based on data in the URL path and headers.
- Can be extended to consider any data in the request.

When a routing middleware executes, it sets an `Endpoint` and route values to a request feature on the HttpContext from the current request:

- Calling HttpContext.GetEndpoint gets the endpoint.
- `HttpRequest.RouteValues` gets the collection of route values.

Middleware running after the routing middleware can inspect the endpoint and take action. For example, an authorization middleware can interrogate the endpoint's metadata collection for an authorization policy. After all of the middleware in the request processing pipeline is executed, the selected endpoint's delegate is invoked.

The routing system in endpoint routing is responsible for all dispatching decisions. Because the middleware applies policies based on the selected endpoint, it's important that:

- Any decision that can affect dispatching or the application of security policies is made inside the routing system.

> **WARNING**
> For backwards-compatibility, when a Controller or Razor Pages endpoint delegate is executed, the properties of RouteContext.RouteData are set to appropriate values based on the request processing performed thus far.
>
> The `RouteContext` type will be marked obsolete in a future release:
>
> - Migrate `RouteData.Values` to `HttpRequest.RouteValues`.
> - Migrate `RouteData.DataTokens` to retrieve IDataTokensMetadata from the endpoint metadata.

URL matching operates in a configurable set of phases. In each phase, the output is a set of matches. The set of matches can be narrowed down further by the next phase. The routing implementation does not guarantee a processing order for matching endpoints. **All** possible matches are processed at once. The URL matching phases occur in the following order. ASP.NET Core:

1. Processes the URL path against the set of endpoints and their route templates, collecting **all** of the matches.
2. Takes the preceding list and removes matches that fail with route constraints applied.
3. Takes the preceding list and removes matches that fail the set of MatcherPolicy instances.
4. Uses the EndpointSelector to make a final decision from the preceding list.

The list of endpoints is prioritized according to:

- The RouteEndpoint.Order
- The route template precedence

All matching endpoints are processed in each phase until the EndpointSelector is reached. The `EndpointSelector` is the final phase. It chooses the highest priority endpoint from the matches as the best match. If there are other matches with the same priority as the best match, an ambiguous match exception is thrown.

The route precedence is computed based on a **more specific** route template being given a higher priority. For example, consider the templates `/hello` and `/{message}`:

- Both match the URL path `/hello`.
- `/hello` is more specific and therefore higher priority.

In general, route precedence does a good job of choosing the best match for the kinds of URL schemes used in practice. Use Order only when necessary to avoid an ambiguity.

Due to the kinds of extensibility provided by routing, it isn't possible for the routing system to compute ahead of time the ambiguous routes. Consider an example such as the route templates `/{message:alpha}` and `/{message:int}`:

- The `alpha` constraint matches only alphabetic characters.
- The `int` constraint matches only numbers.
- These templates have the same route precedence, but there's no single URL they both match.
- If the routing system reported an ambiguity error at startup, it would block this valid use case.

> **WARNING**
>
> The order of operations inside UseEndpoints doesn't influence the behavior of routing, with one exception. MapControllerRoute and MapAreaRoute automatically assign an order value to their endpoints based on the order they are invoked. This simulates long-time behavior of controllers without the routing system providing the same guarantees as older routing implementations.
>
> In the legacy implementation of routing, it's possible to implement routing extensibility that has a dependency on the order in which routes are processed. Endpoint routing in ASP.NET Core 3.0 and later:
>
> - Doesn't have a concept of routes.
> - Doesn't provide ordering guarantees. All endpoints are processed at once.
>
> If this means you're stuck using the legacy routing system, open a GitHub issue for assistance.

Route template precedence and endpoint selection order

Route template precedence is a system that assigns each route template a value based on how specific it is. Route template precedence:

- Avoids the need to adjust the order of endpoints in common cases.
- Attempts to match the common-sense expectations of routing behavior.

For example, consider templates `/Products/List` and `/Products/{id}`. It would be reasonable to assume that `/Products/List` is a better match than `/Products/{id}` for the URL path `/Products/List`. The works because the literal segment `/List` is considered to have better precedence than the parameter segment `/{id}`.

The details of how precedence works are coupled to how route templates are defined:

- Templates with more segments are considered more specific.
- A segment with literal text is considered more specific than a parameter segment.
- A parameter segment with a constraint is considered more specific than one without.
- A complex segment is considered as specific as a parameter segment with a constraint.
- Catch-all parameters are the least specific. See **catch-all** in the Route template reference for important information on catch-all routes.

See the source code on GitHub for a reference of exact values.

URL generation concepts

URL generation:

- Is the process by which routing can create a URL path based on a set of route values.

- Allows for a logical separation between endpoints and the URLs that access them.

Endpoint routing includes the LinkGenerator API. `LinkGenerator` is a singleton service available from DI. The `LinkGenerator` API can be used outside of the context of an executing request. Mvc.IUrlHelper and scenarios that rely on IUrlHelper, such as Tag Helpers, HTML Helpers, and Action Results, use the `LinkGenerator` API internally to provide link generating capabilities.

The link generator is backed by the concept of an **address** and **address schemes**. An address scheme is a way of determining the endpoints that should be considered for link generation. For example, the route name and route values scenarios many users are familiar with from controllers and Razor Pages are implemented as an address scheme.

The link generator can link to controllers and Razor Pages via the following extension methods:

- GetPathByAction
- GetUriByAction
- GetPathByPage
- GetUriByPage

Overloads of these methods accept arguments that include the `HttpContext`. These methods are functionally equivalent to Url.Action and Url.Page, but offer additional flexibility and options.

The `GetPath*` methods are most similar to `Url.Action` and `Url.Page`, in that they generate a URI containing an absolute path. The `GetUri*` methods always generate an absolute URI containing a scheme and host. The methods that accept an `HttpContext` generate a URI in the context of the executing request. The ambient route values, URL base path, scheme, and host from the executing request are used unless overridden.

LinkGenerator is called with an address. Generating a URI occurs in two steps:

1. An address is bound to a list of endpoints that match the address.
2. Each endpoint's RoutePattern is evaluated until a route pattern that matches the supplied values is found. The resulting output is combined with the other URI parts supplied to the link generator and returned.

The methods provided by LinkGenerator support standard link generation capabilities for any type of address. The most convenient way to use the link generator is through extension methods that perform operations for a specific address type:

EXTENSION METHOD	DESCRIPTION
GetPathByAddress	Generates a URI with an absolute path based on the provided values.
GetUriByAddress	Generates an absolute URI based on the provided values.

WARNING

Pay attention to the following implications of calling LinkGenerator methods:

- Use `GetUri*` extension methods with caution in an app configuration that doesn't validate the `Host` header of incoming requests. If the `Host` header of incoming requests isn't validated, untrusted request input can be sent back to the client in URIs in a view or page. We recommend that all production apps configure their server to validate the `Host` header against known valid values.
- Use LinkGenerator with caution in middleware in combination with `Map` or `MapWhen`. `Map*` changes the base path of the executing request, which affects the output of link generation. All of the LinkGenerator APIs allow specifying a base path. Specify an empty base path to undo the `Map*` affect on link generation.

Middleware example

In the following example, a middleware uses the LinkGenerator API to create a link to an action method that lists store products. Using the link generator by injecting it into a class and calling `GenerateLink` is available to any class in an app:

```
public class ProductsLinkMiddleware
{
    private readonly LinkGenerator _linkGenerator;

    public ProductsLinkMiddleware(RequestDelegate next, LinkGenerator linkGenerator)
    {
        _linkGenerator = linkGenerator;
    }

    public async Task InvokeAsync(HttpContext httpContext)
    {
        var url = _linkGenerator.GetPathByAction("ListProducts", "Store");

        httpContext.Response.ContentType = "text/plain";

        await httpContext.Response.WriteAsync($"Go to {url} to see our products.");
    }
}
```

Route template reference

Tokens within `{}` define route parameters that are bound if the route is matched. More than one route parameter can be defined in a route segment, but route parameters must be separated by a literal value. For example, `{controller=Home}{action=Index}` isn't a valid route, since there's no literal value between `{controller}` and `{action}`. Route parameters must have a name and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URL's path. To match a literal route parameter delimiter `{` or `}`, escape the delimiter by repeating the character. For example `{{` or `}}`.

Asterisk `*` or double asterisk `**`:

- Can be used as a prefix to a route parameter to bind to the rest of the URI.
- Are called a **catch-all** parameters. For example, `blog/{**slug}`:
  - Matches any URI that starts with `/blog` and has any value following it.
  - The value following `/blog` is assigned to the slug route value.

WARNING

A **catch-all** parameter may match routes incorrectly due to a bug in routing. Apps impacted by this bug have the following characteristics:

- A catch-all route, for example, `{**slug}"`
- The catch-all route fails to match requests it should match.
- Removing other routes makes catch-all route start working.

See GitHub bugs 18677 and 16579 for example cases that hit this bug.

An opt-in fix for this bug is contained in .NET Core 3.1.301 SDK and later. The following code sets an internal switch that fixes this bug:

```
public static void Main(string[] args)
{
   AppContext.SetSwitch("Microsoft.AspNetCore.Routing.UseCorrectCatchAllBehavior",
                        true);
   CreateHostBuilder(args).Build().Run();
}
// Remaining code removed for brevity.
```

Catch-all parameters can also match the empty string.

The catch-all parameter escapes the appropriate characters when the route is used to generate a URL, including path separator `/` characters. For example, the route `foo/{*path}` with route values `{ path = "my/path" }` generates `foo/my%2Fpath`. Note the escaped forward slash. To round-trip path separator characters, use the `**` route parameter prefix. The route `foo/{**path}` with `{ path = "my/path" }` generates `foo/my/path`.

URL patterns that attempt to capture a file name with an optional file extension have additional considerations. For example, consider the template `files/{filename}.{ext?}`. When values for both `filename` and `ext` exist, both values are populated. If only a value for `filename` exists in the URL, the route matches because the trailing `.` is optional. The following URLs match this route:

- `/files/myFile.txt`
- `/files/myFile`

Route parameters may have **default values** designated by specifying the default value after the parameter name separated by an equals sign (`=`). For example, `{controller=Home}` defines `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. Route parameters are made optional by appending a question mark (`?`) to the end of the parameter name. For example, `id?`. The difference between optional values and default route parameters is:

- A route parameter with a default value always produces a value.
- An optional parameter has a value only when a value is provided by the request URL.

Route parameters may have constraints that must match the route value bound from the URL. Adding `:` and constraint name after the route parameter name specifies an inline constraint on a route parameter. If the constraint requires arguments, they're enclosed in parentheses `(...)` after the constraint name. Multiple *inline constraints* can be specified by appending another `:` and constraint name.

The constraint name and arguments are passed to the IInlineConstraintResolver service to create an instance of IRouteConstraint to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies a `minlength` constraint with the argument `10`. For more information on route constraints and a list of the constraints provided by the framework, see the Route constraint reference section.

Route parameters may also have parameter transformers. Parameter transformers transform a parameter's value when generating links and matching actions and pages to URLs. Like constraints, parameter transformers can be added inline to a route parameter by adding a `:` and transformer name after the route parameter name. For example, the route template `blog/{article:slugify}` specifies a `slugify` transformer. For more information on parameter transformers, see the Parameter transformer reference section.

The following table demonstrates example route templates and their behavior:

ROUTE TEMPLATE	EXAMPLE MATCHING URI	THE REQUEST URI...
`hello`	`/hello`	Only matches the single path `/hello`.
`{Page=Home}`	`/`	Matches and sets `Page` to `Home`.
`{Page=Home}`	`/Contact`	Matches and sets `Page` to `Contact`.
`{controller}/{action}/{id?}`	`/Products/List`	Maps to the `Products` controller and `List` action.
`{controller}/{action}/{id?}`	`/Products/Details/123`	Maps to the `Products` controller and `Details` action with `id` set to 123.
`{controller=Home}/{action=Index}/{id?}`	`/`	Maps to the `Home` controller and `Index` method. `id` is ignored.
`{controller=Home}/{action=Index}/{id?}`	`/Products`	Maps to the `Products` controller and `Index` method. `id` is ignored.

Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

Complex segments

Complex segments are processed by matching up literal delimiters from right to left in a non-greedy way. For example, `[Route("/a{b}c{d}")]` is a complex segment. Complex segments work in a particular way that must be understood to use them successfully. The example in this section demonstrates why complex segments only really work well when the delimiter text doesn't appear inside the parameter values. Using a regex and then manually extracting the values is needed for more complex cases.

> **WARNING**
>
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

This is a summary of the steps that routing performs with the template `/a{b}c{d}` and the URL path `/abcd`. The `|` is used to help visualize how the algorithm works:

- The first literal, right to left, is `c`. So `/abcd` is searched from right and finds `/ab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/ab|c|d` is searched starting where we left off, then `a` is found

`/|a|b|c|d`.

- The value to the right (`b`) is now matched to the route parameter `{b}`.
- There is no remaining text and no remaining route template, so this is a match.

Here's an example of a negative case using the same template `/a{b}c{d}` and the URL path `/aabcd`. The `|` is used to help visualize how the algorithm works. This case isn't a match, which is explained by the same algorithm:

- The first literal, right to left, is `c`. So `/aabcd` is searched from right and finds `/aab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/aab|c|d` is searched starting where we left off, then `a` is found `/a|a|b|c|d`.
- The value to the right (`b`) is now matched to the route parameter `{b}`.
- At this point there is remaining text `a`, but the algorithm has run out of route template to parse, so this is not a match.

Since the matching algorithm is non-greedy:

- It matches the smallest amount of text possible in each step.
- Any case where the delimiter value appears inside the parameter values results in not matching.

Regular expressions provide much more control over their matching behavior.

Greedy matching, also know as lazy matching, matches the largest possible string. Non-greedy matches the smallest possible string.

Route constraint reference

Route constraints execute when a match has occurred to the incoming URL and the URL path is tokenized into route values. Route constraints generally inspect the route value associated via the route template and make a true or false decision about whether the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the HttpMethodRouteConstraint can accept or reject a request based on its HTTP verb. Constraints are used in routing requests and link generation.

> **WARNING**
> Don't use constraints for input validation. If constraints are used for input validation, invalid input results in a `404` Not Found response. Invalid input should produce a `400` Bad Request with an appropriate error message. Route constraints are used to disambiguate similar routes, not to validate the inputs for a particular route.

The following table demonstrates example route constraints and their expected behavior:

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`int`	`{id:int}`	`123456789`, `-123456789`	Matches any integer
`bool`	`{active:bool}`	`true`, `FALSE`	Matches `true` or `false`. Case-insensitive
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Matches a valid `DateTime` value in the invariant culture. See preceding warning.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Matches a valid `decimal` value in the invariant culture. See preceding warning.
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Matches a valid `double` value in the invariant culture. See preceding warning.
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Matches a valid `float` value in the invariant culture. See preceding warning.
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`	Matches a valid `Guid` value
`long`	`{ticks:long}`	`123456789`, `-123456789`	Matches a valid `long` value
`minlength(value)`	`{username:minlength(4)}`	`Rick`	String must be at least 4 characters
`maxlength(value)`	`{filename:maxlength(8)}`	`MyFile`	String must be no more than 8 characters
`length(length)`	`{filename:length(12)}`	`somefile.txt`	String must be exactly 12 characters long
`length(min,max)`	`{filename:length(8,16)}`	`somefile.txt`	String must be at least 8 and no more than 16 characters long
`min(value)`	`{age:min(18)}`	`19`	Integer value must be at least 18
`max(value)`	`{age:max(120)}`	`91`	Integer value must be no more than 120
`range(min,max)`	`{age:range(18,120)}`	`91`	Integer value must be at least 18 but no more than 120
`alpha`	`{name:alpha}`	`Rick`	String must consist of one or more alphabetical characters, `a`-`z` and case-insensitive.
`regex(expression)`	`{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}`	`123-45-6789`	String must match the regular expression. See tips about defining a regular expression.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`required`	`{name:required}`	`Rick`	Used to enforce that a non-parameter value is present during URL generation

> **WARNING**
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Multiple, colon delimited constraints can be applied to a single parameter. For example, the following constraint restricts a parameter to an integer value of 1 or greater:

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { }
```

> **WARNING**
> Route constraints that verify the URL and are converted to a CLR type always use the invariant culture. For example, conversion to the CLR type `int` or `DateTime`. These constraints assume that the URL is not localizable. The framework-provided route constraints don't modify the values stored in route values. All route values parsed from the URL are stored as strings. For example, the `float` constraint attempts to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

Regular expressions in constraints

> **WARNING**
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Regular expressions can be specified as inline constraints using the `regex(...)` route constraint. Methods in the MapControllerRoute family also accept an object literal of constraints. If that form is used, string values are interpreted as regular expressions.

The following code uses an inline regex constraint:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("{message:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}",
        context =>
        {
            return context.Response.WriteAsync("inline-constraint match");
        });
 });
```

The following code uses an object literal to specify a regex constraint:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "people",
        pattern: "People/{ssn}",
        constraints: new { ssn = "^\\d{3}-\\d{2}-\\d{4}$", },
        defaults: new { controller = "People", action = "List", });
});
```

The ASP.NET Core framework adds `RegexOptions.IgnoreCase | RegexOptions.Compiled | RegexOptions.CultureInvariant` to the regular expression constructor. See RegexOptions for a description of these members.

Regular expressions use delimiters and tokens similar to those used by routing and the C# language. Regular expression tokens must be escaped. To use the regular expression `^\d{3}-\d{2}-\d{4}$` in an inline constraint, use one of the following:

- Replace `\` characters provided in the string as `\\` characters in the C# source file in order to escape the `\` string escape character.
- Verbatim string literals.

To escape routing parameter delimiter characters `{`, `}`, `[`, `]`, double the characters in the expression, for example, `{{`, `}}`, `[[`, `]]`. The following table shows a regular expression and its escaped version:

REGULAR EXPRESSION	ESCAPED REGULAR EXPRESSION
`^\d{3}-\d{2}-\d{4}$`	`^\\d{{3}}-\\d{{2}}-\\d{{4}}$`
`^[a-z]{2}$`	`^[[a-z]]{{2}}$`

Regular expressions used in routing often start with the `^` character and match the starting position of the string. The expressions often end with the `$` character and match the end of the string. The `^` and `$` characters ensure that the regular expression matches the entire route parameter value. Without the `^` and `$` characters, the regular expression matches any substring within the string, which is often undesirable. The following table provides examples and explains why they match or fail to match:

EXPRESSION	STRING	MATCH	COMMENT
`[a-z]{2}`	hello	Yes	Substring matches
`[a-z]{2}`	123abc456	Yes	Substring matches
`[a-z]{2}`	mz	Yes	Matches expression
`[a-z]{2}`	MZ	Yes	Not case sensitive
`^[a-z]{2}$`	hello	No	See `^` and `$` above
`^[a-z]{2}$`	123abc456	No	See `^` and `$` above

For more information on regular expression syntax, see .NET Framework Regular Expressions.

To constrain a parameter to a known set of possible values, use a regular expression. For example, `{action:regex(^(list|get|create)$)}` only matches the `action` route value to `list`, `get`, or `create`. If

passed into the constraints dictionary, the string `^(list|get|create)$` is equivalent. Constraints that are passed in the constraints dictionary that don't match one of the known constraints are also treated as regular expressions. Constraints that are passed within a template that don't match one of the known constraints are not treated as regular expressions.

Custom route constraints

Custom route constraints can be created by implementing the IRouteConstraint interface. The `IRouteConstraint` interface contains Match, which returns `true` if the constraint is satisfied and `false` otherwise.

Custom route constraints are rarely needed. Before implementing a custom route constraint, consider alternatives, such as model binding.

The ASP.NET Core Constraints folder provides good examples of creating a constraints. For example, GuidRouteConstraint.

To use a custom `IRouteConstraint`, the route constraint type must be registered with the app's ConstraintMap in the service container. A `ConstraintMap` is a dictionary that maps route constraint keys to `IRouteConstraint` implementations that validate those constraints. An app's `ConstraintMap` can be updated in `Startup.ConfigureServices` either as part of a services.AddRouting call or by configuring RouteOptions directly with `services.Configure<RouteOptions>`. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddRouting(options =>
    {
        options.ConstraintMap.Add("customName", typeof(MyCustomConstraint));
    });
}
```

The preceding constraint is applied in the following code:

```
[Route("api/[controller]")]
[ApiController]
public class TestController : ControllerBase
{
    // GET /api/test/3
    [HttpGet("{id:customName}")]
    public IActionResult Get(string id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    // GET /api/test/my/3
    [HttpGet("my/{id:customName}")]
    public IActionResult Get(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

The implementation of `MyCustomConstraint` prevents `0` being applied to a route parameter:

```
class MyCustomConstraint : IRouteConstraint
{
    private Regex _regex;

    public MyCustomConstraint()
    {
        _regex = new Regex(@"^[1-9]*$",
                           RegexOptions.CultureInvariant | RegexOptions.IgnoreCase,
                           TimeSpan.FromMilliseconds(100));
    }
    public bool Match(HttpContext httpContext, IRouter route, string routeKey,
                      RouteValueDictionary values, RouteDirection routeDirection)
    {
        if (values.TryGetValue(routeKey, out object value))
        {
            var parameterValueString = Convert.ToString(value,
                                                        CultureInfo.InvariantCulture);
            if (parameterValueString == null)
            {
                return false;
            }

            return _regex.IsMatch(parameterValueString);
        }

        return false;
    }
}
```

WARNING

When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

The preceding code:

- Prevents `0` in the `{id}` segment of the route.
- Is shown to provide a basic example of implementing a custom constraint. It should not be used in a production app.

The following code is a better approach to preventing an `id` containing a `0` from being processed:

```
[HttpGet("{id}")]
public IActionResult Get(string id)
{
    if (id.Contains('0'))
    {
        return StatusCode(StatusCodes.Status406NotAcceptable);
    }

    return ControllerContext.MyDisplayRouteInfo(id);
}
```

The preceding code has the following advantages over the `MyCustomConstraint` approach:

- It doesn't require a custom constraint.
- It returns a more descriptive error when the route parameter includes `0`.

Parameter transformer reference

Parameter transformers:

- Execute when generating a link using LinkGenerator.
- Implement Microsoft.AspNetCore.Routing.IOutboundParameterTransformer.
- Are configured using ConstraintMap.
- Take the parameter's route value and transform it to a new string value.
- Result in using the transformed value in the generated link.

For example, a custom `slugify` parameter transformer in route pattern `blog\{article:slugify}` with `Url.Action(new { article = "MyTestArticle" })` generates `blog\my-test-article`.

Consider the following `IOutboundParameterTransformer` implementation:

```
public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) { return null; }

        return Regex.Replace(value.ToString(),
                             "([a-z])([A-Z])",
                             "$1-$2",
                             RegexOptions.CultureInvariant,
                             TimeSpan.FromMilliseconds(100)).ToLowerInvariant();
    }
}
```

To use a parameter transformer in a route pattern, configure it using ConstraintMap in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddRouting(options =>
    {
        options.ConstraintMap["slugify"] = typeof(SlugifyParameterTransformer);
    });
}
```

The ASP.NET Core framework uses parameter transformers to transform the URI where an endpoint resolves. For example, parameter transformers transform the route values used to match an `area`, `controller`, `action`, and `page`.

```
routes.MapControllerRoute(
    name: "default",
    template: "{controller:slugify=Home}/{action:slugify=Index}/{id?}");
```

With the preceding route template, the action `SubscriptionManagementController.GetAll` is matched with the URI `/subscription-management/get-all`. A parameter transformer doesn't change the route values used to generate a link. For example, `Url.Action("GetAll", "SubscriptionManagement")` outputs `/subscription-management/get-all`.

ASP.NET Core provides API conventions for using parameter transformers with generated routes:

- The Microsoft.AspNetCore.Mvc.ApplicationModels.RouteTokenTransformerConvention MVC convention applies a specified parameter transformer to all attribute routes in the app. The parameter transformer transforms attribute route tokens as they are replaced. For more information, see Use a parameter transformer to customize token replacement.
- Razor Pages uses the PageRouteTransformerConvention API convention. This convention applies a specified parameter transformer to all automatically discovered Razor Pages. The parameter transformer transforms the folder and file name segments of Razor Pages routes. For more information, see Use a parameter transformer to customize page routes.

URL generation reference

This section contains a reference for the algorithm implemented by URL generation. In practice, most complex examples of URL generation use controllers or Razor Pages. See routing in controllers for additional information.

The URL generation process begins with a call to LinkGenerator.GetPathByAddress or a similar method. The method is provided with an address, a set of route values, and optionally information about the current request from `HttpContext`.

The first step is to use the address to resolve a set of candidate endpoints using an `IEndpointAddressScheme<TAddress>` that matches the address's type.

Once of set of candidates is found by the address scheme, the endpoints are ordered and processed iteratively until a URL generation operation succeeds. URL generation does **not** check for ambiguities, the first result returned is the final result.

Troubleshooting URL generation with logging

The first step in troubleshooting URL generation is setting the logging level of `Microsoft.AspNetCore.Routing` to `TRACE`. `LinkGenerator` logs many details about its processing which can be useful to troubleshoot problems.

See URL generation reference for details on URL generation.

Addresses

Addresses are the concept in URL generation used to bind a call into the link generator to a set of candidate endpoints.

Addresses are an extensible concept that come with two implementations by default:

- Using *endpoint name* (`string`) as the address:
  - Provides similar functionality to MVC's route name.
  - Uses the IEndpointNameMetadata metadata type.
  - Resolves the provided string against the metadata of all registered endpoints.
  - Throws an exception on startup if multiple endpoints use the same name.
  - Recommended for general-purpose use outside of controllers and Razor Pages.
- Using *route values* (RouteValuesAddress) as the address:
  - Provides similar functionality to controllers and Razor Pages legacy URL generation.
  - Very complex to extend and debug.
  - Provides the implementation used by `IUrlHelper`, Tag Helpers, HTML Helpers, Action Results, etc.

The role of the address scheme is to make the association between the address and matching endpoints by arbitrary criteria:

- The endpoint name scheme performs a basic dictionary lookup.

- The route values scheme has a complex best subset of set algorithm.

Ambient values and explicit values

From the current request, routing accesses the route values of the current request `HttpContext.Request.RouteValues`. The values associated with the current request are referred to as the **ambient values**. For the purpose of clarity, the documentation refers to the route values passed in to methods as **explicit values**.

The following example shows ambient values and explicit values. It provides ambient values from the current request and explicit values: `{ id = 17, }`:

```
public class WidgetController : Controller
{
    private readonly LinkGenerator _linkGenerator;

    public WidgetController(LinkGenerator linkGenerator)
    {
        _linkGenerator = linkGenerator;
    }

    public IActionResult Index()
    {
        var url = _linkGenerator.GetPathByAction(HttpContext,
                                                 null, null,
                                                 new { id = 17, });
        return Content(url);
    }
```

The preceding code:

- Returns `/Widget/Index/17`
- Gets LinkGenerator via DI.

The following code provides no ambient values and explicit values:
`{ controller = "Home", action = "Subscribe", id = 17, }`:

```
public IActionResult Index2()
{
    var url = _linkGenerator.GetPathByAction("Subscribe", "Home",
                                             new { id = 17, });
    return Content(url);
}
```

The preceding method returns `/Home/Subscribe/17`

The following code in the `WidgetController` returns `/Widget/Subscribe/17`:

```
var url = _linkGenerator.GetPathByAction("Subscribe", null,
                                         new { id = 17, });
```

The following code provides the controller from ambient values in the current request and explicit values:
`{ action = "Edit", id = 17, }`:

```
public class GadgetController : Controller
{
    public IActionResult Index()
    {
        var url = Url.Action("Edit", new { id = 17, });
        return Content(url);
    }
```

In the preceding code:

- `/Gadget/Edit/17` is returned.
- Url gets the IUrlHelper.
- Action generates a URL with an absolute path for an action method. The URL contains the specified `action` name and `route` values.

The following code provides ambient values from the current request and explicit values: `{ page = "./Edit, id = 17, }`:

```
public class IndexModel : PageModel
{
    public void OnGet()
    {
        var url = Url.Page("./Edit", new { id = 17, });
        ViewData["URL"] = url;
    }
}
```

The preceding code sets `url` to `/Edit/17` when the Edit Razor Page contains the following page directive:

`@page "{id:int}"`

If the Edit page doesn't contain the `"{id:int}"` route template, `url` is `/Edit?id=17`.

The behavior of MVC's IUrlHelper adds a layer of complexity in addition to the rules described here:

- `IUrlHelper` always provides the route values from the current request as ambient values.
- IUrlHelper.Action always copies the current `action` and `controller` route values as explicit values unless overridden by the developer.
- IUrlHelper.Page always copies the current `page` route value as an explicit value unless overridden.
- `IUrlHelper.Page` always overrides the current `handler` route value with `null` as an explicit values unless overridden.

Users are often surprised by the behavioral details of ambient values, because MVC doesn't seem to follow its own rules. For historical and compatibility reasons, certain route values such as `action`, `controller`, `page`, and `handler` have their own special-case behavior.

The equivalent functionality provided by `LinkGenerator.GetPathByAction` and `LinkGenerator.GetPathByPage` duplicates these anomalies of `IUrlHelper` for compatibility.

URL generation process

Once the set of candidate endpoints are found, the URL generation algorithm:

- Processes the endpoints iteratively.
- Returns the first successful result.

The first step in this process is called **route value invalidation**. Route value invalidation is the process by which routing decides which route values from the ambient values should be used and which should be

ignored. Each ambient value is considered and either combined with the explicit values, or ignored.

The best way to think about the role of ambient values is that they attempt to save application developers typing, in some common cases. Traditionally, the scenarios where ambient values are helpful are related to MVC:

- When linking to another action in the same controller, the controller name doesn't need to be specified.
- When linking to another controller in the same area, the area name doesn't need to be specified.
- When linking to the same action method, route values don't need to be specified.
- When linking to another part of the app, you don't want to carry over route values that have no meaning in that part of the app.

Calls to `LinkGenerator` or `IUrlHelper` that return `null` are usually caused by not understanding route value invalidation. Troubleshoot route value invalidation by explicitly specifying more of the route values to see if that solves the problem.

Route value invalidation works on the assumption that the app's URL scheme is hierarchical, with a hierarchy formed from left-to-right. Consider the basic controller route template `{controller}/{action}/{id?}` to get an intuitive sense of how this works in practice. A **change** to a value **invalidates** all of the route values that appear to the right. This reflects the assumption about hierarchy. If the app has an ambient value for `id`, and the operation specifies a different value for the `controller`:

- `id` won't be reused because `{controller}` is to the left of `{id?}`.

Some examples demonstrating this principle:

- If the explicit values contain a value for `id`, the ambient value for `id` is ignored. The ambient values for `controller` and `action` can be used.
- If the explicit values contain a value for `action`, any ambient value for `action` is ignored. The ambient values for `controller` can be used. If the explicit value for `action` is different from the ambient value for `action`, the `id` value won't be used. If the explicit value for `action` is the same as the ambient value for `action`, the `id` value can be used.
- If the explicit values contain a value for `controller`, any ambient value for `controller` is ignored. If the explicit value for `controller` is different from the ambient value for `controller`, the `action` and `id` values won't be used. If the explicit value for `controller` is the same as the ambient value for `controller`, the `action` and `id` values can be used.

This process is further complicated by the existence of attribute routes and dedicated conventional routes. Controller conventional routes such as `{controller}/{action}/{id?}` specify a hierarchy using route parameters. For dedicated conventional routes and attribute routes to controllers and Razor Pages:

- There is a hierarchy of route values.
- They don't appear in the template.

For these cases, URL generation defines the **required values** concept. Endpoints created by controllers and Razor Pages have required values specified that allow route value invalidation to work.

The route value invalidation algorithm in detail:

- The required value names are combined with the route parameters, then processed from left-to-right.
- For each parameter, the ambient value and explicit value are compared:
  - If the ambient value and explicit value are the same, the process continues.
  - If the ambient value is present and the explicit value isn't, the ambient value is used when generating the URL.
  - If the ambient value isn't present and the explicit value is, reject the ambient value and all

subsequent ambient values.
    - If the ambient value and the explicit value are present, and the two values are different, reject the ambient value and all subsequent ambient values.

At this point, the URL generation operation is ready to evaluate route constraints. The set of accepted values is combined with the parameter default values, which is provided to constraints. If the constraints all pass, the operation continues.

Next, the **accepted values** can be used to expand the route template. The route template is processed:

- From left-to-right.
- Each parameter has its accepted value substituted.
- With the following special cases:
    - If the accepted values is missing a value and the parameter has a default value, the default value is used.
    - If the accepted values is missing a value and the parameter is optional, processing continues.
    - If any route parameter to the right of a missing optional parameter has a value, the operation fails.
    - Contiguous default-valued parameters and optional parameters are collapsed where possible.

Values explicitly provided that don't match a segment of the route are added to the query string. The following table shows the result when using the route template `{controller}/{action}/{id?}`.

AMBIENT VALUES	EXPLICIT VALUES	RESULT
controller = "Home"	action = "About"	`/Home/About`
controller = "Home"	controller = "Order", action = "About"	`/Order/About`
controller = "Home", color = "Red"	action = "About"	`/Home/About`
controller = "Home"	action = "About", color = "Red"	`/Home/About?color=Red`

Problems with route value invalidation

As of ASP.NET Core 3.0, some URL generation schemes used in earlier ASP.NET Core versions don't work well with URL generation. The ASP.NET Core team plans to add features to address these needs in a future release. For now the best solution is to use legacy routing.

The following code shows an example of a URL generation scheme that's not supported by routing.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute("default",
                                 "{culture}/{controller=Home}/{action=Index}/{id?}");
    endpoints.MapControllerRoute("blog", "{culture}/{**slug}",
                                  new { controller = "Blog", action = "ReadPost", });
});
```

In the preceding code, the `culture` route parameter is used for localization. The desire is to have the `culture` parameter always accepted as an ambient value. However, the `culture` parameter is not accepted as an ambient value because of the way required values work:

- In the `"default"` route template, the `culture` route parameter is to the left of `controller`, so changes to `controller` won't invalidate `culture`.

- In the `"blog"` route template, the `culture` route parameter is considered to be to the right of `controller`, which appears in the required values.

Configuring endpoint metadata

The following links provide information on configuring endpoint metadata:

- Enable Cors with endpoint routing
- IAuthorizationPolicyProvider sample using a custom `[MinimumAgeAuthorize]` attribute
- Test authentication with the [Authorize] attribute
- RequireAuthorization
- Selecting the scheme with the [Authorize] attribute
- Apply policies using the [Authorize] attribute
- Role-based authorization in ASP.NET Core

Host matching in routes with RequireHost

RequireHost applies a constraint to the route which requires the specified host. The `RequireHost` or [Host] parameter can be:

- Host: `www.domain.com`, matches `www.domain.com` with any port.
- Host with wildcard: `*.domain.com`, matches `www.domain.com`, `subdomain.domain.com`, or `www.subdomain.domain.com` on any port.
- Port: `*:5000`, matches port 5000 with any host.
- Host and port: `www.domain.com:5000` or `*.domain.com:5000`, matches host and port.

Multiple parameters can be specified using `RequireHost` or `[Host]`. The constraint matches hosts valid for any of the parameters. For example, `[Host("domain.com", "*.domain.com")]` matches `domain.com`, `www.domain.com`, and `subdomain.domain.com`.

The following code uses `RequireHost` to require the specified host on the route:

```
public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", context => context.Response.WriteAsync("Hi Contoso!"))
            .RequireHost("contoso.com");
        endpoints.MapGet("/", context => context.Response.WriteAsync("AdventureWorks!"))
            .RequireHost("adventure-works.com");
        endpoints.MapHealthChecks("/healthz").RequireHost("*:8080");
    });
}
```

The following code uses the `[Host]` attribute on the controller to require any of the specified hosts:

```
[Host("contoso.com", "adventure-works.com")]
public class ProductController : Controller
{
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Host("example.com:8080")]
    public IActionResult Privacy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

When the `[Host]` attribute is applied to both the controller and action method:

- The attribute on the action is used.
- The controller attribute is ignored.

Performance guidance for routing

Most of routing was updated in ASP.NET Core 3.0 to increase performance.

When an app has performance problems, routing is often suspected as the problem. The reason routing is suspected is that frameworks like controllers and Razor Pages report the amount of time spent inside the framework in their logging messages. When there's a significant difference between the time reported by controllers and the total time of the request:

- Developers eliminate their app code as the source of the problem.
- It's common to assume routing is the cause.

Routing is performance tested using thousands of endpoints. It's unlikely that a typical app will encounter a performance problem just by being too large. The most common root cause of slow routing performance is usually a badly-behaving custom middleware.

This following code sample demonstrates a basic technique for narrowing down the source of delay:

```
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
    app.Use(next => async context =>
    {
        var sw = Stopwatch.StartNew();
        await next(context);
        sw.Stop();

        logger.LogInformation("Time 1: {ElapsedMilliseconds}ms", sw.ElapsedMilliseconds);
    });

    app.UseRouting();

    app.Use(next => async context =>
    {
        var sw = Stopwatch.StartNew();
        await next(context);
        sw.Stop();

        logger.LogInformation("Time 2: {ElapsedMilliseconds}ms", sw.ElapsedMilliseconds);
    });

    app.UseAuthorization();

    app.Use(next => async context =>
    {
        var sw = Stopwatch.StartNew();
        await next(context);
        sw.Stop();

        logger.LogInformation("Time 3: {ElapsedMilliseconds}ms", sw.ElapsedMilliseconds);
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Timing test.");
        });
    });
}
```

To time routing:

- Interleave each middleware with a copy of the timing middleware shown in the preceding code.
- Add a unique identifier to correlate the timing data with the code.

This is a basic way to narrow down the delay when it's significant, for example, more than `10ms`. Subtracting `Time 2` from `Time 1` reports the time spent inside the `UseRouting` middleware.

The following code uses a more compact approach to the preceding timing code:

```
public sealed class MyStopwatch : IDisposable
{
    ILogger<Startup> _logger;
    string _message;
    Stopwatch _sw;

    public MyStopwatch(ILogger<Startup> logger, string message)
    {
        _logger = logger;
        _message = message;
        _sw = Stopwatch.StartNew();
    }

    private bool disposed = false;


    public void Dispose()
    {
        if (!disposed)
        {
            _logger.LogInformation("{Message }: {ElapsedMilliseconds}ms",
                                   _message, _sw.ElapsedMilliseconds);

            disposed = true;
        }
    }
}
```

```
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
    int count = 0;
    app.Use(next => async context =>
    {
        using (new MyStopwatch(logger, $"Time {++count}"))
        {
            await next(context);
        }

    });

    app.UseRouting();

    app.Use(next => async context =>
    {
        using (new MyStopwatch(logger, $"Time {++count}"))
        {
            await next(context);
        }
    });

    app.UseAuthorization();

    app.Use(next => async context =>
    {
        using (new MyStopwatch(logger, $"Time {++count}"))
        {
            await next(context);
        }
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Timing test.");
        });
    });
}
```

Potentially expensive routing features

The following list provides some insight into routing features that are relatively expensive compared with basic route templates:

- Regular expressions: It's possible to write regular expressions that are complex, or have long running time with a small amount of input.
- Complex segments (`{x}-{y}-{z}`):
  - Are significantly more expensive than parsing a regular URL path segment.
  - Result in many more substrings being allocated.
  - The complex segment logic was not updated in ASP.NET Core 3.0 routing performance update.
- Synchronous data access: Many complex apps have database access as part of their routing. ASP.NET Core 2.2 and earlier routing might not provide the right extensibility points to support database access routing. For example, IRouteConstraint, and IActionConstraint are synchronous. Extensibility points such as MatcherPolicy and EndpointSelectorContext are asynchronous.

Guidance for library authors

This section contains guidance for library authors building on top of routing. These details are intended to

ensure that app developers have a good experience using libraries and frameworks that extend routing.

Define endpoints

To create a framework that uses routing for URL matching, start by defining a user experience that builds on top of UseEndpoints.

DO build on top of IEndpointRouteBuilder. This allows users to compose your framework with other ASP.NET Core features without confusion. Every ASP.NET Core template includes routing. Assume routing is present and familiar for users.

```
app.UseEndpoints(endpoints =>
{
    // Your framework
    endpoints.MapMyFramework(...);

    endpoints.MapHealthChecks("/healthz");
});
```

DO return a sealed concrete type from a call to `MapMyFramework(...)` that implements IEndpointConventionBuilder. Most framework `Map...` methods follow this pattern. The `IEndpointConventionBuilder` interface:

- Allows composability of metadata.
- Is targeted by a variety of extension methods.

Declaring your own type allows you to add your own framework-specific functionality to the builder. It's ok to wrap a framework-declared builder and forward calls to it.

```
app.UseEndpoints(endpoints =>
{
    // Your framework
    endpoints.MapMyFramework(...).RequireAuthorization()
                                 .WithMyFrameworkFeature(awesome: true);

    endpoints.MapHealthChecks("/healthz");
});
```

CONSIDER writing your own EndpointDataSource. `EndpointDataSource` is the low-level primitive for declaring and updating a collection of endpoints. `EndpointDataSource` is a powerful API used by controllers and Razor Pages.

The routing tests have a basic example of a non-updating data source.

DO NOT attempt to register an `EndpointDataSource` by default. Require users to register your framework in UseEndpoints. The philosophy of routing is that nothing is included by default, and that `UseEndpoints` is the place to register endpoints.

Creating routing-integrated middleware

CONSIDER defining metadata types as an interface.

DO make it possible to use metadata types as an attribute on classes and methods.

```
public interface ICoolMetadata
{
    bool IsCool { get; }
}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class CoolMetadataAttribute : Attribute, ICoolMetadata
{
    public bool IsCool => true;
}
```

Frameworks like controllers and Razor Pages support applying metadata attributes to types and methods. If you declare metadata types:

- Make them accessible as attributes.
- Most users are familiar with applying attributes.

Declaring a metadata type as an interface adds another layer of flexibility:

- Interfaces are composable.
- Developers can declare their own types that combine multiple policies.

DO make it possible to override metadata, as shown in the following example:

```
public interface ICoolMetadata
{
    bool IsCool { get; }
}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class CoolMetadataAttribute : Attribute, ICoolMetadata
{
    public bool IsCool => true;
}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class SuppressCoolMetadataAttribute : Attribute, ICoolMetadata
{
    public bool IsCool => false;
}

[CoolMetadata]
public class MyController : Controller
{
    public void MyCool() { }

    [SuppressCoolMetadata]
    public void Uncool() { }
}
```

The best way to follow these guidelines is to avoid defining **marker metadata**:

- Don't just look for the presence of a metadata type.
- Define a property on the metadata and check the property.

The metadata collection is ordered and supports overriding by priority. In the case of controllers, metadata on the action method is most specific.

DO make middleware useful with and without routing.

```
app.UseRouting();

app.UseAuthorization(new AuthorizationPolicy() { ... });

app.UseEndpoints(endpoints =>
{
    // Your framework
    endpoints.MapMyFramework(...).RequireAuthorization();
});
```

As an example of this guideline, consider the `UseAuthorization` middleware. The authorization middleware allows you to pass in a fallback policy. The fallback policy, if specified, applies to both:

- Endpoints without a specified policy.
- Requests that don't match an endpoint.

This makes the authorization middleware useful outside of the context of routing. The authorization middleware can be used for traditional middleware programming.

Debug diagnostics

For detailed routing diagnostic output, set `Logging:LogLevel:Microsoft` to `Debug`. In the development environment, set the log level in *appsettings.Development.json*:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Debug",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Routing is responsible for mapping request URIs to endpoints and dispatching incoming requests to those endpoints. Routes are defined in the app and configured when the app starts. A route can optionally extract values from the URL contained in the request, and these values can then be used for request processing. Using route information from the app, routing is also able to generate URLs that map to endpoints.

To use the latest routing scenarios in ASP.NET Core 2.2, specify the compatibility version to the MVC services registration in `Startup.ConfigureServices`:

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

The EnableEndpointRouting option determines if routing should internally use endpoint-based logic or the IRouter-based logic of ASP.NET Core 2.1 or earlier. When the compatibility version is set to 2.2 or later, the default value is `true`. Set the value to `false` to use the prior routing logic:

```
// Use the routing logic of ASP.NET Core 2.1 or earlier:
services.AddMvc(options => options.EnableEndpointRouting = false)
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

For more information on IRouter-based routing, see the .

> **IMPORTANT**
>
> This document covers low-level ASP.NET Core routing. For information on ASP.NET Core MVC routing, see Routing to controller actions in ASP.NET Core. For information on routing conventions in Razor Pages, see Razor Pages route and app conventions in ASP.NET Core.

View or download sample code (how to download)

Routing basics

Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful. The default conventional route `{controller=Home}/{action=Index}/{id?}`:

- Supports a basic and descriptive routing scheme.
- Is a useful starting point for UI-based apps.

Developers commonly add additional terse routes to high-traffic areas of an app in specialized situations using attribute routing or dedicated conventional routes. Specialized situations examples include, blog and ecommerce endpoints.

Web APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs. This means that many operations, for example, GET, and POST, on the same logical resource use the same URL. Attribute routing provides a level of control that's needed to carefully design an API's public endpoint layout.

Razor Pages apps use default conventional routing to serve named resources in the *Pages* folder of an app. Additional conventions are available that allow you to customize Razor Pages routing behavior. For more information, see Introduction to Razor Pages in ASP.NET Core and Razor Pages route and app conventions in ASP.NET Core.

URL generation support allows the app to be developed without hard-coding URLs to link the app together. This support allows for starting with a basic routing configuration and modifying the routes after the app's resource layout is determined.

Routing uses *endpoints* (`Endpoint`) to represent logical endpoints in an app.

An endpoint defines a delegate to process requests and a collection of arbitrary metadata. The metadata is used implement cross-cutting concerns based on policies and configuration attached to each endpoint.

The routing system has the following characteristics:

- Route template syntax is used to define routes with tokenized route parameters.
- Conventional-style and attribute-style endpoint configuration is permitted.
- IRouteConstraint is used to determine whether a URL parameter contains a valid value for a given endpoint constraint.
- App models, such as MVC/Razor Pages, register all of their endpoints, which have a predictable implementation of routing scenarios.
- The routing implementation makes routing decisions wherever desired in the middleware pipeline.
- Middleware that appears after a Routing Middleware can inspect the result of the Routing Middleware's endpoint decision for a given request URI.
- It's possible to enumerate all of the endpoints in the app anywhere in the middleware pipeline.

- An app can use routing to generate URLs (for example, for redirection or links) based on endpoint information and thus avoid hard-coded URLs, which helps maintainability.
- URL generation is based on addresses, which support arbitrary extensibility:
  - The Link Generator API (LinkGenerator) can be resolved anywhere using dependency injection (DI) to generate URLs.
  - Where the Link Generator API isn't available via DI, IUrlHelper offers methods to build URLs.

> **NOTE**
> With the release of endpoint routing in ASP.NET Core 2.2, endpoint linking is limited to MVC/Razor Pages actions and pages. The expansions of endpoint-linking capabilities is planned for future releases.

Routing is connected to the middleware pipeline by the RouterMiddleware class. ASP.NET Core MVC adds routing to the middleware pipeline as part of its configuration and handles routing in MVC and Razor Pages apps. To learn how to use routing as a standalone component, see the Use Routing Middleware section.

URL matching

URL matching is the process by which routing dispatches an incoming request to an *endpoint*. This process is based on data in the URL path but can be extended to consider any data in the request. The ability to dispatch requests to separate handlers is key to scaling the size and complexity of an app.

The routing system in endpoint routing is responsible for all dispatching decisions. Since the middleware applies policies based on the selected endpoint, it's important that any decision that can affect dispatching or the application of security policies is made inside the routing system.

When the endpoint delegate is executed, the properties of RouteContext.RouteData are set to appropriate values based on the request processing performed thus far.

RouteData.Values is a dictionary of *route values* produced from the route. These values are usually determined by tokenizing the URL and can be used to accept user input or to make further dispatching decisions inside the app.

RouteData.DataTokens is a property bag of additional data related to the matched route. DataTokens are provided to support associating state data with each route so that the app can make decisions based on which route matched. These values are developer-defined and do **not** affect the behavior of routing in any way. Additionally, values stashed in RouteData.DataTokens can be of any type, in contrast to RouteData.Values, which must be convertible to and from strings.

RouteData.Routers is a list of the routes that took part in successfully matching the request. Routes can be nested inside of one another. The Routers property reflects the path through the logical tree of routes that resulted in a match. Generally, the first item in Routers is the route collection and should be used for URL generation. The last item in Routers is the route handler that matched.

URL generation with LinkGenerator

URL generation is the process by which routing can create a URL path based on a set of route values. This allows for a logical separation between your endpoints and the URLs that access them.

Endpoint routing includes the Link Generator API (LinkGenerator). LinkGenerator is a singleton service that can be retrieved from DI. The API can be used outside of the context of an executing request. MVC's IUrlHelper and scenarios that rely on IUrlHelper, such as Tag Helpers, HTML Helpers, and Action Results, use the link generator to provide link generating capabilities.

The link generator is backed by the concept of an *address* and *address schemes*. An address scheme is a way of determining the endpoints that should be considered for link generation. For example, the route name and

route values scenarios many users are familiar with from MVC/Razor Pages are implemented as an address scheme.

The link generator can link to MVC/Razor Pages actions and pages via the following extension methods:

- GetPathByAction
- GetUriByAction
- GetPathByPage
- GetUriByPage

An overload of these methods accepts arguments that include the `HttpContext`. These methods are functionally equivalent to `Url.Action` and `Url.Page` but offer additional flexibility and options.

The `GetPath*` methods are most similar to `Url.Action` and `Url.Page` in that they generate a URI containing an absolute path. The `GetUri*` methods always generate an absolute URI containing a scheme and host. The methods that accept an `HttpContext` generate a URI in the context of the executing request. The ambient route values, URL base path, scheme, and host from the executing request are used unless overridden.

LinkGenerator is called with an address. Generating a URI occurs in two steps:

1. An address is bound to a list of endpoints that match the address.
2. Each endpoint's `RoutePattern` is evaluated until a route pattern that matches the supplied values is found. The resulting output is combined with the other URI parts supplied to the link generator and returned.

The methods provided by LinkGenerator support standard link generation capabilities for any type of address. The most convenient way to use the link generator is through extension methods that perform operations for a specific address type.

EXTENSION METHOD	DESCRIPTION
GetPathByAddress	Generates a URI with an absolute path based on the provided values.
GetUriByAddress	Generates an absolute URI based on the provided values.

WARNING

Pay attention to the following implications of calling LinkGenerator methods:

- Use `GetUri*` extension methods with caution in an app configuration that doesn't validate the `Host` header of incoming requests. If the `Host` header of incoming requests isn't validated, untrusted request input can be sent back to the client in URIs in a view/page. We recommend that all production apps configure their server to validate the `Host` header against known valid values.
- Use LinkGenerator with caution in middleware in combination with `Map` or `MapWhen`. `Map*` changes the base path of the executing request, which affects the output of link generation. All of the LinkGenerator APIs allow specifying a base path. Always specify an empty base path to undo `Map*`'s affect on link generation.

Differences from earlier versions of routing

A few differences exist between endpoint routing in ASP.NET Core 2.2 or later and earlier versions of routing in ASP.NET Core:

- The endpoint routing system doesn't support IRouter-based extensibility, including inheriting from Route.

- Endpoint routing doesn't support WebApiCompatShim. Use the 2.1 compatibility version (`.SetCompatibilityVersion(CompatibilityVersion.Version_2_1)`) to continue using the compatibility shim.

- Endpoint Routing has different behavior for the casing of generated URIs when using conventional routes.

  Consider the following default route template:

  ```
  app.UseMvc(routes =>
  {
      routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
  });
  ```

  Suppose you generate a link to an action using the following route:

  ```
  var link = Url.Action("ReadPost", "blog", new { id = 17, });
  ```

  With IRouter-based routing, this code generates a URI of `/blog/ReadPost/17`, which respects the casing of the provided route value. Endpoint routing in ASP.NET Core 2.2 or later produces `/Blog/ReadPost/17` ("Blog" is capitalized). Endpoint routing provides the `IOutboundParameterTransformer` interface that can be used to customize this behavior globally or to apply different conventions for mapping URLs.

  For more information, see the Parameter transformer reference section.

- Link Generation used by MVC/Razor Pages with conventional routes behaves differently when attempting to link to an controller/action or page that doesn't exist.

  Consider the following default route template:

  ```
  app.UseMvc(routes =>
  {
      routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
  });
  ```

  Suppose you generate a link to an action using the default template with the following:

  ```
  var link = Url.Action("ReadPost", "Blog", new { id = 17, });
  ```

  With `IRouter`-based routing, the result is always `/Blog/ReadPost/17`, even if the `BlogController` doesn't exist or doesn't have a `ReadPost` action method. As expected, endpoint routing in ASP.NET Core 2.2 or later produces `/Blog/ReadPost/17` if the action method exists. *However, endpoint routing produces an empty string if the action doesn't exist.* Conceptually, endpoint routing doesn't assume that the endpoint exists if the action doesn't exist.

- The link generation *ambient value invalidation algorithm* behaves differently when used with endpoint routing.

  Ambient value invalidation is the algorithm that decides which route values from the currently executing request (the ambient values) can be used in link generation operations. Conventional routing always invalidated extra route values when linking to a different action. Attribute routing didn't have this behavior prior to the release of ASP.NET Core 2.2. In earlier versions of ASP.NET Core, links to another action that use the same route parameter names resulted in link generation errors. In ASP.NET Core 2.2 or later, both forms of routing invalidate values when linking to another action.

Consider the following example in ASP.NET Core 2.1 or earlier. When linking to another action (or another page), route values can be reused in undesirable ways.

In */Pages/Store/Product.cshtml*:

```
@page "{id}"
@Url.Page("/Login")
```

In */Pages/Login.cshtml*:

```
@page "{id?}"
```

If the URI is `/Store/Product/18` in ASP.NET Core 2.1 or earlier, the link generated on the Store/Info page by `@Url.Page("/Login")` is `/Login/18`. The `id` value of 18 is reused, even though the link destination is different part of the app entirely. The `id` route value in the context of the `/Login` page is probably a user ID value, not a store product ID value.

In endpoint routing with ASP.NET Core 2.2 or later, the result is `/Login`. Ambient values aren't reused when the linked destination is a different action or page.

- Round-tripping route parameter syntax: Forward slashes aren't encoded when using a double-asterisk (`**`) catch-all parameter syntax.

  During link generation, the routing system encodes the value captured in a double-asterisk (`**`) catch-all parameter (for example, `{**myparametername}`) except the forward slashes. The double-asterisk catch-all is supported with `IRouter`-based routing in ASP.NET Core 2.2 or later.

  The single asterisk catch-all parameter syntax in prior versions of ASP.NET Core (`{*myparametername}`) remains supported, and forward slashes are encoded.

ROUTE	LINK GENERATED WITH `URL.ACTION(NEW { CATEGORY = "ADMIN/PRODUCTS" })`...
`/search/{*page}`	`/search/admin%2Fproducts` (the forward slash is encoded)
`/search/{**page}`	`/search/admin/products`

Middleware example

In the following example, a middleware uses the LinkGenerator API to create link to an action method that lists store products. Using the link generator by injecting it into a class and calling `GenerateLink` is available to any class in an app.

```
using Microsoft.AspNetCore.Routing;

public class ProductsLinkMiddleware
{
    private readonly LinkGenerator _linkGenerator;

    public ProductsLinkMiddleware(RequestDelegate next, LinkGenerator linkGenerator)
    {
        _linkGenerator = linkGenerator;
    }

    public async Task InvokeAsync(HttpContext httpContext)
    {
        var url = _linkGenerator.GetPathByAction("ListProducts", "Store");

        httpContext.Response.ContentType = "text/plain";

        await httpContext.Response.WriteAsync($"Go to {url} to see our products.");
    }
}
```

Create routes

Most apps create routes by calling MapRoute or one of the similar extension methods defined on IRouteBuilder. Any of the IRouteBuilder extension methods create an instance of Route and add it to the route collection.

MapRoute doesn't accept a route handler parameter. MapRoute only adds routes that are handled by the DefaultHandler. To learn more about routing in MVC, see Routing to controller actions in ASP.NET Core.

The following code example is an example of a MapRoute call used by a typical ASP.NET Core MVC route definition:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

This template matches a URL path and extracts the route values. For example, the path `/Products/Details/17` generates the following route values: `{ controller = Products, action = Details, id = 17 }`.

Route values are determined by splitting the URL path into segments and matching each segment with the *route parameter* name in the route template. Route parameters are named. The parameters defined by enclosing the parameter name in braces `{ ... }`.

The preceding template could also match the URL path `/` and produce the values `{ controller = Home, action = Index }`. This occurs because the `{controller}` and `{action}` route parameters have default values and the `id` route parameter is optional. An equals sign (`=`) followed by a value after the route parameter name defines a default value for the parameter. A question mark (`?`) after the route parameter name defines an optional parameter.

Route parameters with a default value *always* produce a route value when the route matches. Optional parameters don't produce a route value if there is no corresponding URL path segment. See the Route template reference section for a thorough description of route template scenarios and syntax.

In the following example, the route parameter definition `{id:int}` defines a route constraint for the `id` route parameter:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id:int}");
```

This template matches a URL path like `/Products/Details/17` but not `/Products/Details/Apples`. Route constraints implement IRouteConstraint and inspect route values to verify them. In this example, the route value `id` must be convertible to an integer. See route-constraint-reference for an explanation of route constraints provided by the framework.

Additional overloads of MapRoute accept values for `constraints`, `dataTokens`, and `defaults`. The typical usage of these parameters is to pass an anonymously typed object, where the property names of the anonymous type match route parameter names.

The following MapRoute examples create equivalent routes:

```
routes.MapRoute(
    name: "default_route",
    template: "{controller}/{action}/{id?}",
    defaults: new { controller = "Home", action = "Index" });

routes.MapRoute(
    name: "default_route",
    template: "{controller=Home}/{action=Index}/{id?}");
```

TIP

The inline syntax for defining constraints and defaults can be convenient for simple routes. However, there are scenarios, such as data tokens, that aren't supported by inline syntax.

The following example demonstrates a few additional scenarios:

```
routes.MapRoute(
    name: "blog",
    template: "Blog/{**article}",
    defaults: new { controller = "Blog", action = "ReadArticle" });
```

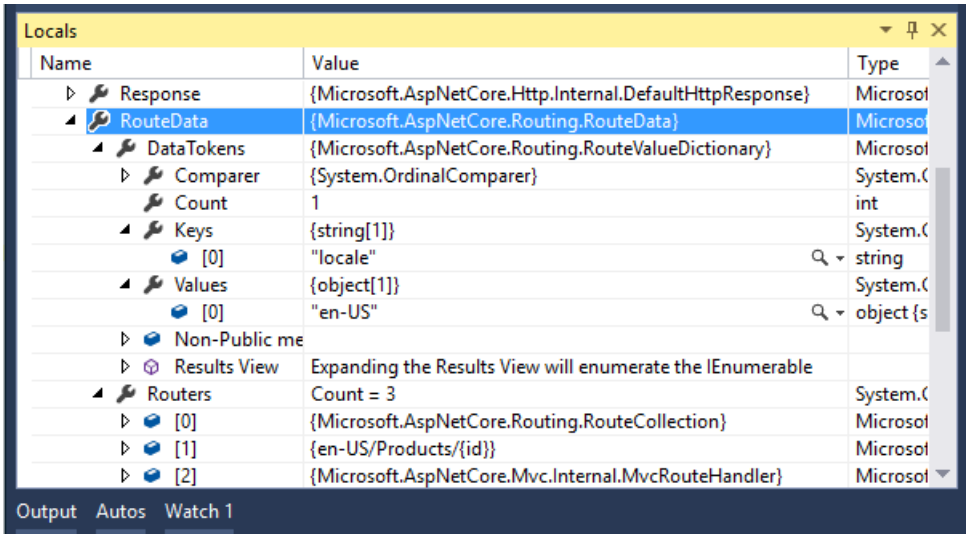
The preceding template matches a URL path like `/Blog/All-About-Routing/Introduction` and extracts the values `{ controller = Blog, action = ReadArticle, article = All-About-Routing/Introduction }`. The default route values for `controller` and `action` are produced by the route even though there are no corresponding route parameters in the template. Default values can be specified in the route template. The `article` route parameter is defined as a *catch-all* by the appearance of an double asterisk (`**`) before the route parameter name. Catch-all route parameters capture the remainder of the URL path and can also match the empty string.

The following example adds route constraints and data tokens:

```
routes.MapRoute(
    name: "us_english_products",
    template: "en-US/Products/{id}",
    defaults: new { controller = "Products", action = "Details" },
    constraints: new { id = new IntRouteConstraint() },
    dataTokens: new { locale = "en-US" });
```

The preceding template matches a URL path like `/en-US/Products/5` and extracts the values

`{ controller = Products, action = Details, id = 5 }` and the data tokens `{ locale = en-US }`.



Route class URL generation

The Route class can also perform URL generation by combining a set of route values with its route template. This is logically the reverse process of matching the URL path.

> **TIP**
>
> To better understand URL generation, imagine what URL you want to generate and then think about how a route template would match that URL. What values would be produced? This is the rough equivalent of how URL generation works in the Route class.

The following example uses a general ASP.NET Core MVC default route:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

With the route values `{ controller = Products, action = List }`, the URL `/Products/List` is generated. The route values are substituted for the corresponding route parameters to form the URL path. Since `id` is an optional route parameter, the URL is successfully generated without a value for `id`.

With the route values `{ controller = Home, action = Index }`, the URL `/` is generated. The provided route values match the default values, and the segments corresponding to the default values are safely omitted.

Both URLs generated round-trip with the following route definition (`/Home/Index` and `/`) produce the same route values that were used to generate the URL.

> **NOTE**
>
> An app using ASP.NET Core MVC should use UrlHelper to generate URLs instead of calling into routing directly.

For more information on URL generation, see the Url generation reference section.

Use Routing Middleware

Reference the Microsoft.AspNetCore.App metapackage in the app's project file.

Add routing to the service container in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRouting();
}
```

Routes must be configured in the `Startup.Configure` method. The sample app uses the following APIs:

- RouteBuilder
- MapGet: Matches only HTTP GET requests.
- UseRouter

```
var trackPackageRouteHandler = new RouteHandler(context =>
{
    var routeValues = context.GetRouteData().Values;
    return context.Response.WriteAsync(
        $"Hello! Route values: {string.Join(", ", routeValues)}");
});

var routeBuilder = new RouteBuilder(app, trackPackageRouteHandler);

routeBuilder.MapRoute(
    "Track Package Route",
    "package/{operation:regex(^track|create$)}/{id:int}");

routeBuilder.MapGet("hello/{name}", context =>
{
    var name = context.GetRouteValue("name");
    // The route handler when HTTP GET "hello/<anything>" matches
    // To match HTTP GET "hello/<anything>/<anything>,
    // use routeBuilder.MapGet("hello/{*name}"
    return context.Response.WriteAsync($"Hi, {name}!");
});

var routes = routeBuilder.Build();
app.UseRouter(routes);
```

The following table shows the responses with the given URIs.

URI	RESPONSE
`/package/create/3`	Hello! Route values: [operation, create], [id, 3]
`/package/track/-3`	Hello! Route values: [operation, track], [id, -3]
`/package/track/-3/`	Hello! Route values: [operation, track], [id, -3]
`/package/track/`	The request falls through, no match.
`GET /hello/Joe`	Hi, Joe!
`POST /hello/Joe`	The request falls through, matches HTTP GET only.
`GET /hello/Joe/Smith`	The request falls through, no match.

The framework provides a set of extension methods for creating routes (RequestDelegateRouteBuilderExtensions):

- MapDelete
- MapGet
- MapMiddlewareDelete
- MapMiddlewareGet
- MapMiddlewarePost
- MapMiddlewarePut
- MapMiddlewareRoute
- MapMiddlewareVerb
- MapPost
- MapPut
- MapRoute
- MapVerb

The `Map[Verb]` methods use constraints to limit the route to the HTTP Verb in the method name. For example, see MapGet and MapVerb.

Route template reference

Tokens within curly braces (`{ ... }`) define *route parameters* that are bound if the route is matched. You can define more than one route parameter in a route segment, but they must be separated by a literal value. For example, `{controller=Home}{action=Index}` isn't a valid route, since there's no literal value between `{controller}` and `{action}`. These route parameters must have a name and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URLs path. To match a literal route parameter delimiter (`{` or `}`), escape the delimiter by repeating the character (`{{` or `}}`).

URL patterns that attempt to capture a file name with an optional file extension have additional considerations. For example, consider the template `files/{filename}.{ext?}` . When values for both `filename` and `ext` exist, both values are populated. If only a value for `filename` exists in the URL, the route matches because the trailing period (`.`) is optional. The following URLs match this route:

- `/files/myFile.txt`
- `/files/myFile`

You can use an asterisk (`*`) or double asterisk (`**`) as a prefix to a route parameter to bind to the rest of the URI. These are called a *catch-all* parameters. For example, `blog/{**slug}` matches any URI that starts with `/blog` and has any value following it, which is assigned to the `slug` route value. Catch-all parameters can also match the empty string.

The catch-all parameter escapes the appropriate characters when the route is used to generate a URL, including path separator (`/`) characters. For example, the route `foo/{*path}` with route values `{ path = "my/path" }` generates `foo/my%2Fpath` . Note the escaped forward slash. To round-trip path separator characters, use the `**` route parameter prefix. The route `foo/{**path}` with `{ path = "my/path" }` generates `foo/my/path` .

Route parameters may have *default values* designated by specifying the default value after the parameter name separated by an equals sign (`=`). For example, `{controller=Home}` defines `Home` as the default value for `controller` . The default value is used if no value is present in the URL for the parameter. Route parameters are made optional by appending a question mark (`?`) to the end of the parameter name, as in `id?` . The difference between optional values and default route parameters is that a route parameter with a

default value always produces a value—an optional parameter has a value only when a value is provided by the request URL.

Route parameters may have constraints that must match the route value bound from the URL. Adding a colon (`:`) and constraint name after the route parameter name specifies an *inline constraint* on a route parameter. If the constraint requires arguments, they're enclosed in parentheses (`(...)`) after the constraint name. Multiple inline constraints can be specified by appending another colon (`:`) and constraint name.

The constraint name and arguments are passed to the IInlineConstraintResolver service to create an instance of IRouteConstraint to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies a `minlength` constraint with the argument `10`. For more information on route constraints and a list of the constraints provided by the framework, see the Route constraint reference section.

Route parameters may also have parameter transformers, which transform a parameter's value when generating links and matching actions and pages to URLs. Like constraints, parameter transformers can be added inline to a route parameter by adding a colon (`:`) and transformer name after the route parameter name. For example, the route template `blog/{article:slugify}` specifies a `slugify` transformer. For more information on parameter transformers, see the Parameter transformer reference section.

The following table demonstrates example route templates and their behavior.

ROUTE TEMPLATE	EXAMPLE MATCHING URI	THE REQUEST URI...
`hello`	`/hello`	Only matches the single path `/hello`.
`{Page=Home}`	`/`	Matches and sets `Page` to `Home`.
`{Page=Home}`	`/Contact`	Matches and sets `Page` to `Contact`.
`{controller}/{action}/{id?}`	`/Products/List`	Maps to the `Products` controller and `List` action.
`{controller}/{action}/{id?}`	`/Products/Details/123`	Maps to the `Products` controller and `Details` action (`id` set to 123).
`{controller=Home}/{action=Index}/{id?}`	`/`	Maps to the `Home` controller and `Index` method (`id` is ignored).

Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

> **TIP**
> Enable Logging to see how the built-in routing implementations, such as Route, match requests.

Reserved routing names

The following keywords are reserved names and can't be used as route names or parameters:

- `action`
- `area`

- `controller`
- `handler`
- `page`

Route constraint reference

Route constraints execute when a match has occurred to the incoming URL and the URL path is tokenized into route values. Route constraints generally inspect the route value associated via the route template and make a yes/no decision about whether or not the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the HttpMethodRouteConstraint can accept or reject a request based on its HTTP verb. Constraints are used in routing requests and link generation.

> **WARNING**
> Don't use constraints for **input validation**. If constraints are used for **input validation**, invalid input results in a *404 - Not Found* response instead of a *400 - Bad Request* with an appropriate error message. Route constraints are used to **disambiguate** similar routes, not to validate the inputs for a particular route.

The following table demonstrates example route constraints and their expected behavior.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`int`	`{id:int}`	`123456789`, `-123456789`	Matches any integer.
`bool`	`{active:bool}`	`true`, `FALSE`	Matches `true` or `false`. Case-insensitive.
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Matches a valid `DateTime` value in the invariant culture. See preceding warning.
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Matches a valid `decimal` value in the invariant culture. See preceding warning.
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Matches a valid `double` value in the invariant culture. See preceding warning.
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Matches a valid `float` value in the invariant culture. See preceding warning.
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	Matches a valid `Guid` value.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`long`	`{ticks:long}`	`123456789`, `-123456789`	Matches a valid `long` value.
`minlength(value)`	`{username:minlength(4)}`	`Rick`	String must be at least 4 characters.
`maxlength(value)`	`{filename:maxlength(8)}`	`MyFile`	String has maximum of 8 characters.
`length(length)`	`{filename:length(12)}`	`somefile.txt`	String must be exactly 12 characters long.
`length(min,max)`	`{filename:length(8,16)}`	`somefile.txt`	String must be at least 8 and has maximum of 16 characters.
`min(value)`	`{age:min(18)}`	`19`	Integer value must be at least 18.
`max(value)`	`{age:max(120)}`	`91`	Integer value maximum of 120.
`range(min,max)`	`{age:range(18,120)}`	`91`	Integer value must be at least 18 and maximum of 120.
`alpha`	`{name:alpha}`	`Rick`	String must consist of one or more alphabetical characters `a`-`z`. Case-insensitive.
`regex(expression)`	`{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}`	`123-45-6789`	String must match the regular expression. See tips about defining a regular expression.
`required`	`{name:required}`	`Rick`	Used to enforce that a non-parameter value is present during URL generation.

Multiple, colon-delimited constraints can be applied to a single parameter. For example, the following constraint restricts a parameter to an integer value of 1 or greater:

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { }
```

WARNING

Route constraints that verify the URL and are converted to a CLR type (such as `int` or `DateTime`) always use the invariant culture. These constraints assume that the URL is non-localizable. The framework-provided route constraints don't modify the values stored in route values. All route values parsed from the URL are stored as strings. For example, the `float` constraint attempts to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

Regular expressions

The ASP.NET Core framework adds `RegexOptions.IgnoreCase | RegexOptions.Compiled | RegexOptions.CultureInvariant` to the regular expression constructor. See RegexOptions for a description of these members.

Regular expressions use delimiters and tokens similar to those used by routing and the C# language. Regular expression tokens must be escaped. To use the regular expression `^\d{3}-\d{2}-\d{4}$` in routing:

- The expression must have the single backslash `\` characters provided in the string as double backslash `\\` characters in the source code.
- The regular expression must us `\\` in order to escape the `\` string escape character.
- The regular expression doesn't require `\\` when using verbatim string literals.

To escape routing parameter delimiter characters `{`, `}`, `[`, `]`, double the characters in the expression `{{`, `}}`, `[[`, `]]`. The following table shows a regular expression and the escaped version:

REGULAR EXPRESSION	ESCAPED REGULAR EXPRESSION
`^\d{3}-\d{2}-\d{4}$`	`^\\d{{3}}-\\d{{2}}-\\d{{4}}$`
`^[a-z]{2}$`	`^[[a-z]]{{2}}$`

Regular expressions used in routing often start with the caret `^` character and match starting position of the string. The expressions often end with the dollar sign `$` character and match end of the string. The `^` and `$` characters ensure that the regular expression match the entire route parameter value. Without the `^` and `$` characters, the regular expression match any substring within the string, which is often undesirable. The following table provides examples and explains why they match or fail to match.

EXPRESSION	STRING	MATCH	COMMENT
`[a-z]{2}`	hello	Yes	Substring matches
`[a-z]{2}`	123abc456	Yes	Substring matches
`[a-z]{2}`	mz	Yes	Matches expression
`[a-z]{2}`	MZ	Yes	Not case sensitive
`^[a-z]{2}$`	hello	No	See `^` and `$` above
`^[a-z]{2}$`	123abc456	No	See `^` and `$` above

For more information on regular expression syntax, see .NET Framework Regular Expressions.

To constrain a parameter to a known set of possible values, use a regular expression. For example, `{action:regex(^(list|get|create)$)}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string `^(list|get|create)$` is equivalent. Constraints that are passed in the constraints dictionary (not inline within a template) that don't match one of the known constraints are also treated as regular expressions.

Custom route constraints

In addition to the built-in route constraints, custom route constraints can be created by implementing the IRouteConstraint interface. The IRouteConstraint interface contains a single method, `Match`, which returns `true` if the constraint is satisfied and `false` otherwise.

To use a custom IRouteConstraint, the route constraint type must be registered with the app's ConstraintMap in the app's service container. A ConstraintMap is a dictionary that maps route constraint keys to IRouteConstraint implementations that validate those constraints. An app's ConstraintMap can be updated in `Startup.ConfigureServices` either as part of a services.AddRouting call or by configuring RouteOptions directly with `services.Configure<RouteOptions>`. For example:

```
services.AddRouting(options =>
{
    options.ConstraintMap.Add("customName", typeof(MyCustomConstraint));
});
```

The constraint can then be applied to routes in the usual manner, using the name specified when registering the constraint type. For example:

```
[HttpGet("{id:customName}")]
public ActionResult<string> Get(string id)
```

Parameter transformer reference

Parameter transformers:

- Execute when generating a link for a Route.
- Implement `Microsoft.AspNetCore.Routing.IOutboundParameterTransformer`.
- Are configured using ConstraintMap.
- Take the parameter's route value and transform it to a new string value.
- Result in using the transformed value in the generated link.

For example, a custom `slugify` parameter transformer in route pattern `blog\{article:slugify}` with `Url.Action(new { article = "MyTestArticle" })` generates `blog\my-test-article`.

To use a parameter transformer in a route pattern, configure it first using ConstraintMap in `Startup.ConfigureServices`:

```
services.AddRouting(options =>
{
    // Replace the type and the name used to refer to it with your own
    // IOutboundParameterTransformer implementation
    options.ConstraintMap["slugify"] = typeof(SlugifyParameterTransformer);
});
```

Parameter transformers are used by the framework to transform the URI where an endpoint resolves. For example, ASP.NET Core MVC uses parameter transformers to transform the route value used to match an

`area`, `controller`, `action`, and `page`.

```
routes.MapRoute(
    name: "default",
    template: "{controller:slugify=Home}/{action:slugify=Index}/{id?}");
```

With the preceding route, the action `SubscriptionManagementController.GetAll` is matched with the URI `/subscription-management/get-all`. A parameter transformer doesn't change the route values used to generate a link. For example, `Url.Action("GetAll", "SubscriptionManagement")` outputs `/subscription-management/get-all`.

ASP.NET Core provides API conventions for using a parameter transformers with generated routes:

- ASP.NET Core MVC has the `Microsoft.AspNetCore.Mvc.ApplicationModels.RouteTokenTransformerConvention` API convention. This convention applies a specified parameter transformer to all attribute routes in the app. The parameter transformer transforms attribute route tokens as they are replaced. For more information, see Use a parameter transformer to customize token replacement.
- Razor Pages has the `Microsoft.AspNetCore.Mvc.ApplicationModels.PageRouteTransformerConvention` API convention. This convention applies a specified parameter transformer to all automatically discovered Razor Pages. The parameter transformer transforms the folder and file name segments of Razor Pages routes. For more information, see Use a parameter transformer to customize page routes.

URL generation reference

The following example shows how to generate a link to a route given a dictionary of route values and a RouteCollection.

```
app.Run(async (context) =>
{
    var dictionary = new RouteValueDictionary
    {
        { "operation", "create" },
        { "id", 123}
    };

    var vpc = new VirtualPathContext(context, null, dictionary,
        "Track Package Route");
    var path = routes.GetVirtualPath(vpc).VirtualPath;

    context.Response.ContentType = "text/html";
    await context.Response.WriteAsync("Menu<hr/>");
    await context.Response.WriteAsync(
        $"<a href='{path}'>Create Package 123</a><br/>");
});
```

The VirtualPath generated at the end of the preceding sample is `/package/create/123`. The dictionary supplies the `operation` and `id` route values of the "Track Package Route" template, `package/{operation}/{id}`. For details, see the sample code in the Use Routing Middleware section or the sample app.

The second parameter to the VirtualPathContext constructor is a collection of *ambient values*. Ambient values are convenient to use because they limit the number of values a developer must specify within a request context. The current route values of the current request are considered ambient values for link generation. In an ASP.NET Core MVC app's `About` action of the `HomeController`, you don't need to specify the controller route value to link to the `Index` action—the ambient value of `Home` is used.

Ambient values that don't match a parameter are ignored. Ambient values are also ignored when an explicitly

provided value overrides the ambient value. Matching occurs from left to right in the URL.

Values explicitly provided but that don't match a segment of the route are added to the query string. The following table shows the result when using the route template `{controller}/{action}/{id?}`.

AMBIENT VALUES	EXPLICIT VALUES	RESULT
controller = "Home"	action = "About"	`/Home/About`
controller = "Home"	controller = "Order", action = "About"	`/Order/About`
controller = "Home", color = "Red"	action = "About"	`/Home/About`
controller = "Home"	action = "About", color = "Red"	`/Home/About?color=Red`

If a route has a default value that doesn't correspond to a parameter and that value is explicitly provided, it must match the default value:

```
routes.MapRoute("blog_route", "blog/{*slug}",
    defaults: new { controller = "Blog", action = "ReadPost" });
```

Link generation only generates a link for this route when the matching values for `controller` and `action` are provided.

Complex segments

Complex segments (for example `[Route("/x{token}y")]`) are processed by matching up literals from right to left in a non-greedy way. See this code for a detailed explanation of how complex segments are matched. The code sample is not used by ASP.NET Core, but it provides a good explanation of complex segments.

Routing is responsible for mapping request URIs to route handlers and dispatching an incoming requests. Routes are defined in the app and configured when the app starts. A route can optionally extract values from the URL contained in the request, and these values can then be used for request processing. Using configured routes from the app, routing is able to generate URLs that map to route handlers.

To use the latest routing scenarios in ASP.NET Core 2.1, specify the compatibility version to the MVC services registration in `Startup.ConfigureServices`:

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

IMPORTANT

This document covers low-level ASP.NET Core routing. For information on ASP.NET Core MVC routing, see Routing to controller actions in ASP.NET Core. For information on routing conventions in Razor Pages, see Razor Pages route and app conventions in ASP.NET Core.

View or download sample code (how to download)

Routing basics

Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful.

The default conventional route `{controller=Home}/{action=Index}/{id?}`:

- Supports a basic and descriptive routing scheme.
- Is a useful starting point for UI-based apps.

Developers commonly add additional terse routes to high-traffic areas of an app in specialized situations (for example, blog and ecommerce endpoints) using attribute routing or dedicated conventional routes.

Web APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs. This means that many operations (for example, GET, POST) on the same logical resource will use the same URL. Attribute routing provides a level of control that's needed to carefully design an API's public endpoint layout.

Razor Pages apps use default conventional routing to serve named resources in the *Pages* folder of an app. Additional conventions are available that allow you to customize Razor Pages routing behavior. For more information, see Introduction to Razor Pages in ASP.NET Core and Razor Pages route and app conventions in ASP.NET Core.

URL generation support allows the app to be developed without hard-coding URLs to link the app together. This support allows for starting with a basic routing configuration and modifying the routes after the app's resource layout is determined.

Routing uses routes implementations of IRouter to:

- Map incoming requests to *route handlers*.
- Generate the URLs used in responses.

By default, an app has a single collection of routes. When a request arrives, the routes in the collection are processed in the order that they exist in the collection. The framework attempts to match an incoming request URL to a route in the collection by calling the RouteAsync method on each route in the collection. A response can use routing to generate URLs (for example, for redirection or links) based on route information and thus avoid hard-coded URLs, which helps maintainability.

The routing system has the following characteristics:

- Route template syntax is used to define routes with tokenized route parameters.
- Conventional-style and attribute-style endpoint configuration is permitted.
- IRouteConstraint is used to determine whether a URL parameter contains a valid value for a given endpoint constraint.
- App models, such as MVC/Razor Pages, register all of their routes, which have a predictable implementation of routing scenarios.
- A response can use routing to generate URLs (for example, for redirection or links) based on route information and thus avoid hard-coded URLs, which helps maintainability.
- URL generation is based on routes, which support arbitrary extensibility. IUrlHelper offers methods to build URLs.

Routing is connected to the middleware pipeline by the RouterMiddleware class. ASP.NET Core MVC adds routing to the middleware pipeline as part of its configuration and handles routing in MVC and Razor Pages apps. To learn how to use routing as a standalone component, see the Use Routing Middleware section.

URL matching

URL matching is the process by which routing dispatches an incoming request to a *handler*. This process is based on data in the URL path but can be extended to consider any data in the request. The ability to dispatch requests to separate handlers is key to scaling the size and complexity of an app.

Incoming requests enter the RouterMiddleware, which calls the RouteAsync method on each route in

sequence. The IRouter instance chooses whether to *handle* the request by setting the RouteContext.Handler to a non-null RequestDelegate. If a route sets a handler for the request, route processing stops, and the handler is invoked to process the request. If no route handler is found to process the request, the middleware hands the request off to the next middleware in the request pipeline.

The primary input to RouteAsync is the RouteContext.HttpContext associated with the current request. The RouteContext.Handler and RouteContext.RouteData are outputs set after a route is matched.

A match that calls RouteAsync also sets the properties of the RouteContext.RouteData to appropriate values based on the request processing performed thus far.

RouteData.Values is a dictionary of *route values* produced from the route. These values are usually determined by tokenizing the URL and can be used to accept user input or to make further dispatching decisions inside the app.

RouteData.DataTokens is a property bag of additional data related to the matched route. DataTokens are provided to support associating state data with each route so that the app can make decisions based on which route matched. These values are developer-defined and do **not** affect the behavior of routing in any way. Additionally, values stashed in RouteData.DataTokens can be of any type, in contrast to RouteData.Values, which must be convertible to and from strings.

RouteData.Routers is a list of the routes that took part in successfully matching the request. Routes can be nested inside of one another. The Routers property reflects the path through the logical tree of routes that resulted in a match. Generally, the first item in Routers is the route collection and should be used for URL generation. The last item in Routers is the route handler that matched.

URL generation

URL generation is the process by which routing can create a URL path based on a set of route values. This allows for a logical separation between route handlers and the URLs that access them.

URL generation follows a similar iterative process, but it starts with user or framework code calling into the GetVirtualPath method of the route collection. Each *route* has its GetVirtualPath method called in sequence until a non-null VirtualPathData is returned.

The primary inputs to GetVirtualPath are:

- VirtualPathContext.HttpContext
- VirtualPathContext.Values
- VirtualPathContext.AmbientValues

Routes primarily use the route values provided by Values and AmbientValues to decide whether it's possible to generate a URL and what values to include. The AmbientValues are the set of route values that were produced from matching the current request. In contrast, Values are the route values that specify how to generate the desired URL for the current operation. The HttpContext is provided in case a route should obtain services or additional data associated with the current context.

> **TIP**
>
> Think of VirtualPathContext.Values as a set of overrides for the VirtualPathContext.AmbientValues. URL generation attempts to reuse route values from the current request to generate URLs for links using the same route or route values.

The output of GetVirtualPath is a VirtualPathData. VirtualPathData is a parallel of RouteData. VirtualPathData contains the VirtualPath for the output URL and some additional properties that should be set by the route.

The VirtualPathData.VirtualPath property contains the *virtual path* produced by the route. Depending on

your needs, you may need to process the path further. If you want to render the generated URL in HTML, prepend the base path of the app.

The VirtualPathData.Router is a reference to the route that successfully generated the URL.

The VirtualPathData.DataTokens properties is a dictionary of additional data related to the route that generated the URL. This is the parallel of RouteData.DataTokens.

Create routes

Routing provides the Route class as the standard implementation of IRouter. Route uses the *route template* syntax to define patterns to match against the URL path when RouteAsync is called. Route uses the same route template to generate a URL when GetVirtualPath is called.

Most apps create routes by calling MapRoute or one of the similar extension methods defined on IRouteBuilder. Any of the IRouteBuilder extension methods create an instance of Route and add it to the route collection.

MapRoute doesn't accept a route handler parameter. MapRoute only adds routes that are handled by the DefaultHandler. The default handler is an `IRouter`, and the handler might not handle the request. For example, ASP.NET Core MVC is typically configured as a default handler that only handles requests that match an available controller and action. To learn more about routing in MVC, see Routing to controller actions in ASP.NET Core.

The following code example is an example of a MapRoute call used by a typical ASP.NET Core MVC route definition:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

This template matches a URL path and extracts the route values. For example, the path `/Products/Details/17` generates the following route values: `{ controller = Products, action = Details, id = 17 }`.

Route values are determined by splitting the URL path into segments and matching each segment with the *route parameter* name in the route template. Route parameters are named. The parameters defined by enclosing the parameter name in braces `{ ... }`.

The preceding template could also match the URL path `/` and produce the values `{ controller = Home, action = Index }`. This occurs because the `{controller}` and `{action}` route parameters have default values and the `id` route parameter is optional. An equals sign (`=`) followed by a value after the route parameter name defines a default value for the parameter. A question mark (`?`) after the route parameter name defines an optional parameter.

Route parameters with a default value *always* produce a route value when the route matches. Optional parameters don't produce a route value if there is no corresponding URL path segment. See the Route template reference section for a thorough description of route template scenarios and syntax.

In the following example, the route parameter definition `{id:int}` defines a route constraint for the `id` route parameter:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id:int}");
```

This template matches a URL path like `/Products/Details/17` but not `/Products/Details/Apples`. Route constraints implement IRouteConstraint and inspect route values to verify them. In this example, the route

value `id` must be convertible to an integer. See route-constraint-reference for an explanation of route constraints provided by the framework.

Additional overloads of MapRoute accept values for `constraints`, `dataTokens`, and `defaults`. The typical usage of these parameters is to pass an anonymously typed object, where the property names of the anonymous type match route parameter names.

The following MapRoute examples create equivalent routes:

```
routes.MapRoute(
    name: "default_route",
    template: "{controller}/{action}/{id?}",
    defaults: new { controller = "Home", action = "Index" });

routes.MapRoute(
    name: "default_route",
    template: "{controller=Home}/{action=Index}/{id?}");
```

TIP

The inline syntax for defining constraints and defaults can be convenient for simple routes. However, there are scenarios, such as data tokens, that aren't supported by inline syntax.

The following example demonstrates a few additional scenarios:

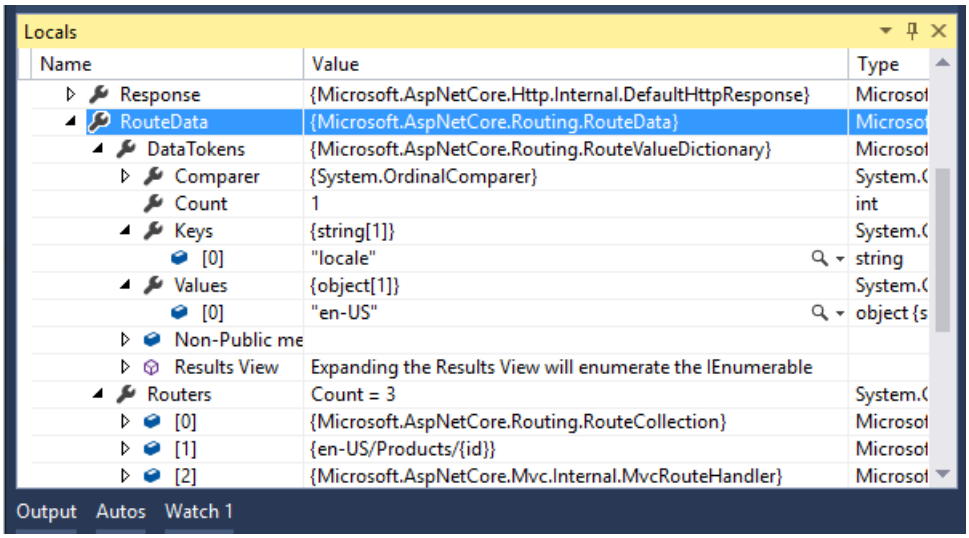
```
routes.MapRoute(
    name: "blog",
    template: "Blog/{*article}",
    defaults: new { controller = "Blog", action = "ReadArticle" });
```

The preceding template matches a URL path like `/Blog/All-About-Routing/Introduction` and extracts the values `{ controller = Blog, action = ReadArticle, article = All-About-Routing/Introduction }`. The default route values for `controller` and `action` are produced by the route even though there are no corresponding route parameters in the template. Default values can be specified in the route template. The `article` route parameter is defined as a *catch-all* by the appearance of an asterisk (`*`) before the route parameter name. Catch-all route parameters capture the remainder of the URL path and can also match the empty string.

The following example adds route constraints and data tokens:

```
routes.MapRoute(
    name: "us_english_products",
    template: "en-US/Products/{id}",
    defaults: new { controller = "Products", action = "Details" },
    constraints: new { id = new IntRouteConstraint() },
    dataTokens: new { locale = "en-US" });
```

The preceding template matches a URL path like `/en-US/Products/5` and extracts the values `{ controller = Products, action = Details, id = 5 }` and the data tokens `{ locale = en-US }`.



Route class URL generation

The Route class can also perform URL generation by combining a set of route values with its route template. This is logically the reverse process of matching the URL path.

> **TIP**
>
> To better understand URL generation, imagine what URL you want to generate and then think about how a route template would match that URL. What values would be produced? This is the rough equivalent of how URL generation works in the Route class.

The following example uses a general ASP.NET Core MVC default route:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{id?}");
```

With the route values `{ controller = Products, action = List }`, the URL `/Products/List` is generated. The route values are substituted for the corresponding route parameters to form the URL path. Since `id` is an optional route parameter, the URL is successfully generated without a value for `id`.

With the route values `{ controller = Home, action = Index }`, the URL `/` is generated. The provided route values match the default values, and the segments corresponding to the default values are safely omitted.

Both URLs generated round-trip with the following route definition (`/Home/Index` and `/`) produce the same route values that were used to generate the URL.

> **NOTE**
>
> An app using ASP.NET Core MVC should use UrlHelper to generate URLs instead of calling into routing directly.

For more information on URL generation, see the Url generation reference section.

Use routing middleware

Reference the Microsoft.AspNetCore.App metapackage in the app's project file.

Add routing to the service container in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRouting();
}
```

Routes must be configured in the `Startup.Configure` method. The sample app uses the following APIs:

- RouteBuilder
- MapGet: Matches only HTTP GET requests.
- UseRouter

```
var trackPackageRouteHandler = new RouteHandler(context =>
{
    var routeValues = context.GetRouteData().Values;
    return context.Response.WriteAsync(
        $"Hello! Route values: {string.Join(", ", routeValues)}");
});

var routeBuilder = new RouteBuilder(app, trackPackageRouteHandler);

routeBuilder.MapRoute(
    "Track Package Route",
    "package/{operation:regex(^track|create$)}/{id:int}");

routeBuilder.MapGet("hello/{name}", context =>
{
    var name = context.GetRouteValue("name");
    // The route handler when HTTP GET "hello/<anything>" matches
    // To match HTTP GET "hello/<anything>/<anything>,
    // use routeBuilder.MapGet("hello/{*name}"
    return context.Response.WriteAsync($"Hi, {name}!");
});

var routes = routeBuilder.Build();
app.UseRouter(routes);
```

The following table shows the responses with the given URIs.

URI	RESPONSE
`/package/create/3`	Hello! Route values: [operation, create], [id, 3]
`/package/track/-3`	Hello! Route values: [operation, track], [id, -3]
`/package/track/-3/`	Hello! Route values: [operation, track], [id, -3]
`/package/track/`	The request falls through, no match.
`GET /hello/Joe`	Hi, Joe!
`POST /hello/Joe`	The request falls through, matches HTTP GET only.
`GET /hello/Joe/Smith`	The request falls through, no match.

If you're configuring a single route, call UseRouter passing in an `IRouter` instance. You won't need to use RouteBuilder.

The framework provides a set of extension methods for creating routes (RequestDelegateRouteBuilderExtensions):

- MapDelete
- MapGet
- MapMiddlewareDelete
- MapMiddlewareGet
- MapMiddlewarePost
- MapMiddlewarePut
- MapMiddlewareRoute
- MapMiddlewareVerb
- MapPost
- MapPut
- MapRoute
- MapVerb

Some of listed methods, such as MapGet, require a RequestDelegate. The RequestDelegate is used as the *route handler* when the route matches. Other methods in this family allow configuring a middleware pipeline for use as the route handler. If the `Map*` method doesn't accept a handler, such as MapRoute, it uses the DefaultHandler.

The `Map[Verb]` methods use constraints to limit the route to the HTTP Verb in the method name. For example, see MapGet and MapVerb.

Route template reference

Tokens within curly braces (`{ ... }`) define *route parameters* that are bound if the route is matched. You can define more than one route parameter in a route segment, but they must be separated by a literal value. For example, `{controller=Home}{action=Index}` isn't a valid route, since there's no literal value between `{controller}` and `{action}`. These route parameters must have a name and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URLs path. To match a literal route parameter delimiter (`{` or `}`), escape the delimiter by repeating the character (`{{` or `}}`).

URL patterns that attempt to capture a file name with an optional file extension have additional considerations. For example, consider the template `files/{filename}.{ext?}`. When values for both `filename` and `ext` exist, both values are populated. If only a value for `filename` exists in the URL, the route matches because the trailing period (`.`) is optional. The following URLs match this route:

- `/files/myFile.txt`
- `/files/myFile`

You can use the asterisk (`*`) as a prefix to a route parameter to bind to the rest of the URI. This is called a *catch-all* parameter. For example, `blog/{*slug}` matches any URI that starts with `/blog` and has any value following it, which is assigned to the `slug` route value. Catch-all parameters can also match the empty string.

The catch-all parameter escapes the appropriate characters when the route is used to generate a URL, including path separator (`/`) characters. For example, the route `foo/{*path}` with route values `{ path = "my/path" }` generates `foo/my%2Fpath`. Note the escaped forward slash.

Route parameters may have *default values* designated by specifying the default value after the parameter name separated by an equals sign (`=`). For example, `{controller=Home}` defines `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. Route parameters are made optional by appending a question mark (`?`) to the end of the parameter name, as in `id?`. The difference between optional values and default route parameters is that a route parameter with a default value always produces a value—an optional parameter has a value only when a value is provided by the request URL.

Route parameters may have constraints that must match the route value bound from the URL. Adding a colon (`:`) and constraint name after the route parameter name specifies an *inline constraint* on a route parameter. If the constraint requires arguments, they're enclosed in parentheses (`(...)`) after the constraint name. Multiple inline constraints can be specified by appending another colon (`:`) and constraint name.

The constraint name and arguments are passed to the IInlineConstraintResolver service to create an instance of IRouteConstraint to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies a `minlength` constraint with the argument `10`. For more information on route constraints and a list of the constraints provided by the framework, see the Route constraint reference section.

The following table demonstrates example route templates and their behavior.

ROUTE TEMPLATE	EXAMPLE MATCHING URI	THE REQUEST URI...
`hello`	`/hello`	Only matches the single path `/hello`.
`{Page=Home}`	`/`	Matches and sets `Page` to `Home`.
`{Page=Home}`	`/Contact`	Matches and sets `Page` to `Contact`.
`{controller}/{action}/{id?}`	`/Products/List`	Maps to the `Products` controller and `List` action.
`{controller}/{action}/{id?}`	`/Products/Details/123`	Maps to the `Products` controller and `Details` action (`id` set to 123).
`{controller=Home}/{action=Index}/{id?}`	`/`	Maps to the `Home` controller and `Index` method (`id` is ignored).

Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

> **TIP**
> Enable Logging to see how the built-in routing implementations, such as Route, match requests.

Route constraint reference

Route constraints execute when a match has occurred to the incoming URL and the URL path is tokenized into route values. Route constraints generally inspect the route value associated via the route template and make a yes/no decision about whether or not the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the

HttpMethodRouteConstraint can accept or reject a request based on its HTTP verb. Constraints are used in routing requests and link generation.

> **WARNING**
> Don't use constraints for **input validation**. If constraints are used for **input validation**, invalid input results in a *404 - Not Found* response instead of a *400 - Bad Request* with an appropriate error message. Route constraints are used to **disambiguate** similar routes, not to validate the inputs for a particular route.

The following table demonstrates example route constraints and their expected behavior.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`int`	`{id:int}`	`123456789`, `-123456789`	Matches any integer
`bool`	`{active:bool}`	`true`, `FALSE`	Matches `true` or `false` (case-insensitive)
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Matches a valid `DateTime` value in the invariant culture. See preceding warning.
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Matches a valid `decimal` value in the invariant culture. See preceding warning.
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Matches a valid `double` value in the invariant culture. See preceding warning.
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Matches a valid `float` value in the invariant culture. See preceding warning.
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	Matches a valid `Guid` value
`long`	`{ticks:long}`	`123456789`, `-123456789`	Matches a valid `long` value
`minlength(value)`	`{username:minlength(4)}`	`Rick`	String must be at least 4 characters
`maxlength(value)`	`{filename:maxlength(8)}`	`Richard`	String must be no more than 8 characters
`length(length)`	`{filename:length(12)}`	`somefile.txt`	String must be exactly 12 characters long

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	NOTES
`length(min,max)`	`{filename:length(8,16)}`	`somefile.txt`	String must be at least 8 and no more than 16 characters long
`min(value)`	`{age:min(18)}`	`19`	Integer value must be at least 18
`max(value)`	`{age:max(120)}`	`91`	Integer value must be no more than 120
`range(min,max)`	`{age:range(18,120)}`	`91`	Integer value must be at least 18 but no more than 120
`alpha`	`{name:alpha}`	`Rick`	String must consist of one or more alphabetical characters (`a`-`z`, case-insensitive)
`regex(expression)`	`{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}`	`123-45-6789`	String must match the regular expression (see tips about defining a regular expression)
`required`	`{name:required}`	`Rick`	Used to enforce that a non-parameter value is present during URL generation

Multiple, colon-delimited constraints can be applied to a single parameter. For example, the following constraint restricts a parameter to an integer value of 1 or greater:

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { }
```

WARNING

Route constraints that verify the URL and are converted to a CLR type (such as `int` or `DateTime`) always use the invariant culture. These constraints assume that the URL is non-localizable. The framework-provided route constraints don't modify the values stored in route values. All route values parsed from the URL are stored as strings. For example, the `float` constraint attempts to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

Regular expressions

The ASP.NET Core framework adds `RegexOptions.IgnoreCase | RegexOptions.Compiled | RegexOptions.CultureInvariant` to the regular expression constructor. See RegexOptions for a description of these members.

Regular expressions use delimiters and tokens similar to those used by Routing and the C# language. Regular expression tokens must be escaped. To use the regular expression `^\d{3}-\d{2}-\d{4}$` in routing, the expression must have the `\` (single backslash) characters provided in the string as `\\` (double backslash)

characters in the C# source file in order to escape the `\` string escape character (unless using verbatim string literals). To escape routing parameter delimiter characters (`{`, `}`, `[`, `]`), double the characters in the expression (`{{`, `}`, `[[`, `]]`). The following table shows a regular expression and the escaped version.

REGULAR EXPRESSION	ESCAPED REGULAR EXPRESSION
`^\d{3}-\d{2}-\d{4}$`	`^\\d{{3}}-\\d{{2}}-\\d{{4}}$`
`^[a-z]{2}$`	`^[[a-z]]{{2}}$`

Regular expressions used in routing often start with the caret (`^`) character and match starting position of the string. The expressions often end with the dollar sign (`$`) character and match end of the string. The `^` and `$` characters ensure that the regular expression match the entire route parameter value. Without the `^` and `$` characters, the regular expression match any substring within the string, which is often undesirable. The following table provides examples and explains why they match or fail to match.

EXPRESSION	STRING	MATCH	COMMENT
`[a-z]{2}`	hello	Yes	Substring matches
`[a-z]{2}`	123abc456	Yes	Substring matches
`[a-z]{2}`	mz	Yes	Matches expression
`[a-z]{2}`	MZ	Yes	Not case sensitive
`^[a-z]{2}$`	hello	No	See `^` and `$` above
`^[a-z]{2}$`	123abc456	No	See `^` and `$` above

For more information on regular expression syntax, see .NET Framework Regular Expressions.

To constrain a parameter to a known set of possible values, use a regular expression. For example, `{action:regex(^(list|get|create)$)}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string `^(list|get|create)$` is equivalent. Constraints that are passed in the constraints dictionary (not inline within a template) that don't match one of the known constraints are also treated as regular expressions.

Custom Route Constraints

In addition to the built-in route constraints, custom route constraints can be created by implementing the IRouteConstraint interface. The IRouteConstraint interface contains a single method, `Match`, which returns `true` if the constraint is satisfied and `false` otherwise.

To use a custom IRouteConstraint, the route constraint type must be registered with the app's ConstraintMap in the app's service container. A ConstraintMap is a dictionary that maps route constraint keys to IRouteConstraint implementations that validate those constraints. An app's ConstraintMap can be updated in `Startup.ConfigureServices` either as part of a services.AddRouting call or by configuring RouteOptions directly with `services.Configure<RouteOptions>`. For example:

```
services.AddRouting(options =>
{
    options.ConstraintMap.Add("customName", typeof(MyCustomConstraint));
});
```

The constraint can then be applied to routes in the usual manner, using the name specified when registering the constraint type. For example:

```
[HttpGet("{id:customName}")]
public ActionResult<string> Get(string id)
```

URL generation reference

The following example shows how to generate a link to a route given a dictionary of route values and a RouteCollection.

```
app.Run(async (context) =>
{
    var dictionary = new RouteValueDictionary
    {
        { "operation", "create" },
        { "id", 123}
    };

    var vpc = new VirtualPathContext(context, null, dictionary,
        "Track Package Route");
    var path = routes.GetVirtualPath(vpc).VirtualPath;

    context.Response.ContentType = "text/html";
    await context.Response.WriteAsync("Menu<hr/>");
    await context.Response.WriteAsync(
        $"<a href='{path}'>Create Package 123</a><br/>");
});
```

The VirtualPath generated at the end of the preceding sample is `/package/create/123`. The dictionary supplies the `operation` and `id` route values of the "Track Package Route" template, `package/{operation}/{id}`. For details, see the sample code in the Use Routing Middleware section or the sample app.

The second parameter to the VirtualPathContext constructor is a collection of *ambient values*. Ambient values are convenient to use because they limit the number of values a developer must specify within a request context. The current route values of the current request are considered ambient values for link generation. In an ASP.NET Core MVC app's `About` action of the `HomeController`, you don't need to specify the controller route value to link to the `Index` action—the ambient value of `Home` is used.

Ambient values that don't match a parameter are ignored. Ambient values are also ignored when an explicitly provided value overrides the ambient value. Matching occurs from left to right in the URL.

Values explicitly provided but that don't match a segment of the route are added to the query string. The following table shows the result when using the route template `{controller}/{action}/{id?}`.

AMBIENT VALUES	EXPLICIT VALUES	RESULT
controller = "Home"	action = "About"	`/Home/About`

AMBIENT VALUES	EXPLICIT VALUES	RESULT
controller = "Home"	controller = "Order", action = "About"	`/Order/About`
controller = "Home", color = "Red"	action = "About"	`/Home/About`
controller = "Home"	action = "About", color = "Red"	`/Home/About?color=Red`

If a route has a default value that doesn't correspond to a parameter and that value is explicitly provided, it must match the default value:

```
routes.MapRoute("blog_route", "blog/{*slug}",
    defaults: new { controller = "Blog", action = "ReadPost" });
```

Link generation only generates a link for this route when the matching values for `controller` and `action` are provided.

Complex segments

Complex segments (for example `[Route("/x{token}y")]`) are processed by matching up literals from right to left in a non-greedy way. See this code for a detailed explanation of how complex segments are matched. The code sample is not used by ASP.NET Core, but it provides a good explanation of complex segments.

Handle errors in ASP.NET Core

9/22/2020 • 19 minutes to read • Edit Online

By Kirk Larkin, Tom Dykstra, and Steve Smith

This article covers common approaches to handling errors in ASP.NET Core web apps. See Handle errors in ASP.NET Core web APIs for web APIs.

View or download sample code. (How to download.) The network tab on the F12 browser developer tools is useful when testing the sample app.

Developer Exception Page

The *Developer Exception Page* displays detailed information about request exceptions. The ASP.NET Core templates generate the following code:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

The preceding highlighted code enables the developer exception page when the app is running in the Development environment.

The templates place UseDeveloperExceptionPage early in the middleware pipeline so that it can catch exceptions thrown in middleware that follows.

The preceding code enables the Developer Exception Page ***only*** when the app runs in the Development environment. Detailed exception information should not be displayed publicly when the app runs in the Production environment. For more information on configuring environments, see Use multiple environments in ASP.NET Core.

The Developer Exception Page includes the following information about the exception and the request:

- Stack trace
- Query string parameters if any

- Cookies if any
- Headers

Exception handler page

To configure a custom error handling page for the Production environment, call UseExceptionHandler. This exception handling middleware:

- Catches and logs exceptions.
- Re-executes the request in an alternate pipeline using the path indicated. The request isn't re-executed if the response has started. The template generated code re-executes the request using the `/Error` path.

In the following example, UseExceptionHandler adds the exception handling middleware in non-Development environments:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

The Razor Pages app template provides an Error page (*.cshtml*) and PageModel class (`ErrorModel`) in the *Pages* folder. For an MVC app, the project template includes an `Error` action method and an Error view for the Home controller.

Don't mark the error handler action method with HTTP method attributes, such as `HttpGet`. Explicit verbs prevent some requests from reaching the action method. Allow anonymous access to the method if unauthenticated users should see the error view.

Access the exception

Use IExceptionHandlerPathFeature to access the exception and the original request path in an error handler. The following code adds `ExceptionMessage` to the default *Pages/Error.cshtml.cs* generated by the ASP.NET Core templates:

```
[ResponseCache(Duration=0, Location=ResponseCacheLocation.None, NoStore=true)]
[IgnoreAntiforgeryToken]
public class ErrorModel : PageModel
{
    public string RequestId { get; set; }
    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);
    public string ExceptionMessage { get; set; }
    private readonly ILogger<ErrorModel> _logger;

    public ErrorModel(ILogger<ErrorModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;

        var exceptionHandlerPathFeature =
        HttpContext.Features.Get<IExceptionHandlerPathFeature>();
        if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
        {
            ExceptionMessage = "File error thrown";
            _logger.LogError(ExceptionMessage);
        }
        if (exceptionHandlerPathFeature?.Path == "/index")
        {
            ExceptionMessage += " from home page";
        }
    }
}
```

WARNING

Do **not** serve sensitive error information to clients. Serving errors is a security risk.

To test the exception in the sample app:

- Set the environment to production.
- Remove the comments from `webBuilder.UseStartup<Startup>();` in *Program.cs*.
- Select **Trigger an exception** on the home page.

Exception handler lambda

An alternative to a custom exception handler page is to provide a lambda to UseExceptionHandler. Using a lambda allows access to the error before returning the response.

The following code uses a lambda for exception handling:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler(errorApp =>
        {
            errorApp.Run(async context =>
            {
                context.Response.StatusCode = 500;
                context.Response.ContentType = "text/html";

                await context.Response.WriteAsync("<html lang=\"en\"><body>\r\n");
                await context.Response.WriteAsync("ERROR!<br><br>\r\n");

                var exceptionHandlerPathFeature =
                    context.Features.Get<IExceptionHandlerPathFeature>();

                if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
                {
                    await context.Response.WriteAsync(
                                            "File error thrown!<br><br>\r\n");
                }

                await context.Response.WriteAsync(
                                            "<a href=\"/\">Home</a><br>\r\n");
                await context.Response.WriteAsync("</body></html>\r\n");
                await context.Response.WriteAsync(new string(' ', 512));
            });
        });
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

WARNING

Do **not** serve sensitive error information from IExceptionHandlerFeature or IExceptionHandlerPathFeature to clients. Serving errors is a security risk.

To test the exception handling lambda in the sample app:

- Set the environment to production.
- Remove the comments from `webBuilder.UseStartup<StartupLambda>();` in *Program.cs*.
- Select **Trigger an exception** on the home page.

UseStatusCodePages

By default, an ASP.NET Core app doesn't provide a status code page for HTTP error status codes, such as *404 - Not Found*. When the app encounters an HTTP 400-499 error condition that doesn't have a body, it returns the status code and an empty response body. To provide status code pages, use the status code pages middleware. To enable default text-only handlers for common error status codes, call UseStatusCodePages in the `Startup.Configure` method:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseStatusCodePages();

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Call `UseStatusCodePages` before request handling middleware. For example, call `UseStatusCodePages` before the Static File Middleware and the Endpoints Middleware.

When `UseStatusCodePages` isn't used, navigating to a URL without an endpoint returns a browser dependent error message indicating the endpoint can't be found. For example, navigating to `Home/Privacy2`. When `UseStatusCodePages` is called, the browser returns:

```
Status Code: 404; Not Found
```

`UseStatusCodePages` isn't typically used in production because it returns a message that isn't useful to users.

To test `UseStatusCodePages` in the sample app:

- Set the environment to production.
- Remove the comments from `webBuilder.UseStartup<StartupUseStatusCodePages>();` in *Program.cs*.
- Select the links on the home page on the home page.

UseStatusCodePages with format string

To customize the response content type and text, use the overload of UseStatusCodePages that takes a content type and format string:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseStatusCodePages(
        "text/plain", "Status code page, status code: {0}");

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

In the preceding code, `{0}` is a placeholder for the error code.

`UseStatusCodePages` with a format string isn't typically used in production because it returns a message that isn't useful to users.

To test `UseStatusCodePages` in the sample app, remove the comments from `webBuilder.UseStartup<StartupFormat>();` in *Program.cs*.

UseStatusCodePages with lambda

To specify custom error-handling and response-writing code, use the overload of UseStatusCodePages that takes a lambda expression:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseStatusCodePages(async context =>
    {
        context.HttpContext.Response.ContentType = "text/plain";

        await context.HttpContext.Response.WriteAsync(
            "Status code page, status code: " +
            context.HttpContext.Response.StatusCode);
    });

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

`UseStatusCodePages` with a lambda isn't typically used in production because it returns a message that isn't useful to users.

To test `UseStatusCodePages` in the sample app, remove the comments from `webBuilder.UseStartup<StartupStatusLambda>();` in *Program.cs*.

UseStatusCodePagesWithRedirects

The UseStatusCodePagesWithRedirects extension method:

- Sends a 302 - Found status code to the client.
- Redirects the client to the error handling endpoint provided in the URL template. The error handling endpoint typically displays error information and returns HTTP 200.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseStatusCodePagesWithRedirects("/MyStatusCode?code={0}");

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

The URL template can include a `{0}` placeholder for the status code, as shown in the preceding code. If the URL template starts with `~` (tilde), the `~` is replaced by the app's `PathBase`. When specifying an endpoint in the app, create an MVC view or Razor page for the endpoint. For a Razor Pages example, see Pages/MyStatusCode.cshtml in the sample app.

This method is commonly used when the app:

- Should redirect the client to a different endpoint, usually in cases where a different app processes the error. For web apps, the client's browser address bar reflects the redirected endpoint.
- Shouldn't preserve and return the original status code with the initial redirect response.

To test `UseStatusCodePages` in the sample app, remove the comments from `webBuilder.UseStartup<StartupSCredirect>();` in *Program.cs*.

UseStatusCodePagesWithReExecute

The UseStatusCodePagesWithReExecute extension method:

- Returns the original status code to the client.
- Generates the response body by re-executing the request pipeline using an alternate path.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseStatusCodePagesWithReExecute("/MyStatusCode2", "?code={0}");

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

If an endpoint within the app is specified, create an MVC view or Razor page for the endpoint. Ensure `UseStatusCodePagesWithReExecute` is placed before `UseRouting` so the request can be rerouted to the status page. For a Razor Pages example, see Pages/MyStatusCode2.cshtml in the sample app.

This method is commonly used when the app should:

- Process the request without redirecting to a different endpoint. For web apps, the client's browser address bar reflects the originally requested endpoint.
- Preserve and return the original status code with the response.

The URL and query string templates may include a placeholder `{0}` for the status code. The URL template must start with `/`.

The endpoint that processes the error can get the original URL that generated the error, as shown in the following example:

```
[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
public class MyStatusCode2Model : PageModel
{
    public string RequestId { get; set; }
    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public string ErrorStatusCode { get; set; }

    public string OriginalURL { get; set; }
    public bool ShowOriginalURL => !string.IsNullOrEmpty(OriginalURL);

    public void OnGet(string code)
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
        ErrorStatusCode = code;

        var statusCodeReExecuteFeature = HttpContext.Features.Get<
                                            IStatusCodeReExecuteFeature>();
        if (statusCodeReExecuteFeature != null)
        {
            OriginalURL =
                statusCodeReExecuteFeature.OriginalPathBase
                + statusCodeReExecuteFeature.OriginalPath
                + statusCodeReExecuteFeature.OriginalQueryString;
        }
    }
}
```

For a Razor Pages example, see Pages/MyStatusCode2.cshtml in the sample app.

To test `UseStatusCodePages` in the sample app, remove the comments from `webBuilder.UseStartup<StartupSCreX>();` in *Program.cs*.

Disable status code pages

To disable status code pages for an MVC controller or action method, use the [SkipStatusCodePages] attribute.

To disable status code pages for specific requests in a Razor Pages handler method or in an MVC controller, use IStatusCodePagesFeature:

```
public void OnGet()
{
    // using Microsoft.AspNetCore.Diagnostics;
    var statusCodePagesFeature = HttpContext.Features.Get<IStatusCodePagesFeature>();

    if (statusCodePagesFeature != null)
    {
        statusCodePagesFeature.Enabled = false;
    }
}
```

Exception-handling code

Code in exception handling pages can also throw exceptions. Production error pages should be tested thoroughly and take extra care to avoid throwing exceptions of their own.

Response headers

Once the headers for a response are sent:

- The app can't change the response's status code.

- Any exception pages or handlers can't run. The response must be completed or the connection aborted.

Server exception handling

In addition to the exception handling logic in an app, the HTTP server implementation can handle some exceptions. If the server catches an exception before response headers are sent, the server sends a `500 - Internal Server Error` response without a response body. If the server catches an exception after response headers are sent, the server closes the connection. Requests that aren't handled by the app are handled by the server. Any exception that occurs when the server is handling the request is handled by the server's exception handling. The app's custom error pages, exception handling middleware, and filters don't affect this behavior.

Startup exception handling

Only the hosting layer can handle exceptions that take place during app startup. The host can be configured to capture startup errors and capture detailed errors.

The hosting layer can show an error page for a captured startup error only if the error occurs after host address/port binding. If binding fails:

- The hosting layer logs a critical exception.
- The dotnet process crashes.
- No error page is displayed when the HTTP server is Kestrel.

When running on IIS (or Azure App Service) or IIS Express, a *502.5 - Process Failure* is returned by the ASP.NET Core Module if the process can't start. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Database error page

The Database developer page exception filter `AddDatabaseDeveloperPageExceptionFilter` captures database-related exceptions that can be resolved by using Entity Framework Core migrations. When these exceptions occur, an HTML response is generated with details of possible actions to resolve the issue. This page is enabled only in the Development environment. The following code was generated by the ASP.NET Core Razor Pages templates when individual user accounts were specified:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDatabaseDeveloperPageExceptionFilter();
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
}
```

Exception filters

In MVC apps, exception filters can be configured globally or on a per-controller or per-action basis. In Razor Pages apps, they can be configured globally or per page model. These filters handle any unhandled exceptions that occur during the execution of a controller action or another filter. For more information, see Filters in ASP.NET Core.

Exception filters are useful for trapping exceptions that occur within MVC actions, but they're not as flexible as

the built-in exception handling middleware, `UseExceptionHandler`. We recommend using `UseExceptionHandler`, unless you need to perform error handling differently based on which MVC action is chosen.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Model state errors

For information about how to handle model state errors, see Model binding and Model validation.

Additional resources

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common errors reference for Azure App Service and IIS with ASP.NET Core

By Tom Dykstra, and Steve Smith

This article covers common approaches to handling errors in ASP.NET Core web apps. See Handle errors in ASP.NET Core web APIs for web APIs.

View or download sample code. (How to download.)

Developer Exception Page

The *Developer Exception Page* displays detailed information about request exceptions. The ASP.NET Core templates generate the following code:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

The preceding code enables the developer exception page when the app is running in the Development

environment.

The templates place UseDeveloperExceptionPage before any middleware so exceptions are caught in the middleware that follows.

The preceding code enables the Developer Exception Page **only when the app is running in the Development environment**. Detailed exception information should not be displayed publicly when the app runs in production. For more information on configuring environments, see Use multiple environments in ASP.NET Core.

The Developer Exception Page includes the following information about the exception and the request:

- Stack trace
- Query string parameters if any
- Cookies if any
- Headers

Exception handler page

To configure a custom error handling page for the Production environment, use the Exception Handling Middleware. The middleware:

- Catches and logs exceptions.
- Re-executes the request in an alternate pipeline for the page or controller indicated. The request isn't re-executed if the response has started. The template generated code re-executes the request to `/Error`.

In the following example, UseExceptionHandler adds the Exception Handling Middleware in non-Development environments:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

The Razor Pages app template provides an Error page (*.cshtml*) and PageModel class (`ErrorModel`) in the *Pages* folder. For an MVC app, the project template includes an Error action method and an Error view in the Home controller.

Don't mark the error handler action method with HTTP method attributes, such as `HttpGet`. Explicit verbs prevent some requests from reaching the method. Allow anonymous access to the method if unauthenticated users should see the error view.

Access the exception

Use IExceptionHandlerPathFeature to access the exception and the original request path in an error handler controller or page:

```
[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
public class ErrorModel : PageModel
{
    public string RequestId { get; set; }
    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);
    public string ExceptionMessage { get; set; }

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;

        var exceptionHandlerPathFeature =
            HttpContext.Features.Get<IExceptionHandlerPathFeature>();
        if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
        {
            ExceptionMessage = "File error thrown";
        }
        if (exceptionHandlerPathFeature?.Path == "/index")
        {
            ExceptionMessage += " from home page";
        }
    }
}
```

WARNING

Do **not** serve sensitive error information to clients. Serving errors is a security risk.

To trigger the preceding exception handling page, set the environment to productions and force an exception.

Exception handler lambda

An alternative to a custom exception handler page is to provide a lambda to UseExceptionHandler. Using a lambda allows access to the error before returning the response.

Here's an example of using a lambda for exception handling:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
   app.UseExceptionHandler(errorApp =>
   {
        errorApp.Run(async context =>
        {
            context.Response.StatusCode = 500;
            context.Response.ContentType = "text/html";

            await context.Response.WriteAsync("<html lang=\"en\"><body>\r\n");
            await context.Response.WriteAsync("ERROR!<br><br>\r\n");

            var exceptionHandlerPathFeature =
                context.Features.Get<IExceptionHandlerPathFeature>();

            if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
            {
                await context.Response.WriteAsync("File error thrown!<br><br>\r\n");
            }

            await context.Response.WriteAsync("<a href=\"/\">Home</a><br>\r\n");
            await context.Response.WriteAsync("</body></html>\r\n");
            await context.Response.WriteAsync(new string(' ', 512)); // IE padding
        });
    });
    app.UseHsts();
}
```

In the preceding code, `await context.Response.WriteAsync(new string(' ', 512));` is added so the Internet Explorer browser displays the error message rather than an IE error message. For more information, see this GitHub issue.

> **WARNING**
>
> Do **not** serve sensitive error information from IExceptionHandlerFeature or IExceptionHandlerPathFeature to clients. Serving errors is a security risk.

To see the result of the exception handling lambda in the sample app, use the `ProdEnvironment` and `ErrorHandlerLambda` preprocessor directives, and select **Trigger an exception** on the home page.

UseStatusCodePages

By default, an ASP.NET Core app doesn't provide a status code page for HTTP status codes, such as *404 - Not Found*. The app returns a status code and an empty response body. To provide status code pages, use Status Code Pages middleware.

The middleware is made available by the Microsoft.AspNetCore.Diagnostics package.

To enable default text-only handlers for common error status codes, call UseStatusCodePages in the `Startup.Configure` method:

```
app.UseStatusCodePages();
```

Call `UseStatusCodePages` before request handling middleware (for example, Static File Middleware and MVC Middleware).

When `UseStatusCodePages` isn't used, navigating to a URL without an endpoint returns a browser dependent error message indicating the endpoint can't be found. For example, navigating to `Home/Privacy2`. When `UseStatusCodePages` is called, the browser returns:

```
Status Code: 404; Not Found
```

UseStatusCodePages with format string

To customize the response content type and text, use the overload of UseStatusCodePages that takes a content type and format string:

```
app.UseStatusCodePages(
    "text/plain", "Status code page, status code: {0}");
```

UseStatusCodePages with lambda

To specify custom error-handling and response-writing code, use the overload of UseStatusCodePages that takes a lambda expression:

```
app.UseStatusCodePages(async context =>
{
    context.HttpContext.Response.ContentType = "text/plain";

    await context.HttpContext.Response.WriteAsync(
        "Status code page, status code: " +
        context.HttpContext.Response.StatusCode);
});
```

UseStatusCodePagesWithRedirects

The UseStatusCodePagesWithRedirects extension method:

- Sends a *302 - Found* status code to the client.
- Redirects the client to the location provided in the URL template.

```
app.UseStatusCodePagesWithRedirects("/StatusCode?code={0}");
```

The URL template can include a `{0}` placeholder for the status code, as shown in the example. If the URL template starts with `~` (tilde), the `~` is replaced by the app's `PathBase`. If you point to an endpoint within the app, create an MVC view or Razor page for the endpoint. For a Razor Pages example, see *Pages/StatusCode.cshtml* in the sample app.

This method is commonly used when the app:

- Should redirect the client to a different endpoint, usually in cases where a different app processes the error. For web apps, the client's browser address bar reflects the redirected endpoint.
- Shouldn't preserve and return the original status code with the initial redirect response.

UseStatusCodePagesWithReExecute

The UseStatusCodePagesWithReExecute extension method:

- Returns the original status code to the client.
- Generates the response body by re-executing the request pipeline using an alternate path.

```
app.UseStatusCodePagesWithReExecute("/StatusCode","?code={0}");
```

If you point to an endpoint within the app, create an MVC view or Razor page for the endpoint. Ensure `UseStatusCodePagesWithReExecute` is placed before `UseRouting` so the request can be rerouted to the status page. For a Razor Pages example, see *Pages/StatusCode.cshtml* in the sample app.

This method is commonly used when the app should:

- Process the request without redirecting to a different endpoint. For web apps, the client's browser address bar reflects the originally requested endpoint.
- Preserve and return the original status code with the response.

The URL and query string templates may include a placeholder (`{0}`) for the status code. The URL template must start with a slash (`/`). When using a placeholder in the path, confirm that the endpoint (page or controller) can process the path segment. For example, a Razor Page for errors should accept the optional path segment value with the `@page` directive:

```
@page "{code?}"
```

The endpoint that processes the error can get the original URL that generated the error, as shown in the following example:

```
var statusCodeReExecuteFeature = HttpContext.Features.Get<IStatusCodeReExecuteFeature>();
if (statusCodeReExecuteFeature != null)
{
    OriginalURL =
        statusCodeReExecuteFeature.OriginalPathBase
        + statusCodeReExecuteFeature.OriginalPath
        + statusCodeReExecuteFeature.OriginalQueryString;
}
```

Disable status code pages

To disable status code pages for an MVC controller or action method, use the `[SkipStatusCodePages]` attribute.

To disable status code pages for specific requests in a Razor Pages handler method or in an MVC controller, use IStatusCodePagesFeature:

```
var statusCodePagesFeature = HttpContext.Features.Get<IStatusCodePagesFeature>();

if (statusCodePagesFeature != null)
{
    statusCodePagesFeature.Enabled = false;
}
```

Exception-handling code

Code in exception handling pages can throw exceptions. It's often a good idea for production error pages to consist of purely static content.

Response headers

Once the headers for a response are sent:

- The app can't change the response's status code.
- Any exception pages or handlers can't run. The response must be completed or the connection aborted.

Server exception handling

In addition to the exception handling logic in your app, the HTTP server implementation can handle some exceptions. If the server catches an exception before response headers are sent, the server sends a *500 - Internal Server Error* response without a response body. If the server catches an exception after response headers are sent, the server closes the connection. Requests that aren't handled by your app are handled by the server. Any exception that occurs when the server is handling the request is handled by the server's exception handling. The app's custom error pages, exception handling middleware, and filters don't affect this behavior.

Startup exception handling

Only the hosting layer can handle exceptions that take place during app startup. The host can be configured to capture startup errors and capture detailed errors.

The hosting layer can show an error page for a captured startup error only if the error occurs after host address/port binding. If binding fails:

- The hosting layer logs a critical exception.
- The dotnet process crashes.
- No error page is displayed when the HTTP server is Kestrel.

When running on IIS (or Azure App Service) or IIS Express, a *502.5 - Process Failure* is returned by the ASP.NET Core Module if the process can't start. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Database error page

Database Error Page Middleware captures database-related exceptions that can be resolved by using Entity Framework migrations. When these exceptions occur, an HTML response with details of possible actions to resolve the issue is generated. This page should be enabled only in the Development environment. Enable the page by adding code to `Startup.Configure`:

```
if (env.IsDevelopment())
{
    app.UseDatabaseErrorPage();
}
```

UseDatabaseErrorPage requires the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package.

Exception filters

In MVC apps, exception filters can be configured globally or on a per-controller or per-action basis. In Razor Pages apps, they can be configured globally or per page model. These filters handle any unhandled exception that occurs during the execution of a controller action or another filter. For more information, see Filters in ASP.NET Core.

> **TIP**
>
> Exception filters are useful for trapping exceptions that occur within MVC actions, but they're not as flexible as the Exception Handling Middleware. We recommend using the middleware. Use filters only where you need to perform error handling differently based on which MVC action is chosen.

Model state errors

For information about how to handle model state errors, see Model binding and Model validation.

Additional resources

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common errors reference for Azure App Service and IIS with ASP.NET Core

Make HTTP requests using IHttpClientFactory in ASP.NET Core

9/22/2020 • 47 minutes to read • Edit Online

By Glenn Condron, Ryan Nowak, Steve Gordon, Rick Anderson, and Kirk Larkin

An IHttpClientFactory can be registered and used to configure and create HttpClient instances in an app. `IHttpClientFactory` offers the following benefits:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, a client named *github* could be registered and configured to access GitHub. A default client can be registered for general access.
- Codifies the concept of outgoing middleware via delegating handlers in `HttpClient`. Provides extensions for Polly-based middleware to take advantage of delegating handlers in `HttpClient`.
- Manages the pooling and lifetime of underlying `HttpClientMessageHandler` instances. Automatic management avoids common DNS (Domain Name System) problems that occur when manually managing `HttpClient` lifetimes.
- Adds a configurable logging experience (via `ILogger`) for all requests sent through clients created by the factory.

View or download sample code (how to download).

The sample code in this topic version uses System.Text.Json to deserialize JSON content returned in HTTP responses. For samples that use `Json.NET` and `ReadAsAsync<T>`, use the version selector to select a 2.x version of this topic.

Consumption patterns

There are several ways `IHttpClientFactory` can be used in an app:

- Basic usage
- Named clients
- Typed clients
- Generated clients

The best approach depends upon the app's requirements.

Basic usage

`IHttpClientFactory` can be registered by calling `AddHttpClient`:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHttpClient();
        // Remaining code deleted for brevity.
```

An `IHttpClientFactory` can be requested using dependency injection (DI). The following code uses `IHttpClientFactory` to create an `HttpClient` instance:

```
public class BasicUsageModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubBranch> Branches { get; private set; }

    public bool GetBranchesError { get; private set; }

    public BasicUsageModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "https://api.github.com/repos/aspnet/AspNetCore.Docs/branches");
        request.Headers.Add("Accept", "application/vnd.github.v3+json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        var client = _clientFactory.CreateClient();

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            using var responseStream = await response.Content.ReadAsStreamAsync();
            Branches = await JsonSerializer.DeserializeAsync
                <IEnumerable<GitHubBranch>>(responseStream);
        }
        else
        {
            GetBranchesError = true;
            Branches = Array.Empty<GitHubBranch>();
        }
    }
}
```

Using `IHttpClientFactory` like in the preceding example is a good way to refactor an existing app. It has no impact on how `HttpClient` is used. In places where `HttpClient` instances are created in an existing app, replace those occurrences with calls to CreateClient.

Named clients

Named clients are a good choice when:

- The app requires many distinct uses of `HttpClient`.

- Many `HttpClient`s have different configuration.

Configuration for a named `HttpClient` can be specified during registration in `Startup.ConfigureServices`:

```
services.AddHttpClient("github", c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    // Github API versioning
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    // Github requires a user-agent
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

In the preceding code the client is configured with:

- The base address `https://api.github.com/`.
- Two headers required to work with the GitHub API.

CreateClient

Each time CreateClient is called:

- A new instance of `HttpClient` is created.
- The configuration action is called.

To create a named client, pass its name into `CreateClient`:

```
public class NamedClientModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubPullRequest> PullRequests { get; private set; }

    public bool GetPullRequestsError { get; private set; }

    public bool HasPullRequests => PullRequests.Any();

    public NamedClientModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "repos/aspnet/AspNetCore.Docs/pulls");

        var client = _clientFactory.CreateClient("github");

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            using var responseStream = await response.Content.ReadAsStreamAsync();
            PullRequests = await JsonSerializer.DeserializeAsync
                    <IEnumerable<GitHubPullRequest>>(responseStream);
        }
        else
        {
            GetPullRequestsError = true;
            PullRequests = Array.Empty<GitHubPullRequest>();
        }
    }
}
```

In the preceding code, the request doesn't need to specify a hostname. The code can pass just the path, since the base address configured for the client is used.

Typed clients

Typed clients:

- Provide the same capabilities as named clients without the need to use strings as keys.
- Provides IntelliSense and compiler help when consuming clients.
- Provide a single location to configure and interact with a particular `HttpClient`. For example, a single typed client might be used:
  - For a single backend endpoint.
  - To encapsulate all logic dealing with the endpoint.
- Work with DI and can be injected where required in the app.

A typed client accepts an `HttpClient` parameter in its constructor:

```
public class GitHubService
{
    public HttpClient Client { get; }

    public GitHubService(HttpClient client)
    {
        client.BaseAddress = new Uri("https://api.github.com/");
        // GitHub API versioning
        client.DefaultRequestHeaders.Add("Accept",
            "application/vnd.github.v3+json");
        // GitHub requires a user-agent
        client.DefaultRequestHeaders.Add("User-Agent",
            "HttpClientFactory-Sample");

        Client = client;
    }

    public async Task<IEnumerable<GitHubIssue>> GetAspNetDocsIssues()
    {
        var response = await Client.GetAsync(
            "/repos/aspnet/AspNetCore.Docs/issues?state=open&sort=created&direction=desc");

        response.EnsureSuccessStatusCode();

        using var responseStream = await response.Content.ReadAsStreamAsync();
        return await JsonSerializer.DeserializeAsync
            <IEnumerable<GitHubIssue>>(responseStream);
    }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

In the preceding code:

- The configuration is moved into the typed client.
- The `HttpClient` object is exposed as a public property.

API-specific methods can be created that expose `HttpClient` functionality. For example, the `GetAspNetDocsIssues` method encapsulates code to retrieve open issues.

The following code calls AddHttpClient in `Startup.ConfigureServices` to register a typed client class:

```
services.AddHttpClient<GitHubService>();
```

The typed client is registered as transient with DI. In the preceding code, `AddHttpClient` registers `GitHubService` as a transient service. This registration uses a factory method to:

1. Create an instance of `HttpClient`.
2. Create an instance of `GitHubService`, passing in the instance of `HttpClient` to its constructor.

The typed client can be injected and consumed directly:

```
public class TypedClientModel : PageModel
{
    private readonly GitHubService _gitHubService;

    public IEnumerable<GitHubIssue> LatestIssues { get; private set; }

    public bool HasIssue => LatestIssues.Any();

    public bool GetIssuesError { get; private set; }

    public TypedClientModel(GitHubService gitHubService)
    {
        _gitHubService = gitHubService;
    }

    public async Task OnGet()
    {
        try
        {
            LatestIssues = await _gitHubService.GetAspNetDocsIssues();
        }
        catch(HttpRequestException)
        {
            GetIssuesError = true;
            LatestIssues = Array.Empty<GitHubIssue>();
        }
    }
}
```

The configuration for a typed client can be specified during registration in `Startup.ConfigureServices`, rather than in the typed client's constructor:

```
services.AddHttpClient<RepoService>(c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

The `HttpClient` can be encapsulated within a typed client. Rather than exposing it as a property, define a method which calls the `HttpClient` instance internally:

```
public class RepoService
{
    // _httpClient isn't exposed publicly
    private readonly HttpClient _httpClient;

    public RepoService(HttpClient client)
    {
        _httpClient = client;
    }

    public async Task<IEnumerable<string>> GetRepos()
    {
        var response = await _httpClient.GetAsync("aspnet/repos");

        response.EnsureSuccessStatusCode();

        using var responseStream = await response.Content.ReadAsStreamAsync();
        return await JsonSerializer.DeserializeAsync
            <IEnumerable<string>>(responseStream);
    }
}
```

In the preceding code, the `HttpClient` is stored in a private field. Access to the `HttpClient` is by the public `GetRepos` method.

Generated clients

`IHttpClientFactory` can be used in combination with third-party libraries such as Refit. Refit is a REST library for .NET. It converts REST APIs into live interfaces. An implementation of the interface is generated dynamically by the `RestService`, using `HttpClient` to make the external HTTP calls.

An interface and a reply are defined to represent the external API and its response:

```
public interface IHelloClient
{
    [Get("/helloworld")]
    Task<Reply> GetMessageAsync();
}

public class Reply
{
    public string Message { get; set; }
}
```

A typed client can be added, using Refit to generate the implementation:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("hello", c =>
    {
        c.BaseAddress = new Uri("http://localhost:5000");
    })
    .AddTypedClient(c => Refit.RestService.For<IHelloClient>(c));

    services.AddControllers();
}
```

The defined interface can be consumed where necessary, with the implementation provided by DI and Refit:

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHelloClient _client;

    public ValuesController(IHelloClient client)
    {
        _client = client;
    }

    [HttpGet("/")]
    public async Task<ActionResult<Reply>> Index()
    {
        return await _client.GetMessageAsync();
    }
}
```

Make POST, PUT, and DELETE requests

In the preceding examples, all HTTP requests use the GET HTTP verb. `HttpClient` also supports other HTTP verbs, including:

- POST
- PUT
- DELETE
- PATCH

For a complete list of supported HTTP verbs, see HttpMethod.

The following example shows how to make an HTTP POST request:

```
public async Task CreateItemAsync(TodoItem todoItem)
{
    var todoItemJson = new StringContent(
        JsonSerializer.Serialize(todoItem, _jsonSerializerOptions),
        Encoding.UTF8,
        "application/json");

    using var httpResponse =
        await _httpClient.PostAsync("/api/TodoItems", todoItemJson);

    httpResponse.EnsureSuccessStatusCode();
}
```

In the preceding code, the `CreateItemAsync` method:

- Serializes the `TodoItem` parameter to JSON using `System.Text.Json`. This uses an instance of JsonSerializerOptions to configure the serialization process.
- Creates an instance of StringContent to package the serialized JSON for sending in the HTTP request's body.
- Calls PostAsync to send the JSON content to the specified URL. This is a relative URL that gets added to the HttpClient.BaseAddress.
- Calls EnsureSuccessStatusCode to throw an exception if the response status code does not indicate success.

`HttpClient` also supports other types of content. For example, MultipartContent and StreamContent. For a complete list of supported content, see HttpContent.

The following example shows an HTTP PUT request:

```
public async Task SaveItemAsync(TodoItem todoItem)
{
    var todoItemJson = new StringContent(
        JsonSerializer.Serialize(todoItem),
        Encoding.UTF8,
        "application/json");

    using var httpResponse =
        await _httpClient.PutAsync($"/api/TodoItems/{todoItem.Id}", todoItemJson);

    httpResponse.EnsureSuccessStatusCode();
}
```

The preceding code is very similar to the POST example. The `SaveItemAsync` method calls PutAsync instead of `PostAsync`.

The following example shows an HTTP DELETE request:

```
public async Task DeleteItemAsync(long itemId)
{
    using var httpResponse =
        await _httpClient.DeleteAsync($"/api/TodoItems/{itemId}");

    httpResponse.EnsureSuccessStatusCode();
}
```

In the preceding code, the `DeleteItemAsync` method calls DeleteAsync. Because HTTP DELETE requests typically contain no body, the `DeleteAsync` method doesn't provide an overload that accepts an instance of `HttpContent`.

To learn more about using different HTTP verbs with `HttpClient`, see HttpClient.

Outgoing request middleware

`HttpClient` has the concept of delegating handlers that can be linked together for outgoing HTTP requests. `IHttpClientFactory`:

- Simplifies defining the handlers to apply for each named client.
- Supports registration and chaining of multiple handlers to build an outgoing request middleware pipeline. Each of these handlers is able to perform work before and after the outgoing request. This pattern:
  - Is similar to the inbound middleware pipeline in ASP.NET Core.
  - Provides a mechanism to manage cross-cutting concerns around HTTP requests, such as:
    - caching
    - error handling
    - serialization
    - logging

To create a delegating handler:

- Derive from DelegatingHandler.
- Override SendAsync. Execute code before passing the request to the next handler in the pipeline:

```
public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest)
            {
                Content = new StringContent(
                    "You must supply an API key header called X-API-KEY")
            };
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

The preceding code checks if the `X-API-KEY` header is in the request. If `X-API-KEY` is missing, BadRequest is returned.

More than one handler can be added to the configuration for an `HttpClient` with Microsoft.Extensions.DependencyInjection.HttpClientBuilderExtensions.AddHttpMessageHandler:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<ValidateHeaderHandler>();

    services.AddHttpClient("externalservice", c =>
    {
        // Assume this is an "external" service which requires an API KEY
        c.BaseAddress = new Uri("https://localhost:5001/");
    })
    .AddHttpMessageHandler<ValidateHeaderHandler>();

    // Remaining code deleted for brevity.
```

In the preceding code, the `ValidateHeaderHandler` is registered with DI. The `IHttpClientFactory` creates a separate DI scope for each handler. Handlers can depend upon services of any scope. Services that handlers depend upon are disposed when the handler is disposed.

Once registered, AddHttpMessageHandler can be called, passing in the type for the handler.

Multiple handlers can be registered in the order that they should execute. Each handler wraps the next handler until the final `HttpClientHandler` executes the request:

```
services.AddTransient<SecureRequestHandler>();
services.AddTransient<RequestDataHandler>();

services.AddHttpClient("clientwithhandlers")
    // This handler is on the outside and called first during the
    // request, last during the response.
    .AddHttpMessageHandler<SecureRequestHandler>()
    // This handler is on the inside, closest to the request being
    // sent.
    .AddHttpMessageHandler<RequestDataHandler>();
```

Use one of the following approaches to share per-request state with message handlers:

- Pass data into the handler using HttpRequestMessage.Properties.

- Use IHttpContextAccessor to access the current request.
- Create a custom AsyncLocal<T> storage object to pass the data.

Use Polly-based handlers

`IHttpClientFactory` integrates with the third-party library Polly. Polly is a comprehensive resilience and transient fault-handling library for .NET. It allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Extension methods are provided to enable the use of Polly policies with configured `HttpClient` instances. The Polly extensions support adding Polly-based handlers to clients. Polly requires the Microsoft.Extensions.Http.Polly NuGet package.

Handle transient faults

Faults typically occur when external HTTP calls are transient. AddTransientHttpErrorPolicy allows a policy to be defined to handle transient errors. Policies configured with `AddTransientHttpErrorPolicy` handle the following responses:

- HttpRequestException
- HTTP 5xx
- HTTP 408

`AddTransientHttpErrorPolicy` provides access to a `PolicyBuilder` object configured to handle errors representing a possible transient fault:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<UnreliableEndpointCallerService>()
        .AddTransientHttpErrorPolicy(p =>
            p.WaitAndRetryAsync(3, _ => TimeSpan.FromMilliseconds(600)));

    // Remaining code deleted for brevity.
```

In the preceding code, a `WaitAndRetryAsync` policy is defined. Failed requests are retried up to three times with a delay of 600 ms between attempts.

Dynamically select policies

Extension methods are provided to add Polly-based handlers, for example, AddPolicyHandler. The following `AddPolicyHandler` overload inspects the request to decide which policy to apply:

```
var timeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

services.AddHttpClient("conditionalpolicy")
// Run some code to select a policy based on the request
    .AddPolicyHandler(request =>
        request.Method == HttpMethod.Get ? timeout : longTimeout);
```

In the preceding code, if the outgoing request is an HTTP GET, a 10-second timeout is applied. For any other HTTP method, a 30-second timeout is used.

Add multiple Polly handlers

It's common to nest Polly policies:

```
services.AddHttpClient("multiplepolicies")
    .AddTransientHttpErrorPolicy(p => p.RetryAsync(3))
    .AddTransientHttpErrorPolicy(
        p => p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```

In the preceding example:

- Two handlers are added.
- The first handler uses AddTransientHttpErrorPolicy to add a retry policy. Failed requests are retried up to three times.
- The second `AddTransientHttpErrorPolicy` call adds a circuit breaker policy. Further external requests are blocked for 30 seconds if 5 failed attempts occur sequentially. Circuit breaker policies are stateful. All calls through this client share the same circuit state.

Add policies from the Polly registry

An approach to managing regularly used policies is to define them once and register them with a `PolicyRegistry`
.

In the following code:

- The "regular" and "long" policies are added.
- AddPolicyHandlerFromRegistry adds the "regular" and "long" policies from the registry.

```
public void ConfigureServices(IServiceCollection services)
{
    var timeout = Policy.TimeoutAsync<HttpResponseMessage>(
        TimeSpan.FromSeconds(10));
    var longTimeout = Policy.TimeoutAsync<HttpResponseMessage>(
        TimeSpan.FromSeconds(30));

    var registry = services.AddPolicyRegistry();

    registry.Add("regular", timeout);
    registry.Add("long", longTimeout);

    services.AddHttpClient("regularTimeoutHandler")
        .AddPolicyHandlerFromRegistry("regular");

    services.AddHttpClient("longTimeoutHandler")
       .AddPolicyHandlerFromRegistry("long");

    // Remaining code deleted for brevity.
```

For more information on `IHttpClientFactory` and Polly integrations, see the Polly wiki.

HttpClient and lifetime management

A new `HttpClient` instance is returned each time `CreateClient` is called on the `IHttpClientFactory`. An HttpMessageHandler is created per named client. The factory manages the lifetimes of the `HttpMessageHandler` instances.

`IHttpClientFactory` pools the `HttpMessageHandler` instances created by the factory to reduce resource consumption. An `HttpMessageHandler` instance may be reused from the pool when creating a new `HttpClient` instance if its lifetime hasn't expired.

Pooling of handlers is desirable as each handler typically manages its own underlying HTTP connections. Creating more handlers than necessary can result in connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS (Domain Name System) changes.

The default handler lifetime is two minutes. The default value can be overridden on a per named client basis:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("extendedhandlerlifetime")
        .SetHandlerLifetime(TimeSpan.FromMinutes(5));

    // Remaining code deleted for brevity.
```

`HttpClient` instances can generally be treated as .NET objects **not** requiring disposal. Disposal cancels outgoing requests and guarantees the given `HttpClient` instance can't be used after calling Dispose. `IHttpClientFactory` tracks and disposes resources used by `HttpClient` instances.

Keeping a single `HttpClient` instance alive for a long duration is a common pattern used before the inception of `IHttpClientFactory`. This pattern becomes unnecessary after migrating to `IHttpClientFactory`.

Alternatives to IHttpClientFactory

Using `IHttpClientFactory` in a DI-enabled app avoids:

- Resource exhaustion problems by pooling `HttpMessageHandler` instances.
- Stale DNS problems by cycling `HttpMessageHandler` instances at regular intervals.

There are alternative ways to solve the preceding problems using a long-lived SocketsHttpHandler instance.

- Create an instance of `SocketsHttpHandler` when the app starts and use it for the life of the app.
- Configure PooledConnectionLifetime to an appropriate value based on DNS refresh times.
- Create `HttpClient` instances using `new HttpClient(handler, disposeHandler: false)` as needed.

The preceding approaches solve the resource management problems that `IHttpClientFactory` solves in a similar way.

- The `SocketsHttpHandler` shares connections across `HttpClient` instances. This sharing prevents socket exhaustion.
- The `SocketsHttpHandler` cycles connections according to `PooledConnectionLifetime` to avoid stale DNS problems.

Cookies

The pooled `HttpMessageHandler` instances results in `CookieContainer` objects being shared. Unanticipated `CookieContainer` object sharing often results in incorrect code. For apps that require cookies, consider either:

- Disabling automatic cookie handling
- Avoiding `IHttpClientFactory`

Call ConfigurePrimaryHttpMessageHandler to disable automatic cookie handling:

```
services.AddHttpClient("configured-disable-automatic-cookies")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new SocketsHttpHandler()
        {
            UseCookies = false,
        };
    });
```

Logging

Clients created via `IHttpClientFactory` record log messages for all requests. Enable the appropriate information level in the logging configuration to see the default log messages. Additional logging, such as the logging of request headers, is only included at trace level.

The log category used for each client includes the name of the client. A client named *MyNamedClient*, for example, logs messages with a category of "System.Net.Http.HttpClient.**MyNamedClient**.LogicalHandler". Messages suffixed with *LogicalHandler* occur outside the request handler pipeline. On the request, messages are logged before any other handlers in the pipeline have processed it. On the response, messages are logged after any other pipeline handlers have received the response.

Logging also occurs inside the request handler pipeline. In the *MyNamedClient* example, those messages are logged with the log category "System.Net.Http.HttpClient.**MyNamedClient**.ClientHandler". For the request, this occurs after all other handlers have run and immediately before the request is sent. On the response, this logging includes the state of the response before it passes back through the handler pipeline.

Enabling logging outside and inside the pipeline enables inspection of the changes made by the other pipeline handlers. This may include changes to request headers or to the response status code.

Including the name of the client in the log category enables log filtering for specific named clients.

Configure the HttpMessageHandler

It may be necessary to control the configuration of the inner `HttpMessageHandler` used by a client.

An `IHttpClientBuilder` is returned when adding named or typed clients. The ConfigurePrimaryHttpMessageHandler extension method can be used to define a delegate. The delegate is used to create and configure the primary `HttpMessageHandler` used by that client:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("configured-inner-handler")
        .ConfigurePrimaryHttpMessageHandler(() =>
        {
            return new HttpClientHandler()
            {
                AllowAutoRedirect = false,
                UseDefaultCredentials = true
            };
        });

    // Remaining code deleted for brevity.
```

Use IHttpClientFactory in a console app

In a console app, add the following package references to the project:

- Microsoft.Extensions.Hosting
- Microsoft.Extensions.Http

In the following example:

- IHttpClientFactory is registered in the Generic Host's service container.
- `MyService` creates a client factory instance from the service, which is used to create an `HttpClient`. `HttpClient` is used to retrieve a webpage.
- `Main` creates a scope to execute the service's `GetPage` method and write the first 500 characters of the webpage content to the console.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
class Program
{
    static async Task<int> Main(string[] args)
    {
        var builder = new HostBuilder()
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHttpClient();
                services.AddTransient<IMyService, MyService>();
            }).UseConsoleLifetime();

        var host = builder.Build();

        using (var serviceScope = host.Services.CreateScope())
        {
            var services = serviceScope.ServiceProvider;

            try
            {
                var myService = services.GetRequiredService<IMyService>();
                var pageContent = await myService.GetPage();

                Console.WriteLine(pageContent.Substring(0, 500));
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();

                logger.LogError(ex, "An error occurred.");
            }
        }

        return 0;
    }

    public interface IMyService
    {
        Task<string> GetPage();
    }

    public class MyService : IMyService
    {
        private readonly IHttpClientFactory _clientFactory;

        public MyService(IHttpClientFactory clientFactory)
        {
            _clientFactory = clientFactory;
        }

        public async Task<string> GetPage()
        {
            // Content from BBC One: Dr. Who website (©BBC)
            var request = new HttpRequestMessage(HttpMethod.Get,
                "https://www.bbc.co.uk/programmes/b006q2x0");
            var client = _clientFactory.CreateClient();
            var response = await client.SendAsync(request);

            if (response.IsSuccessStatusCode)
            {
                return await response.Content.ReadAsStringAsync();
            }
            else
            {
```

```
                return $"StatusCode: {response.StatusCode}";
            }
        }
    }
}
```

Header propagation middleware

Header propagation is an ASP.NET Core middleware to propagate HTTP headers from the incoming request to the outgoing HTTP Client requests. To use header propagation:

- Reference the Microsoft.AspNetCore.HeaderPropagation package.
- Configure the middleware and `HttpClient` in `Startup`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    services.AddHttpClient("MyForwardingClient").AddHeaderPropagation();
    services.AddHeaderPropagation(options =>
    {
        options.Headers.Add("X-TraceId");
    });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseHeaderPropagation();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

- The client includes the configured headers on outbound requests:

```
var client = clientFactory.CreateClient("MyForwardingClient");
var response = client.GetAsync(...);
```

Additional resources

- Use HttpClientFactory to implement resilient HTTP requests
- Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies
- Implement the Circuit Breaker pattern
- How to serialize and deserialize JSON in .NET

By Glenn Condron, Ryan Nowak, and Steve Gordon

An IHttpClientFactory can be registered and used to configure and create HttpClient instances in an app. It offers the following benefits:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, a *github* client can be registered and configured to access GitHub. A default client can be registered for other purposes.
- Codifies the concept of outgoing middleware via delegating handlers in `HttpClient` and provides extensions for Polly-based middleware to take advantage of that.
- Manages the pooling and lifetime of underlying `HttpClientMessageHandler` instances to avoid common DNS problems that occur when manually managing `HttpClient` lifetimes.
- Adds a configurable logging experience (via `ILogger`) for all requests sent through clients created by the factory.

View or download sample code (how to download)

Consumption patterns

There are several ways `IHttpClientFactory` can be used in an app:

- Basic usage
- Named clients
- Typed clients
- Generated clients

None of them are strictly superior to another. The best approach depends upon the app's constraints.

Basic usage

The `IHttpClientFactory` can be registered by calling the `AddHttpClient` extension method on the `IServiceCollection`, inside the `Startup.ConfigureServices` method.

```
services.AddHttpClient();
```

Once registered, code can accept an `IHttpClientFactory` anywhere services can be injected with dependency injection (DI). The `IHttpClientFactory` can be used to create an `HttpClient` instance:

```
public class BasicUsageModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubBranch> Branches { get; private set; }

    public bool GetBranchesError { get; private set; }

    public BasicUsageModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "https://api.github.com/repos/aspnet/AspNetCore.Docs/branches");
        request.Headers.Add("Accept", "application/vnd.github.v3+json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        var client = _clientFactory.CreateClient();

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            Branches = await response.Content
                .ReadAsAsync<IEnumerable<GitHubBranch>>();
        }
        else
        {
            GetBranchesError = true;
            Branches = Array.Empty<GitHubBranch>();
        }
    }
}
```

Using `IHttpClientFactory` in this fashion is a good way to refactor an existing app. It has no impact on the way `HttpClient` is used. In places where `HttpClient` instances are currently created, replace those occurrences with a call to CreateClient.

Named clients

If an app requires many distinct uses of `HttpClient`, each with a different configuration, an option is to use **named clients**. Configuration for a named `HttpClient` can be specified during registration in `Startup.ConfigureServices`.

```
services.AddHttpClient("github", c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    // Github API versioning
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    // Github requires a user-agent
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

In the preceding code, `AddHttpClient` is called, providing the name *github*. This client has some default configuration applied—namely the base address and two headers required to work with the GitHub API.

Each time `CreateClient` is called, a new instance of `HttpClient` is created and the configuration action is called.

To consume a named client, a string parameter can be passed to `CreateClient`. Specify the name of the client to be created:

```
public class NamedClientModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubPullRequest> PullRequests { get; private set; }

    public bool GetPullRequestsError { get; private set; }

    public bool HasPullRequests => PullRequests.Any();

    public NamedClientModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "repos/aspnet/AspNetCore.Docs/pulls");

        var client = _clientFactory.CreateClient("github");

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            PullRequests = await response.Content
                .ReadAsAsync<IEnumerable<GitHubPullRequest>>();
        }
        else
        {
            GetPullRequestsError = true;
            PullRequests = Array.Empty<GitHubPullRequest>();
        }
    }
}
```

In the preceding code, the request doesn't need to specify a hostname. It can pass just the path, since the base address configured for the client is used.

Typed clients

Typed clients:

- Provide the same capabilities as named clients without the need to use strings as keys.
- Provides IntelliSense and compiler help when consuming clients.
- Provide a single location to configure and interact with a particular `HttpClient`. For example, a single typed client might be used for a single backend endpoint and encapsulate all logic dealing with that endpoint.
- Work with DI and can be injected where required in your app.

A typed client accepts an `HttpClient` parameter in its constructor:

```
public class GitHubService
{
    public HttpClient Client { get; }

    public GitHubService(HttpClient client)
    {
        client.BaseAddress = new Uri("https://api.github.com/");
        // GitHub API versioning
        client.DefaultRequestHeaders.Add("Accept",
            "application/vnd.github.v3+json");
        // GitHub requires a user-agent
        client.DefaultRequestHeaders.Add("User-Agent",
            "HttpClientFactory-Sample");

        Client = client;
    }

    public async Task<IEnumerable<GitHubIssue>> GetAspNetDocsIssues()
    {
        var response = await Client.GetAsync(
            "/repos/aspnet/AspNetCore.Docs/issues?state=open&sort=created&direction=desc");

        response.EnsureSuccessStatusCode();

        var result = await response.Content
            .ReadAsAsync<IEnumerable<GitHubIssue>>();

        return result;
    }
}
```

In the preceding code, the configuration is moved into the typed client. The `HttpClient` object is exposed as a public property. It's possible to define API-specific methods that expose `HttpClient` functionality. The `GetAspNetDocsIssues` method encapsulates the code needed to query for and parse out the latest open issues from a GitHub repository.

To register a typed client, the generic AddHttpClient extension method can be used within `Startup.ConfigureServices`, specifying the typed client class:

```
services.AddHttpClient<GitHubService>();
```

The typed client is registered as transient with DI. The typed client can be injected and consumed directly:

```
public class TypedClientModel : PageModel
{
    private readonly GitHubService _gitHubService;

    public IEnumerable<GitHubIssue> LatestIssues { get; private set; }

    public bool HasIssue => LatestIssues.Any();

    public bool GetIssuesError { get; private set; }

    public TypedClientModel(GitHubService gitHubService)
    {
        _gitHubService = gitHubService;
    }

    public async Task OnGet()
    {
        try
        {
            LatestIssues = await _gitHubService.GetAspNetDocsIssues();
        }
        catch(HttpRequestException)
        {
            GetIssuesError = true;
            LatestIssues = Array.Empty<GitHubIssue>();
        }
    }
}
```

If preferred, the configuration for a typed client can be specified during registration in `Startup.ConfigureServices`, rather than in the typed client's constructor:

```
services.AddHttpClient<RepoService>(c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

It's possible to entirely encapsulate the `HttpClient` within a typed client. Rather than exposing it as a property, public methods can be provided which call the `HttpClient` instance internally.

```
public class RepoService
{
    // _httpClient isn't exposed publicly
    private readonly HttpClient _httpClient;

    public RepoService(HttpClient client)
    {
        _httpClient = client;
    }

    public async Task<IEnumerable<string>> GetRepos()
    {
        var response = await _httpClient.GetAsync("aspnet/repos");

        response.EnsureSuccessStatusCode();

        var result = await response.Content
            .ReadAsAsync<IEnumerable<string>>();

        return result;
    }
}
```

In the preceding code, the `HttpClient` is stored as a private field. All access to make external calls goes through the `GetRepos` method.

Generated clients

`IHttpClientFactory` can be used in combination with other third-party libraries such as Refit. Refit is a REST library for .NET. It converts REST APIs into live interfaces. An implementation of the interface is generated dynamically by the `RestService`, using `HttpClient` to make the external HTTP calls.

An interface and a reply are defined to represent the external API and its response:

```
public interface IHelloClient
{
    [Get("/helloworld")]
    Task<Reply> GetMessageAsync();
}

public class Reply
{
    public string Message { get; set; }
}
```

A typed client can be added, using Refit to generate the implementation:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("hello", c =>
    {
        c.BaseAddress = new Uri("https://localhost:5001");
    })
    .AddTypedClient(c => Refit.RestService.For<IHelloClient>(c));

    services.AddMvc();
}
```

The defined interface can be consumed where necessary, with the implementation provided by DI and Refit:

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHelloClient _client;

    public ValuesController(IHelloClient client)
    {
        _client = client;
    }

    [HttpGet("/")]
    public async Task<ActionResult<Reply>> Index()
    {
        return await _client.GetMessageAsync();
    }
}
```

Outgoing request middleware

`HttpClient` already has the concept of delegating handlers that can be linked together for outgoing HTTP requests. The `IHttpClientFactory` makes it easy to define the handlers to apply for each named client. It supports registration and chaining of multiple handlers to build an outgoing request middleware pipeline. Each of these handlers is able to perform work before and after the outgoing request. This pattern is similar to the inbound middleware pipeline in ASP.NET Core. The pattern provides a mechanism to manage cross-cutting concerns around HTTP requests, including caching, error handling, serialization, and logging.

To create a handler, define a class deriving from DelegatingHandler. Override the `SendAsync` method to execute code before passing the request to the next handler in the pipeline:

```
public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest)
            {
                Content = new StringContent(
                    "You must supply an API key header called X-API-KEY")
            };
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

The preceding code defines a basic handler. It checks to see if an `X-API-KEY` header has been included on the request. If the header is missing, it can avoid the HTTP call and return a suitable response.

During registration, one or more handlers can be added to the configuration for an `HttpClient`. This task is accomplished via extension methods on the IHttpClientBuilder.

```
services.AddTransient<ValidateHeaderHandler>();

services.AddHttpClient("externalservice", c =>
{
    // Assume this is an "external" service which requires an API KEY
    c.BaseAddress = new Uri("https://localhost:5000/");
})
.AddHttpMessageHandler<ValidateHeaderHandler>();
```

In the preceding code, the `ValidateHeaderHandler` is registered with DI. The `IHttpClientFactory` creates a separate DI scope for each handler. Handlers are free to depend upon services of any scope. Services that handlers depend upon are disposed when the handler is disposed.

Once registered, AddHttpMessageHandler can be called, passing in the type for the handler.

Multiple handlers can be registered in the order that they should execute. Each handler wraps the next handler until the final `HttpClientHandler` executes the request:

```
services.AddTransient<SecureRequestHandler>();
services.AddTransient<RequestDataHandler>();

services.AddHttpClient("clientwithhandlers")
    // This handler is on the outside and called first during the
    // request, last during the response.
    .AddHttpMessageHandler<SecureRequestHandler>()
    // This handler is on the inside, closest to the request being
    // sent.
    .AddHttpMessageHandler<RequestDataHandler>();
```

Use one of the following approaches to share per-request state with message handlers:

- Pass data into the handler using `HttpRequestMessage.Properties`.
- Use `IHttpContextAccessor` to access the current request.
- Create a custom `AsyncLocal` storage object to pass the data.

Use Polly-based handlers

`IHttpClientFactory` integrates with a popular third-party library called Polly. Polly is a comprehensive resilience and transient fault-handling library for .NET. It allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Extension methods are provided to enable the use of Polly policies with configured `HttpClient` instances. The Polly extensions:

- Support adding Polly-based handlers to clients.
- Can be used after installing the Microsoft.Extensions.Http.Polly NuGet package. The package isn't included in the ASP.NET Core shared framework.

Handle transient faults

Most common faults occur when external HTTP calls are transient. A convenient extension method called `AddTransientHttpErrorPolicy` is included which allows a policy to be defined to handle transient errors. Policies configured with this extension method handle `HttpRequestException`, HTTP 5xx responses, and HTTP 408 responses.

The `AddTransientHttpErrorPolicy` extension can be used within `Startup.ConfigureServices`. The extension provides access to a `PolicyBuilder` object configured to handle errors representing a possible transient fault:

```
services.AddHttpClient<UnreliableEndpointCallerService>()
    .AddTransientHttpErrorPolicy(p =>
        p.WaitAndRetryAsync(3, _ => TimeSpan.FromMilliseconds(600)));
```

In the preceding code, a `WaitAndRetryAsync` policy is defined. Failed requests are retried up to three times with a delay of 600 ms between attempts.

Dynamically select policies

Additional extension methods exist which can be used to add Polly-based handlers. One such extension is `AddPolicyHandler`, which has multiple overloads. One overload allows the request to be inspected when defining which policy to apply:

```
var timeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

services.AddHttpClient("conditionalpolicy")
// Run some code to select a policy based on the request
    .AddPolicyHandler(request =>
        request.Method == HttpMethod.Get ? timeout : longTimeout);
```

In the preceding code, if the outgoing request is an HTTP GET, a 10-second timeout is applied. For any other HTTP method, a 30-second timeout is used.

Add multiple Polly handlers

It's common to nest Polly policies to provide enhanced functionality:

```
services.AddHttpClient("multiplepolicies")
    .AddTransientHttpErrorPolicy(p => p.RetryAsync(3))
    .AddTransientHttpErrorPolicy(
        p => p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```

In the preceding example, two handlers are added. The first uses the `AddTransientHttpErrorPolicy` extension to add a retry policy. Failed requests are retried up to three times. The second call to `AddTransientHttpErrorPolicy` adds a circuit breaker policy. Further external requests are blocked for 30 seconds if five failed attempts occur sequentially. Circuit breaker policies are stateful. All calls through this client share the same circuit state.

Add policies from the Polly registry

An approach to managing regularly used policies is to define them once and register them with a `PolicyRegistry`. An extension method is provided which allows a handler to be added using a policy from the registry:

```
var registry = services.AddPolicyRegistry();

registry.Add("regular", timeout);
registry.Add("long", longTimeout);

services.AddHttpClient("regulartimeouthandler")
    .AddPolicyHandlerFromRegistry("regular");
```

In the preceding code, two policies are registered when the `PolicyRegistry` is added to the `ServiceCollection`. To use a policy from the registry, the `AddPolicyHandlerFromRegistry` method is used, passing the name of the policy to apply.

Further information about `IHttpClientFactory` and Polly integrations can be found on the Polly wiki.

HttpClient and lifetime management

A new `HttpClient` instance is returned each time `CreateClient` is called on the `IHttpClientFactory`. There's an HttpMessageHandler per named client. The factory manages the lifetimes of the `HttpMessageHandler` instances.

`IHttpClientFactory` pools the `HttpMessageHandler` instances created by the factory to reduce resource consumption. An `HttpMessageHandler` instance may be reused from the pool when creating a new `HttpClient` instance if its lifetime hasn't expired.

Pooling of handlers is desirable as each handler typically manages its own underlying HTTP connections. Creating more handlers than necessary can result in connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS changes.

The default handler lifetime is two minutes. The default value can be overridden on a per named client basis. To override it, call SetHandlerLifetime on the `IHttpClientBuilder` that is returned when creating the client:

```
services.AddHttpClient("extendedhandlerlifetime")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

Disposal of the client isn't required. Disposal cancels outgoing requests and guarantees the given `HttpClient` instance can't be used after calling Dispose. `IHttpClientFactory` tracks and disposes resources used by `HttpClient` instances. The `HttpClient` instances can generally be treated as .NET objects not requiring disposal.

Keeping a single `HttpClient` instance alive for a long duration is a common pattern used before the inception of `IHttpClientFactory`. This pattern becomes unnecessary after migrating to `IHttpClientFactory`.

Alternatives to IHttpClientFactory

Using `IHttpClientFactory` in a DI-enabled app avoids:

- Resource exhaustion problems by pooling `HttpMessageHandler` instances.
- Stale DNS problems by cycling `HttpMessageHandler` instances at regular intervals.

There are alternative ways to solve the preceding problems using a long-lived SocketsHttpHandler instance.

- Create an instance of `SocketsHttpHandler` when the app starts and use it for the life of the app.
- Configure PooledConnectionLifetime to an appropriate value based on DNS refresh times.
- Create `HttpClient` instances using `new HttpClient(handler, disposeHandler: false)` as needed.

The preceding approaches solve the resource management problems that `IHttpClientFactory` solves in a similar way.

- The `SocketsHttpHandler` shares connections across `HttpClient` instances. This sharing prevents socket exhaustion.
- The `SocketsHttpHandler` cycles connections according to `PooledConnectionLifetime` to avoid stale DNS problems.

Cookies

The pooled `HttpMessageHandler` instances results in `CookieContainer` objects being shared. Unanticipated `CookieContainer` object sharing often results in incorrect code. For apps that require cookies, consider either:

- Disabling automatic cookie handling
- Avoiding `IHttpClientFactory`

Call ConfigurePrimaryHttpMessageHandler to disable automatic cookie handling:

```
services.AddHttpClient("configured-disable-automatic-cookies")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new SocketsHttpHandler()
        {
            UseCookies = false,
        };
    });
```

Logging

Clients created via `IHttpClientFactory` record log messages for all requests. Enable the appropriate information level in your logging configuration to see the default log messages. Additional logging, such as the logging of request headers, is only included at trace level.

The log category used for each client includes the name of the client. A client named *MyNamedClient*, for example, logs messages with a category of `System.Net.Http.HttpClient.MyNamedClient.LogicalHandler`. Messages suffixed with *LogicalHandler* occur outside the request handler pipeline. On the request, messages are logged before any other handlers in the pipeline have processed it. On the response, messages are logged after any other pipeline handlers have received the response.

Logging also occurs inside the request handler pipeline. In the *MyNamedClient* example, those messages are logged against the log category `System.Net.Http.HttpClient.MyNamedClient.ClientHandler`. For the request, this occurs after all other handlers have run and immediately before the request is sent out on the network. On the response, this logging includes the state of the response before it passes back through the handler pipeline.

Enabling logging outside and inside the pipeline enables inspection of the changes made by the other pipeline handlers. This may include changes to request headers, for example, or to the response status code.

Including the name of the client in the log category enables log filtering for specific named clients where necessary.

Configure the HttpMessageHandler

It may be necessary to control the configuration of the inner `HttpMessageHandler` used by a client.

An `IHttpClientBuilder` is returned when adding named or typed clients. The ConfigurePrimaryHttpMessageHandler extension method can be used to define a delegate. The delegate is used to create and configure the primary `HttpMessageHandler` used by that client:

```
services.AddHttpClient("configured-inner-handler")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new HttpClientHandler()
        {
            AllowAutoRedirect = false,
            UseDefaultCredentials = true
        };
    });
```

Use IHttpClientFactory in a console app

In a console app, add the following package references to the project:

- Microsoft.Extensions.Hosting
- Microsoft.Extensions.Http

In the following example:

- IHttpClientFactory is registered in the Generic Host's service container.
- `MyService` creates a client factory instance from the service, which is used to create an `HttpClient`. `HttpClient` is used to retrieve a webpage.
- `Main` creates a scope to execute the service's `GetPage` method and write the first 500 characters of the webpage content to the console.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
class Program
{
    static async Task<int> Main(string[] args)
    {
        var builder = new HostBuilder()
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHttpClient();
                services.AddTransient<IMyService, MyService>();
            }).UseConsoleLifetime();

        var host = builder.Build();

        using (var serviceScope = host.Services.CreateScope())
        {
            var services = serviceScope.ServiceProvider;

            try
            {
                var myService = services.GetRequiredService<IMyService>();
                var pageContent = await myService.GetPage();

                Console.WriteLine(pageContent.Substring(0, 500));
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();

                logger.LogError(ex, "An error occurred.");
            }
        }

        return 0;
    }

    public interface IMyService
    {
        Task<string> GetPage();
    }

    public class MyService : IMyService
    {
        private readonly IHttpClientFactory _clientFactory;

        public MyService(IHttpClientFactory clientFactory)
        {
            _clientFactory = clientFactory;
        }

        public async Task<string> GetPage()
        {
            // Content from BBC One: Dr. Who website (©BBC)
```

```
            var request = new HttpRequestMessage(HttpMethod.Get,
                "https://www.bbc.co.uk/programmes/b006q2x0");
            var client = _clientFactory.CreateClient();
            var response = await client.SendAsync(request);

            if (response.IsSuccessStatusCode)
            {
                return await response.Content.ReadAsStringAsync();
            }
            else
            {
                return $"StatusCode: {response.StatusCode}";
            }
        }
    }
}
```

Additional resources

- Use HttpClientFactory to implement resilient HTTP requests
- Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies
- Implement the Circuit Breaker pattern

By Glenn Condron, Ryan Nowak, and Steve Gordon

An IHttpClientFactory can be registered and used to configure and create HttpClient instances in an app. It offers the following benefits:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, a *github* client can be registered and configured to access GitHub. A default client can be registered for other purposes.
- Codifies the concept of outgoing middleware via delegating handlers in `HttpClient` and provides extensions for Polly-based middleware to take advantage of that.
- Manages the pooling and lifetime of underlying `HttpClientMessageHandler` instances to avoid common DNS problems that occur when manually managing `HttpClient` lifetimes.
- Adds a configurable logging experience (via `ILogger`) for all requests sent through clients created by the factory.

View or download sample code (how to download)

Prerequisites

Projects targeting .NET Framework require installation of the Microsoft.Extensions.Http NuGet package. Projects that target .NET Core and reference the Microsoft.AspNetCore.App metapackage already include the `Microsoft.Extensions.Http` package.

Consumption patterns

There are several ways `IHttpClientFactory` can be used in an app:

- Basic usage
- Named clients
- Typed clients
- Generated clients

None of them are strictly superior to another. The best approach depends upon the app's constraints.

Basic usage

The `IHttpClientFactory` can be registered by calling the `AddHttpClient` extension method on the `IServiceCollection`, inside the `Startup.ConfigureServices` method.

```
services.AddHttpClient();
```

Once registered, code can accept an `IHttpClientFactory` anywhere services can be injected with dependency injection (DI). The `IHttpClientFactory` can be used to create an `HttpClient` instance:

```
public class BasicUsageModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubBranch> Branches { get; private set; }

    public bool GetBranchesError { get; private set; }

    public BasicUsageModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "https://api.github.com/repos/aspnet/AspNetCore.Docs/branches");
        request.Headers.Add("Accept", "application/vnd.github.v3+json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        var client = _clientFactory.CreateClient();

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            Branches = await response.Content
                .ReadAsAsync<IEnumerable<GitHubBranch>>();
        }
        else
        {
            GetBranchesError = true;
            Branches = Array.Empty<GitHubBranch>();
        }
    }
}
```

Using `IHttpClientFactory` in this fashion is a good way to refactor an existing app. It has no impact on the way `HttpClient` is used. In places where `HttpClient` instances are currently created, replace those occurrences with a call to CreateClient.

Named clients

If an app requires many distinct uses of `HttpClient`, each with a different configuration, an option is to use **named clients**. Configuration for a named `HttpClient` can be specified during registration in `Startup.ConfigureServices`.

```
services.AddHttpClient("github", c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    // Github API versioning
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    // Github requires a user-agent
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

In the preceding code, `AddHttpClient` is called, providing the name *github*. This client has some default configuration applied—namely the base address and two headers required to work with the GitHub API.

Each time `CreateClient` is called, a new instance of `HttpClient` is created and the configuration action is called.

To consume a named client, a string parameter can be passed to `CreateClient`. Specify the name of the client to be created:

```
public class NamedClientModel : PageModel
{
    private readonly IHttpClientFactory _clientFactory;

    public IEnumerable<GitHubPullRequest> PullRequests { get; private set; }

    public bool GetPullRequestsError { get; private set; }

    public bool HasPullRequests => PullRequests.Any();

    public NamedClientModel(IHttpClientFactory clientFactory)
    {
        _clientFactory = clientFactory;
    }

    public async Task OnGet()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "repos/aspnet/AspNetCore.Docs/pulls");

        var client = _clientFactory.CreateClient("github");

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            PullRequests = await response.Content
                .ReadAsAsync<IEnumerable<GitHubPullRequest>>();
        }
        else
        {
            GetPullRequestsError = true;
            PullRequests = Array.Empty<GitHubPullRequest>();
        }
    }
}
```

In the preceding code, the request doesn't need to specify a hostname. It can pass just the path, since the base address configured for the client is used.

Typed clients

Typed clients:

- Provide the same capabilities as named clients without the need to use strings as keys.
- Provides IntelliSense and compiler help when consuming clients.

- Provide a single location to configure and interact with a particular `HttpClient`. For example, a single typed client might be used for a single backend endpoint and encapsulate all logic dealing with that endpoint.
- Work with DI and can be injected where required in your app.

A typed client accepts an `HttpClient` parameter in its constructor:

```
public class GitHubService
{
    public HttpClient Client { get; }

    public GitHubService(HttpClient client)
    {
        client.BaseAddress = new Uri("https://api.github.com/");
        // GitHub API versioning
        client.DefaultRequestHeaders.Add("Accept",
            "application/vnd.github.v3+json");
        // GitHub requires a user-agent
        client.DefaultRequestHeaders.Add("User-Agent",
            "HttpClientFactory-Sample");

        Client = client;
    }

    public async Task<IEnumerable<GitHubIssue>> GetAspNetDocsIssues()
    {
        var response = await Client.GetAsync(
            "/repos/aspnet/AspNetCore.Docs/issues?state=open&sort=created&direction=desc");

        response.EnsureSuccessStatusCode();

        var result = await response.Content
            .ReadAsAsync<IEnumerable<GitHubIssue>>();

        return result;
    }
}
```

In the preceding code, the configuration is moved into the typed client. The `HttpClient` object is exposed as a public property. It's possible to define API-specific methods that expose `HttpClient` functionality. The `GetAspNetDocsIssues` method encapsulates the code needed to query for and parse out the latest open issues from a GitHub repository.

To register a typed client, the generic AddHttpClient extension method can be used within `Startup.ConfigureServices`, specifying the typed client class:

```
services.AddHttpClient<GitHubService>();
```

The typed client is registered as transient with DI. The typed client can be injected and consumed directly:

```
public class TypedClientModel : PageModel
{
    private readonly GitHubService _gitHubService;

    public IEnumerable<GitHubIssue> LatestIssues { get; private set; }

    public bool HasIssue => LatestIssues.Any();

    public bool GetIssuesError { get; private set; }

    public TypedClientModel(GitHubService gitHubService)
    {
        _gitHubService = gitHubService;
    }

    public async Task OnGet()
    {
        try
        {
            LatestIssues = await _gitHubService.GetAspNetDocsIssues();
        }
        catch(HttpRequestException)
        {
            GetIssuesError = true;
            LatestIssues = Array.Empty<GitHubIssue>();
        }
    }
}
```

If preferred, the configuration for a typed client can be specified during registration in `Startup.ConfigureServices`, rather than in the typed client's constructor:

```
services.AddHttpClient<RepoService>(c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    c.DefaultRequestHeaders.Add("Accept", "application/vnd.github.v3+json");
    c.DefaultRequestHeaders.Add("User-Agent", "HttpClientFactory-Sample");
});
```

It's possible to entirely encapsulate the `HttpClient` within a typed client. Rather than exposing it as a property, public methods can be provided which call the `HttpClient` instance internally.

```
public class RepoService
{
    // _httpClient isn't exposed publicly
    private readonly HttpClient _httpClient;

    public RepoService(HttpClient client)
    {
        _httpClient = client;
    }

    public async Task<IEnumerable<string>> GetRepos()
    {
        var response = await _httpClient.GetAsync("aspnet/repos");

        response.EnsureSuccessStatusCode();

        var result = await response.Content
            .ReadAsAsync<IEnumerable<string>>();

        return result;
    }
}
```

In the preceding code, the `HttpClient` is stored as a private field. All access to make external calls goes through the `GetRepos` method.

Generated clients

`IHttpClientFactory` can be used in combination with other third-party libraries such as Refit. Refit is a REST library for .NET. It converts REST APIs into live interfaces. An implementation of the interface is generated dynamically by the `RestService`, using `HttpClient` to make the external HTTP calls.

An interface and a reply are defined to represent the external API and its response:

```
public interface IHelloClient
{
    [Get("/helloworld")]
    Task<Reply> GetMessageAsync();
}

public class Reply
{
    public string Message { get; set; }
}
```

A typed client can be added, using Refit to generate the implementation:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("hello", c =>
    {
        c.BaseAddress = new Uri("http://localhost:5000");
    })
    .AddTypedClient(c => Refit.RestService.For<IHelloClient>(c));

    services.AddMvc();
}
```

The defined interface can be consumed where necessary, with the implementation provided by DI and Refit:

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHelloClient _client;

    public ValuesController(IHelloClient client)
    {
        _client = client;
    }

    [HttpGet("/")]
    public async Task<ActionResult<Reply>> Index()
    {
        return await _client.GetMessageAsync();
    }
}
```

Outgoing request middleware

`HttpClient` already has the concept of delegating handlers that can be linked together for outgoing HTTP requests. The `IHttpClientFactory` makes it easy to define the handlers to apply for each named client. It supports registration and chaining of multiple handlers to build an outgoing request middleware pipeline. Each of these handlers is able to perform work before and after the outgoing request. This pattern is similar to the inbound middleware pipeline in ASP.NET Core. The pattern provides a mechanism to manage cross-cutting concerns around HTTP requests, including caching, error handling, serialization, and logging.

To create a handler, define a class deriving from DelegatingHandler. Override the `SendAsync` method to execute code before passing the request to the next handler in the pipeline:

```
public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest)
            {
                Content = new StringContent(
                    "You must supply an API key header called X-API-KEY")
            };
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

The preceding code defines a basic handler. It checks to see if an `X-API-KEY` header has been included on the request. If the header is missing, it can avoid the HTTP call and return a suitable response.

During registration, one or more handlers can be added to the configuration for an `HttpClient`. This task is accomplished via extension methods on the IHttpClientBuilder.

```
services.AddTransient<ValidateHeaderHandler>();

services.AddHttpClient("externalservice", c =>
{
    // Assume this is an "external" service which requires an API KEY
    c.BaseAddress = new Uri("https://localhost:5000/");
})
.AddHttpMessageHandler<ValidateHeaderHandler>();
```

In the preceding code, the `ValidateHeaderHandler` is registered with DI. The handler **must** be registered in DI as a transient service, never scoped. If the handler is registered as a scoped service and any services that the handler depends upon are disposable:

- The handler's services could be disposed before the handler goes out of scope.
- The disposed handler services causes the handler to fail.

Once registered, AddHttpMessageHandler can be called, passing in the handler type.

Multiple handlers can be registered in the order that they should execute. Each handler wraps the next handler until the final `HttpClientHandler` executes the request:

```
services.AddTransient<SecureRequestHandler>();
services.AddTransient<RequestDataHandler>();

services.AddHttpClient("clientwithhandlers")
    // This handler is on the outside and called first during the
    // request, last during the response.
    .AddHttpMessageHandler<SecureRequestHandler>()
    // This handler is on the inside, closest to the request being
    // sent.
    .AddHttpMessageHandler<RequestDataHandler>();
```

Use one of the following approaches to share per-request state with message handlers:

- Pass data into the handler using `HttpRequestMessage.Properties`.
- Use `IHttpContextAccessor` to access the current request.
- Create a custom `AsyncLocal` storage object to pass the data.

Use Polly-based handlers

`IHttpClientFactory` integrates with a popular third-party library called Polly. Polly is a comprehensive resilience and transient fault-handling library for .NET. It allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Extension methods are provided to enable the use of Polly policies with configured `HttpClient` instances. The Polly extensions:

- Support adding Polly-based handlers to clients.
- Can be used after installing the Microsoft.Extensions.Http.Polly NuGet package. The package isn't included in the ASP.NET Core shared framework.

Handle transient faults

Most common faults occur when external HTTP calls are transient. A convenient extension method called `AddTransientHttpErrorPolicy` is included which allows a policy to be defined to handle transient errors. Policies configured with this extension method handle `HttpRequestException`, HTTP 5xx responses, and HTTP 408 responses.

The `AddTransientHttpErrorPolicy` extension can be used within `Startup.ConfigureServices`. The extension provides access to a `PolicyBuilder` object configured to handle errors representing a possible transient fault:

```
services.AddHttpClient<UnreliableEndpointCallerService>()
    .AddTransientHttpErrorPolicy(p =>
        p.WaitAndRetryAsync(3, _ => TimeSpan.FromMilliseconds(600)));
```

In the preceding code, a `WaitAndRetryAsync` policy is defined. Failed requests are retried up to three times with a delay of 600 ms between attempts.

Dynamically select policies

Additional extension methods exist which can be used to add Polly-based handlers. One such extension is `AddPolicyHandler`, which has multiple overloads. One overload allows the request to be inspected when defining which policy to apply:

```
var timeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeout = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

services.AddHttpClient("conditionalpolicy")
// Run some code to select a policy based on the request
    .AddPolicyHandler(request =>
        request.Method == HttpMethod.Get ? timeout : longTimeout);
```

In the preceding code, if the outgoing request is an HTTP GET, a 10-second timeout is applied. For any other HTTP method, a 30-second timeout is used.

Add multiple Polly handlers

It's common to nest Polly policies to provide enhanced functionality:

```
services.AddHttpClient("multiplepolicies")
    .AddTransientHttpErrorPolicy(p => p.RetryAsync(3))
    .AddTransientHttpErrorPolicy(
        p => p.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```

In the preceding example, two handlers are added. The first uses the `AddTransientHttpErrorPolicy` extension to add a retry policy. Failed requests are retried up to three times. The second call to `AddTransientHttpErrorPolicy` adds a circuit breaker policy. Further external requests are blocked for 30 seconds if five failed attempts occur sequentially. Circuit breaker policies are stateful. All calls through this client share the same circuit state.

Add policies from the Polly registry

An approach to managing regularly used policies is to define them once and register them with a `PolicyRegistry`. An extension method is provided which allows a handler to be added using a policy from the registry:

```
var registry = services.AddPolicyRegistry();

registry.Add("regular", timeout);
registry.Add("long", longTimeout);

services.AddHttpClient("regulartimeouthandler")
    .AddPolicyHandlerFromRegistry("regular");
```

In the preceding code, two policies are registered when the `PolicyRegistry` is added to the `ServiceCollection`. To use a policy from the registry, the `AddPolicyHandlerFromRegistry` method is used, passing the name of the policy

to apply.

Further information about `IHttpClientFactory` and Polly integrations can be found on the Polly wiki.

HttpClient and lifetime management

A new `HttpClient` instance is returned each time `CreateClient` is called on the `IHttpClientFactory`. There's an HttpMessageHandler per named client. The factory manages the lifetimes of the `HttpMessageHandler` instances.

`IHttpClientFactory` pools the `HttpMessageHandler` instances created by the factory to reduce resource consumption. An `HttpMessageHandler` instance may be reused from the pool when creating a new `HttpClient` instance if its lifetime hasn't expired.

Pooling of handlers is desirable as each handler typically manages its own underlying HTTP connections. Creating more handlers than necessary can result in connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS changes.

The default handler lifetime is two minutes. The default value can be overridden on a per named client basis. To override it, call SetHandlerLifetime on the `IHttpClientBuilder` that is returned when creating the client:

```
services.AddHttpClient("extendedhandlerlifetime")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

Disposal of the client isn't required. Disposal cancels outgoing requests and guarantees the given `HttpClient` instance can't be used after calling Dispose. `IHttpClientFactory` tracks and disposes resources used by `HttpClient` instances. The `HttpClient` instances can generally be treated as .NET objects not requiring disposal.

Keeping a single `HttpClient` instance alive for a long duration is a common pattern used before the inception of `IHttpClientFactory`. This pattern becomes unnecessary after migrating to `IHttpClientFactory`.

Alternatives to IHttpClientFactory

Using `IHttpClientFactory` in a DI-enabled app avoids:

- Resource exhaustion problems by pooling `HttpMessageHandler` instances.
- Stale DNS problems by cycling `HttpMessageHandler` instances at regular intervals.

There are alternative ways to solve the preceding problems using a long-lived SocketsHttpHandler instance.

- Create an instance of `SocketsHttpHandler` when the app starts and use it for the life of the app.
- Configure PooledConnectionLifetime to an appropriate value based on DNS refresh times.
- Create `HttpClient` instances using `new HttpClient(handler, disposeHandler: false)` as needed.

The preceding approaches solve the resource management problems that `IHttpClientFactory` solves in a similar way.

- The `SocketsHttpHandler` shares connections across `HttpClient` instances. This sharing prevents socket exhaustion.
- The `SocketsHttpHandler` cycles connections according to `PooledConnectionLifetime` to avoid stale DNS problems.

Cookies

The pooled `HttpMessageHandler` instances results in `CookieContainer` objects being shared. Unanticipated `CookieContainer` object sharing often results in incorrect code. For apps that require cookies, consider either:

- Disabling automatic cookie handling
- Avoiding `IHttpClientFactory`

Call ConfigurePrimaryHttpMessageHandler to disable automatic cookie handling:

```
services.AddHttpClient("configured-disable-automatic-cookies")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new SocketsHttpHandler()
        {
            UseCookies = false,
        };
    });
```

Logging

Clients created via `IHttpClientFactory` record log messages for all requests. Enable the appropriate information level in your logging configuration to see the default log messages. Additional logging, such as the logging of request headers, is only included at trace level.

The log category used for each client includes the name of the client. A client named *MyNamedClient*, for example, logs messages with a category of `System.Net.Http.HttpClient.MyNamedClient.LogicalHandler`. Messages suffixed with *LogicalHandler* occur outside the request handler pipeline. On the request, messages are logged before any other handlers in the pipeline have processed it. On the response, messages are logged after any other pipeline handlers have received the response.

Logging also occurs inside the request handler pipeline. In the *MyNamedClient* example, those messages are logged against the log category `System.Net.Http.HttpClient.MyNamedClient.ClientHandler`. For the request, this occurs after all other handlers have run and immediately before the request is sent out on the network. On the response, this logging includes the state of the response before it passes back through the handler pipeline.

Enabling logging outside and inside the pipeline enables inspection of the changes made by the other pipeline handlers. This may include changes to request headers, for example, or to the response status code.

Including the name of the client in the log category enables log filtering for specific named clients where necessary.

Configure the HttpMessageHandler

It may be necessary to control the configuration of the inner `HttpMessageHandler` used by a client.

An `IHttpClientBuilder` is returned when adding named or typed clients. The ConfigurePrimaryHttpMessageHandler extension method can be used to define a delegate. The delegate is used to create and configure the primary `HttpMessageHandler` used by that client:

```
services.AddHttpClient("configured-inner-handler")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new HttpClientHandler()
        {
            AllowAutoRedirect = false,
            UseDefaultCredentials = true
        };
    });
```

Use IHttpClientFactory in a console app

In a console app, add the following package references to the project:

- Microsoft.Extensions.Hosting

- Microsoft.Extensions.Http

In the following example:

- IHttpClientFactory is registered in the Generic Host's service container.
- `MyService` creates a client factory instance from the service, which is used to create an `HttpClient`. `HttpClient` is used to retrieve a webpage.
- `Main` creates a scope to execute the service's `GetPage` method and write the first 500 characters of the webpage content to the console.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
class Program
{
    static async Task<int> Main(string[] args)
    {
        var builder = new HostBuilder()
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHttpClient();
                services.AddTransient<IMyService, MyService>();
            }).UseConsoleLifetime();

        var host = builder.Build();

        using (var serviceScope = host.Services.CreateScope())
        {
            var services = serviceScope.ServiceProvider;

            try
            {
                var myService = services.GetRequiredService<IMyService>();
                var pageContent = await myService.GetPage();

                Console.WriteLine(pageContent.Substring(0, 500));
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();

                logger.LogError(ex, "An error occurred.");
            }
        }

        return 0;
    }

    public interface IMyService
    {
        Task<string> GetPage();
    }

    public class MyService : IMyService
    {
        private readonly IHttpClientFactory _clientFactory;

        public MyService(IHttpClientFactory clientFactory)
        {
            _clientFactory = clientFactory;
        }

        public async Task<string> GetPage()
```

```
        {
            // Content from BBC One: Dr. Who website (©BBC)
            var request = new HttpRequestMessage(HttpMethod.Get,
                "https://www.bbc.co.uk/programmes/b006q2x0");
            var client = _clientFactory.CreateClient();
            var response = await client.SendAsync(request);

            if (response.IsSuccessStatusCode)
            {
                return await response.Content.ReadAsStringAsync();
            }
            else
            {
                return $"StatusCode: {response.StatusCode}";
            }
        }
    }
}
```

Header propagation middleware

Header propagation is a community supported middleware to propagate HTTP headers from the incoming request to the outgoing HTTP Client requests. To use header propagation:

- Reference the community supported port of the package HeaderPropagation. ASP.NET Core 3.1 and later supports Microsoft.AspNetCore.HeaderPropagation.
- Configure the middleware and `HttpClient` in `Startup`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddHttpClient("MyForwardingClient").AddHeaderPropagation();
    services.AddHeaderPropagation(options =>
    {
        options.Headers.Add("X-TraceId");
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseHeaderPropagation();

    app.UseMvc();
}
```

- The client includes the configured headers on outbound requests:

```
var client = clientFactory.CreateClient("MyForwardingClient");
var response = client.GetAsync(...);
```

Additional resources

- Use HttpClientFactory to implement resilient HTTP requests
- Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies
- Implement the Circuit Breaker pattern

Static files in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson and Kirk Larkin

Static files, such as HTML, CSS, images, and JavaScript, are assets an ASP.NET Core app serves directly to clients by default.

View or download sample code (how to download)

Serve static files

Static files are stored within the project's web root directory. The default directory is `{content root}/wwwroot`, but it can be changed with the UseWebRoot method. For more information, see Content root and Web root.

The CreateDefaultBuilder method sets the content root to the current directory:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The preceding code was created with the web app template.

Static files are accessible via a path relative to the web root. For example, the **Web Application** project templates contain several folders within the `wwwroot` folder:

- `wwwroot`
  - `css`
  - `js`
  - `lib`

Consider creating the *wwwroot/images* folder and adding the *wwwroot/images/MyImage.jpg* file. The URI format to access a file in the `images` folder is `https://<hostname>/images/<image_file_name>`. For example, `https://localhost:5001/images/MyImage.jpg`

Serve files in web root

The default web app templates call the UseStaticFiles method in `Startup.Configure`, which enables static files to be served:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

The parameterless `UseStaticFiles` method overload marks the files in web root as servable. The following markup references *wwwroot/images/MyImage.jpg*:

```
<img src="~/images/MyImage.jpg" class="img" alt="My image" />
```

In the preceding code, the tilde character `~/` points to the web root.

Serve files outside of web root

Consider a directory hierarchy in which the static files to be served reside outside of the web root:

- `wwwroot`
  - `css`
  - `images`
  - `js`
- `MyStaticFiles`
  - `images`
    - `red-rose.jpg`

A request can access the `red-rose.jpg` file by configuring the Static File Middleware as follows:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    // using Microsoft.Extensions.FileProviders;
    // using System.IO;
    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(env.ContentRootPath, "MyStaticFiles")),
        RequestPath = "/StaticFiles"
    });

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

In the preceding code, the *MyStaticFiles* directory hierarchy is exposed publicly via the *StaticFiles* URI segment. A request to `https://<hostname>/StaticFiles/images/red-rose.jpg` serves the *red-rose.jpg* file.

The following markup references *MyStaticFiles/images/red-rose.jpg*:

```
<img src="~/StaticFiles/images/red-rose.jpg" class="img" alt="A red rose" />
```

Set HTTP response headers

A StaticFileOptions object can be used to set HTTP response headers. In addition to configuring static file serving from the web root, the following code sets the `Cache-Control` header:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    const string cacheMaxAge = "604800";
    app.UseStaticFiles(new StaticFileOptions
    {
        OnPrepareResponse = ctx =>
        {
            // using Microsoft.AspNetCore.Http;
            ctx.Context.Response.Headers.Append(
                 "Cache-Control", $"public, max-age={cacheMaxAge}");
        }
    });

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

Static files are publicly cacheable for 600 seconds:

Static file authorization

The Static File Middleware doesn't provide authorization checks. Any files served by it, including those under `wwwroot`, are publicly accessible. To serve files based on authorization:

- Store them outside of `wwwroot` and any directory accessible to the default Static File Middleware.
- Call `UseStaticFiles` after `UseAuthorization` and specify the path:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    // wwwroot css, JavaScrip, and images don't require authentication.
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
                    Path.Combine(env.ContentRootPath, "MyStaticFiles")),
        RequestPath = "/StaticFiles"
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

The preceding approach requires users to be authenticated:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount
= true)
            .AddEntityFrameworkStores<ApplicationDbContext>();

        services.AddRazorPages();

        services.AddAuthorization(options =>
        {
            options.FallbackPolicy = new AuthorizationPolicyBuilder()
                .RequireAuthenticatedUser()
                .Build();
        });
    }

    // Remaining code ommitted for brevity.
```

For information on how to globally require all users to be authenticated, see Require authenticated users.

An alternative approach to serve files based on authorization:

- Store them outside of `wwwroot` and any directory accessible to the Static File Middleware.
- Serve them via an action method to which authorization is applied and return a FileResult object:

```
[Authorize]
public IActionResult BannerImage()
{
    var filePath = Path.Combine(
        _env.ContentRootPath, "MyStaticFiles", "images", "red-rose.jpg");

    return PhysicalFile(filePath, "image/jpeg");
}
```

Directory browsing

Directory browsing allows directory listing within specified directories.

Directory browsing is disabled by default for security reasons. For more information, see Considerations.

Enable directory browsing with:

- AddDirectoryBrowser in `Startup.ConfigureServices`.
- UseDirectoryBrowser in `Startup.Configure`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddDirectoryBrowser();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    // using Microsoft.Extensions.FileProviders;
    // using System.IO;
    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(env.WebRootPath, "images")),
        RequestPath = "/MyImages"
    });

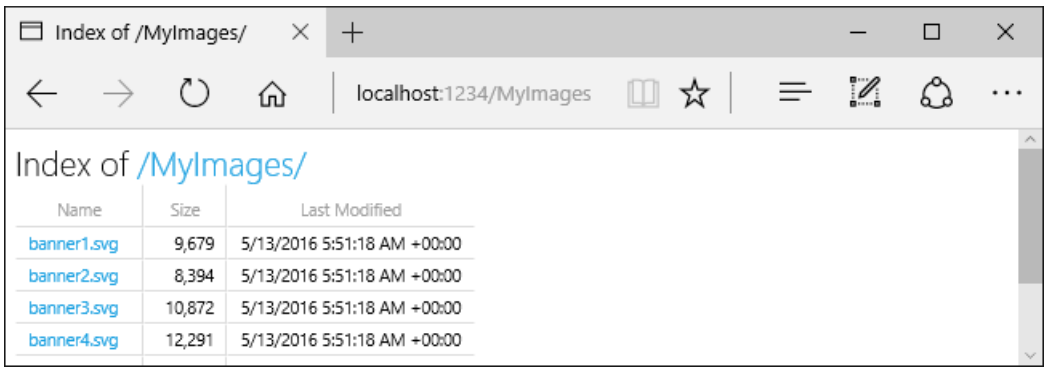
    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(env.WebRootPath, "images")),
        RequestPath = "/MyImages"
    });

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

The preceding code allows directory browsing of the *wwwroot/images* folder using the URL `https://<hostname>/MyImages`, with links to each file and folder:



Serve default documents

Setting a default page provides visitors a starting point on a site. To serve a default page from `wwwroot`

without a fully qualified URI, call the UseDefaultFiles method:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseDefaultFiles();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

`UseDefaultFiles` must be called before `UseStaticFiles` to serve the default file. `UseDefaultFiles` is a URL rewriter that doesn't serve the file.

With `UseDefaultFiles`, requests to a folder in `wwwroot` search for:

- *default.htm*
- *default.html*
- *index.htm*
- *index.html*

The first file found from the list is served as though the request were the fully qualified URI. The browser URL continues to reflect the URI requested.

The following code changes the default file name to *mydefault.html*:

```
var options = new DefaultFilesOptions();
options.DefaultFileNames.Clear();
options.DefaultFileNames.Add("mydefault.html");
app.UseDefaultFiles(options);
app.UseStaticFiles();
```

The following code shows `Startup.Configure` with the preceding code:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    var options = new DefaultFilesOptions();
    options.DefaultFileNames.Clear();
    options.DefaultFileNames.Add("mydefault.html");
    app.UseDefaultFiles(options);
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

UseFileServer for default documents

UseFileServer combines the functionality of `UseStaticFiles`, `UseDefaultFiles`, and optionally `UseDirectoryBrowser`.

Call `app.UseFileServer` to enable the serving of static files and the default file. Directory browsing isn't enabled. The following code shows `Startup.Configure` with `UseFileServer`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseFileServer();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

The following code enables the serving of static files, the default file, and directory browsing:

```
app.UseFileServer(enableDirectoryBrowsing: true);
```

The following code shows `Startup.Configure` with the preceding code:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseFileServer(enableDirectoryBrowsing: true);

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

Consider the following directory hierarchy:

- `wwwroot`
  - `css`
  - `images`
  - `js`
- `MyStaticFiles`
  - `images`
    - `MyImage.jpg`
  - `default.html`

The following code enables the serving of static files, the default file, and directory browsing of `MyStaticFiles`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddDirectoryBrowser();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseStaticFiles(); // For the wwwroot folder.

    // using Microsoft.Extensions.FileProviders;
    // using System.IO;
    app.UseFileServer(new FileServerOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(env.ContentRootPath, "MyStaticFiles")),
        RequestPath = "/StaticFiles",
        EnableDirectoryBrowsing = true
    });

    app.UseRouting();

    app.UseAuthorization();

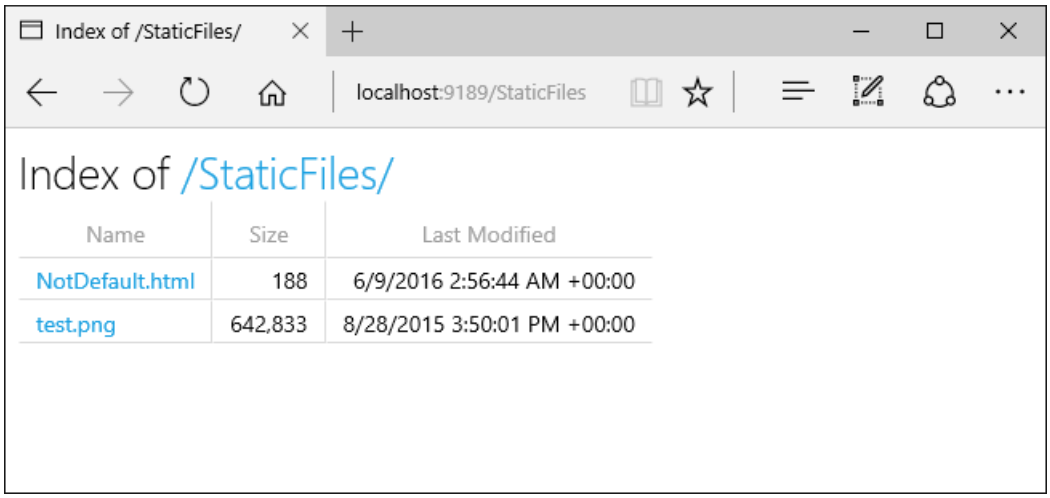
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

AddDirectoryBrowser must be called when the `EnableDirectoryBrowsing` property value is `true`.

Using the file hierarchy and preceding code, URLs resolve as follows:

URI	RESPONSE
`https://<hostname>/StaticFiles/images/MyImage.jpg`	*MyStaticFiles/images/MyImage.jpg*
`https://<hostname>/StaticFiles`	*MyStaticFiles/default.html*

If no default-named file exists in the *MyStaticFiles* directory, `https://<hostname>/StaticFiles` returns the directory listing with clickable links:



UseDefaultFiles and UseDirectoryBrowser perform a client-side redirect from the target URI without a trailing `/` to the target URI with a trailing `/`. For example, from `https://<hostname>/StaticFiles` to `https://<hostname>/StaticFiles/`. Relative URLs within the *StaticFiles* directory are invalid without a trailing slash (`/`).

FileExtensionContentTypeProvider

The FileExtensionContentTypeProvider class contains a `Mappings` property that serves as a mapping of file extensions to MIME content types. In the following sample, several file extensions are mapped to known MIME types. The *.rtf* extension is replaced, and *.mp4* is removed:

```
// using Microsoft.AspNetCore.StaticFiles;
// using Microsoft.Extensions.FileProviders;
// using System.IO;

// Set up custom content types - associating file extension to MIME type
var provider = new FileExtensionContentTypeProvider();
// Add new mappings
provider.Mappings[".myapp"] = "application/x-msdownload";
provider.Mappings[".htm3"] = "text/html";
provider.Mappings[".image"] = "image/png";
// Replace an existing mapping
provider.Mappings[".rtf"] = "application/x-msdownload";
// Remove MP4 videos.
provider.Mappings.Remove(".mp4");

app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(env.WebRootPath, "images")),
    RequestPath = "/MyImages",
    ContentTypeProvider = provider
});

app.UseDirectoryBrowser(new DirectoryBrowserOptions
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(env.WebRootPath, "images")),
    RequestPath = "/MyImages"
});
```

The following code shows `Startup.Configure` with the preceding code:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    // using Microsoft.AspNetCore.StaticFiles;
    // using Microsoft.Extensions.FileProviders;
    // using System.IO;

    // Set up custom content types - associating file extension to MIME type
    var provider = new FileExtensionContentTypeProvider();
    // Add new mappings
    provider.Mappings[".myapp"] = "application/x-msdownload";
    provider.Mappings[".htm3"] = "text/html";
    provider.Mappings[".image"] = "image/png";
    // Replace an existing mapping
    provider.Mappings[".rtf"] = "application/x-msdownload";
    // Remove MP4 videos.
    provider.Mappings.Remove(".mp4");

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(env.WebRootPath, "images")),
        RequestPath = "/MyImages",
        ContentTypeProvider = provider
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(env.WebRootPath, "images")),
        RequestPath = "/MyImages"
    });

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

See MIME content types.

Non-standard content types

The Static File Middleware understands almost 400 known file content types. If the user requests a file with an unknown file type, the Static File Middleware passes the request to the next middleware in the pipeline. If no middleware handles the request, a *404 Not Found* response is returned. If directory browsing is enabled, a link to the file is displayed in a directory listing.

The following code enables serving unknown types and renders the unknown file as an image:

```
app.UseStaticFiles(new StaticFileOptions
{
    ServeUnknownFileTypes = true,
    DefaultContentType = "image/png"
});
```

The following code shows `Startup.Configure` with the preceding code:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseStaticFiles(new StaticFileOptions
    {
        ServeUnknownFileTypes = true,
        DefaultContentType = "image/png"
    });

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}
```

With the preceding code, a request for a file with an unknown content type is returned as an image.

> **WARNING**
>
> Enabling ServeUnknownFileTypes is a security risk. It's disabled by default, and its use is discouraged. FileExtensionContentTypeProvider provides a safer alternative to serving files with non-standard extensions.

Serve files from multiple locations

`UseStaticFiles` and `UseFileServer` default to the file provider pointing at `wwwroot`. Additional instances of `UseStaticFiles` and `UseFileServer` can be provided with other file providers to serve files from other locations. For more information, see this GitHub issue.

Security considerations for static files

> **WARNING**
>
> `UseDirectoryBrowser` and `UseStaticFiles` can leak secrets. Disabling directory browsing in production is highly recommended. Carefully review which directories are enabled via `UseStaticFiles` or `UseDirectoryBrowser`. The entire directory and its sub-directories become publicly accessible. Store files suitable for serving to the public in a dedicated directory, such as `<content_root>/wwwroot`. Separate these files from MVC views, Razor Pages, configuration files, etc.

- The URLs for content exposed with `UseDirectoryBrowser` and `UseStaticFiles` are subject to the case sensitivity and character restrictions of the underlying file system. For example, Windows is case insensitive, but macOS and Linux aren't.
- ASP.NET Core apps hosted in IIS use the ASP.NET Core Module to forward all requests to the app, including static file requests. The IIS static file handler isn't used and has no chance to handle requests.
- Complete the following steps in IIS Manager to remove the IIS static file handler at the server or website level:
  1. Navigate to the **Modules** feature.
  2. Select **StaticFileModule** in the list.
  3. Click **Remove** in the **Actions** sidebar.

> **WARNING**
>
> If the IIS static file handler is enabled **and** the ASP.NET Core Module is configured incorrectly, static files are served. This happens, for example, if the *web.config* file isn't deployed.

- Place code files, including *.cs* and *.cshtml*, outside of the app project's web root. A logical separation is therefore created between the app's client-side content and server-based code. This prevents server-side code from being leaked.

Additional resources

- Middleware
- Introduction to ASP.NET Core

By Rick Anderson and Scott Addie

Static files, such as HTML, CSS, images, and JavaScript, are assets an ASP.NET Core app serves directly to clients. Some configuration is required to enable serving of these files.

View or download sample code (how to download)

Serve static files

Static files are stored within the project's web root directory. The default directory is *{content root}/wwwroot*, but it can be changed via the UseWebRoot method. See Content root and Web root for more information.

The app's web host must be made aware of the content root directory.

The `WebHost.CreateDefaultBuilder` method sets the content root to the current directory:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Static files are accessible via a path relative to the web root. For example, the **Web Application** project template contains several folders within the `wwwroot` folder:

- `wwwroot`
  - `css`
  - `images`
  - `js`

The URI format to access a file in the *images* subfolder is *http://<server_address>/images/<image_file_name>*. For example, *http://localhost:9189/images/banner3.svg*.

If targeting .NET Framework, add the Microsoft.AspNetCore.StaticFiles package to the project. If targeting .NET Core, the Microsoft.AspNetCore.App metapackage includes this package.

Configure the middleware, which enables the serving of static files.

Serve files inside of web root

Invoke the UseStaticFiles method within `Startup.Configure`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

The parameterless `UseStaticFiles` method overload marks the files in web root as servable. The following markup references *wwwroot/images/banner1.svg*:

```
<img src="~/images/banner1.svg" alt="ASP.NET" class="img-responsive" />
```

In the preceding code, the tilde character `~/` points to the web root.

Serve files outside of web root

Consider a directory hierarchy in which the static files to be served reside outside of the web root:

- `wwwroot`
  - `css`
  - `images`
  - `js`
- `MyStaticFiles`
  - `images`
    - `banner1.svg`

A request can access the *banner1.svg* file by configuring the Static File Middleware as follows:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),
        RequestPath = "/StaticFiles"
    });
}
```

In the preceding code, the *MyStaticFiles* directory hierarchy is exposed publicly via the *StaticFiles* URI segment. A request to *http://<server_address>/StaticFiles/images/banner1.svg* serves the *banner1.svg* file.

The following markup references *MyStaticFiles/images/banner1.svg*:

```
<img src="~/StaticFiles/images/banner1.svg" alt="ASP.NET" class="img-responsive" />
```

Set HTTP response headers

A StaticFileOptions object can be used to set HTTP response headers. In addition to configuring static file serving from the web root, the following code sets the `Cache-Control` header:

```
public void Configure(IApplicationBuilder app)
{
    var cachePeriod =  "604800";
    app.UseStaticFiles(new StaticFileOptions
    {
        OnPrepareResponse = ctx =>
        {
            // Requires the following import:
            // using Microsoft.AspNetCore.Http;
            ctx.Context.Response.Headers.Append("Cache-Control", $"public, max-age={cachePeriod}");
        }
    });
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The HeaderDictionaryExtensions.Append method exists in the Microsoft.AspNetCore.Http package.

The files have been made publicly cacheable for 10 minutes (600 seconds) in the Development environment:

Static file authorization

The Static File Middleware doesn't provide authorization checks. Any files served by it, including those under *wwwroot*, are publicly accessible. To serve files based on authorization:

- Store them outside of *wwwroot* and any directory accessible to the Static File Middleware.
- Serve them via an action method to which authorization is applied. Return a FileResult object:

```
[Authorize]
public IActionResult BannerImage()
{
    var file = Path.Combine(Directory.GetCurrentDirectory(),
                            "MyStaticFiles", "images", "banner1.svg");

    return PhysicalFile(file, "image/svg+xml");
}
```

Enable directory browsing

Directory browsing allows users of your web app to see a directory listing and files within a specified directory. Directory browsing is disabled by default for security reasons (see Considerations). Enable directory browsing by invoking the UseDirectoryBrowser method in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

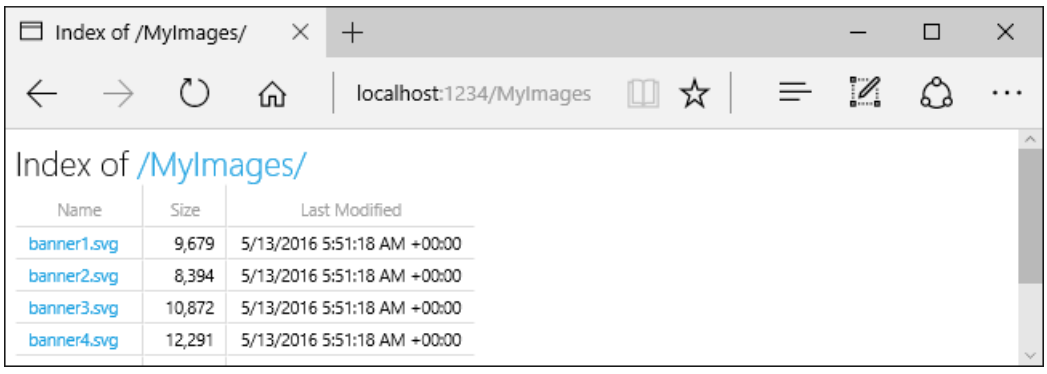
    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

Add required services by invoking the AddDirectoryBrowser method from `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

The preceding code allows directory browsing of the *wwwroot/images* folder using the URL *http://<server_address>/MyImages*, with links to each file and folder:



See Considerations on the security risks when enabling browsing.

Note the two `UseStaticFiles` calls in the following example. The first call enables the serving of static files in the *wwwroot* folder. The second call enables directory browsing of the *wwwroot/images* folder using the URL *http://<server_address>/MyImages*:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

Serve a default document

Setting a default home page provides visitors a logical starting point when visiting your site. To serve a default page without the user fully qualifying the URI, call the UseDefaultFiles method from `Startup.Configure`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```

IMPORTANT

`UseDefaultFiles` must be called before `UseStaticFiles` to serve the default file. `UseDefaultFiles` is a URL rewriter that doesn't actually serve the file. Enable Static File Middleware via `UseStaticFiles` to serve the file.

With `UseDefaultFiles`, requests to a folder search for:

- *default.htm*
- *default.html*
- *index.htm*
- *index.html*

The first file found from the list is served as though the request were the fully qualified URI. The browser URL continues to reflect the URI requested.

The following code changes the default file name to *mydefault.html*:

```
public void Configure(IApplicationBuilder app)
{
    // Serve my app-specific default file, if present.
    DefaultFilesOptions options = new DefaultFilesOptions();
    options.DefaultFileNames.Clear();
    options.DefaultFileNames.Add("mydefault.html");
    app.UseDefaultFiles(options);
    app.UseStaticFiles();
}
```

UseFileServer

UseFileServer combines the functionality of `UseStaticFiles`, `UseDefaultFiles`, and optionally `UseDirectoryBrowser`.

The following code enables the serving of static files and the default file. Directory browsing isn't enabled.

```
app.UseFileServer();
```

The following code builds upon the parameterless overload by enabling directory browsing:

```
app.UseFileServer(enableDirectoryBrowsing: true);
```

Consider the following directory hierarchy:

- **wwwroot**
  - **css**
  - **images**
  - **js**
- **MyStaticFiles**
  - **images**
    - *banner1.svg*
  - *default.html*

The following code enables static files, default files, and directory browsing of `MyStaticFiles`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseFileServer(new FileServerOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),
        RequestPath = "/StaticFiles",
        EnableDirectoryBrowsing = true
    });
}
```

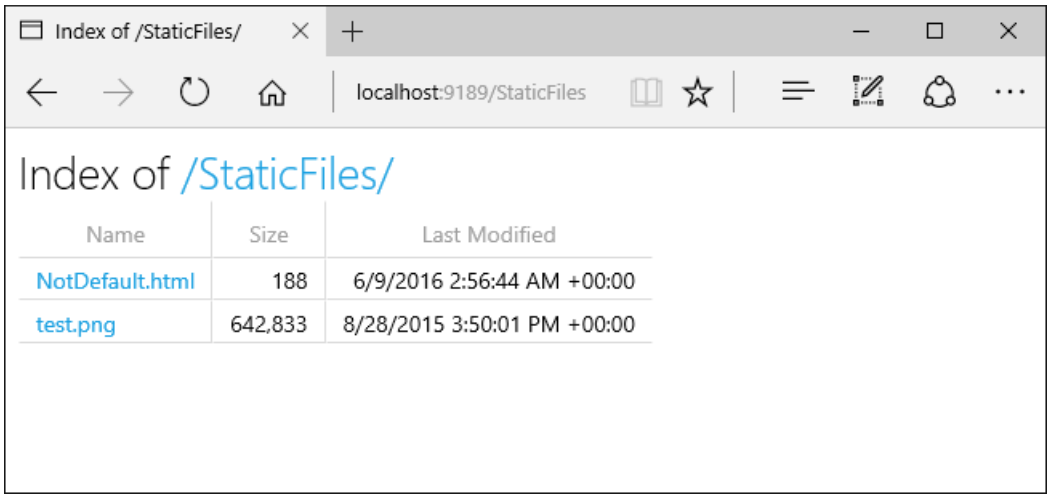
`AddDirectoryBrowser` must be called when the `EnableDirectoryBrowsing` property value is `true`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDirectoryBrowser();
}
```

Using the file hierarchy and preceding code, URLs resolve as follows:

URI	RESPONSE
http://<server_address>/StaticFiles/images/banner1.svg	MyStaticFiles/images/banner1.svg
http://<server_address>/StaticFiles	MyStaticFiles/default.html

If no default-named file exists in the *MyStaticFiles* directory, *http://<server_address>/StaticFiles* returns the directory listing with clickable links:



> **NOTE**
>
> UseDefaultFiles and UseDirectoryBrowser perform a client-side redirect from `http://{SERVER ADDRESS}/StaticFiles` (without a trailing slash) to `http://{SERVER ADDRESS}/StaticFiles/` (with a trailing slash). Relative URLs within the *StaticFiles* directory are invalid without a trailing slash.

FileExtensionContentTypeProvider

The FileExtensionContentTypeProvider class contains a `Mappings` property serving as a mapping of file extensions to MIME content types. In the following sample, several file extensions are registered to known MIME types. The *.rtf* extension is replaced, and *.mp4* is removed.

```
public void Configure(IApplicationBuilder app)
{
    // Set up custom content types - associating file extension to MIME type
    var provider = new FileExtensionContentTypeProvider();
    // Add new mappings
    provider.Mappings[".myapp"] = "application/x-msdownload";
    provider.Mappings[".htm3"] = "text/html";
    provider.Mappings[".image"] = "image/png";
    // Replace an existing mapping
    provider.Mappings[".rtf"] = "application/x-msdownload";
    // Remove MP4 videos.
    provider.Mappings.Remove(".mp4");

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages",
        ContentTypeProvider = provider
    });

    app.UseDirectoryBrowser(new DirectoryBrowserOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "images")),
        RequestPath = "/MyImages"
    });
}
```

See MIME content types.

For information on using a custom FileExtensionContentTypeProvider or to configure other StaticFileOptions in Blazor Server apps, see ASP.NET Core Blazor hosting model configuration.

Non-standard content types

Static File Middleware understands almost 400 known file content types. If the user requests a file with an unknown file type, Static File Middleware passes the request to the next middleware in the pipeline. If no middleware handles the request, a *404 Not Found* response is returned. If directory browsing is enabled, a link to the file is displayed in a directory listing.

The following code enables serving unknown types and renders the unknown file as an image:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(new StaticFileOptions
    {
        ServeUnknownFileTypes = true,
        DefaultContentType = "image/png"
    });
}
```

With the preceding code, a request for a file with an unknown content type is returned as an image.

WARNING

Enabling ServeUnknownFileTypes is a security risk. It's disabled by default, and its use is discouraged. FileExtensionContentTypeProvider provides a safer alternative to serving files with non-standard extensions.

Serve files from multiple locations

`UseStaticFiles` and `UseFileServer` defaults to the file provider pointing at *wwwroot*. You can provide additional instances of `UseStaticFiles` and `UseFileServer` with other file providers to serve files from other locations. For more information, see this GitHub issue.

Considerations

> **WARNING**
>
> `UseDirectoryBrowser` and `UseStaticFiles` can leak secrets. Disabling directory browsing in production is highly recommended. Carefully review which directories are enabled via `UseStaticFiles` or `UseDirectoryBrowser`. The entire directory and its sub-directories become publicly accessible. Store files suitable for serving to the public in a dedicated directory, such as *<content_root>/wwwroot*. Separate these files from MVC views, Razor Pages (2.x only), configuration files, etc.

- The URLs for content exposed with `UseDirectoryBrowser` and `UseStaticFiles` are subject to the case sensitivity and character restrictions of the underlying file system. For example, Windows is case insensitive—macOS and Linux aren't.
- ASP.NET Core apps hosted in IIS use the ASP.NET Core Module to forward all requests to the app, including static file requests. The IIS static file handler isn't used. It has no chance to handle requests before they're handled by the module.
- Complete the following steps in IIS Manager to remove the IIS static file handler at the server or website level:
  1. Navigate to the **Modules** feature.
  2. Select **StaticFileModule** in the list.
  3. Click **Remove** in the **Actions** sidebar.

> **WARNING**
>
> If the IIS static file handler is enabled **and** the ASP.NET Core Module is configured incorrectly, static files are served. This happens, for example, if the *web.config* file isn't deployed.

- Place code files (including *.cs* and *.cshtml*) outside of the app project's web root. A logical separation is therefore created between the app's client-side content and server-based code. This prevents server-side code from being leaked.

Additional resources

- Middleware
- Introduction to ASP.NET Core

Introduction to Razor Pages in ASP.NET Core

9/22/2020 • 43 minutes to read • Edit Online

By Rick Anderson and Ryan Nowak

Razor Pages can make coding page-focused scenarios easier and more productive than using controllers and views.

If you're looking for a tutorial that uses the Model-View-Controller approach, see Get started with ASP.NET Core MVC.

This document provides an introduction to Razor Pages. It's not a step by step tutorial. If you find some of the sections too advanced, see Get started with Razor Pages. For an overview of ASP.NET Core, see the Introduction to ASP.NET Core.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK or later

Create a Razor Pages project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

See Get started with Razor Pages for detailed instructions on how to create a Razor Pages project.

Razor Pages

Razor Pages is enabled in *Startup.cs*:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

Consider a basic page:

```
@page

<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

The preceding code looks a lot like a Razor view file used in an ASP.NET Core app with controllers and views. What makes it different is the `@page` directive. `@page` makes the file into an MVC action - which means that it handles requests directly, without going through a controller. `@page` must be the first Razor directive on a page. `@page` affects the behavior of other Razor constructs. Razor Pages file names have a *.cshtml* suffix.

A similar page, using a `PageModel` class, is shown in the following two files. The *Pages/Index2.cshtml* file:

```
@page
@using RazorPagesIntro.Pages
@model Index2Model

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>
```

The *Pages/Index2.cshtml.cs* page model:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System;

namespace RazorPagesIntro.Pages
{
    public class Index2Model : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}
```

By convention, the `PageModel` class file has the same name as the Razor Page file with *.cs* appended. For example, the previous Razor Page is *Pages/Index2.cshtml*. The file containing the `PageModel` class is named *Pages/Index2.cshtml.cs*.

The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Page path and the matching URL:

FILE NAME AND PATH	MATCHING URL
/Pages/Index.cshtml	`/` or `/Index`
/Pages/Contact.cshtml	`/Contact`
/Pages/Store/Contact.cshtml	`/Store/Contact`
/Pages/Store/Index.cshtml	`/Store` or `/Store/Index`

Notes:

- The runtime looks for Razor Pages files in the *Pages* folder by default.
- `Index` is the default page when a URL doesn't include a page.

Write a basic form

Razor Pages is designed to make common patterns used with web browsers easy to implement when building an app. Model binding, Tag Helpers, and HTML helpers all *just work* with the properties defined in a Razor Page class. Consider a page that implements a basic "contact us" form for the `Contact` model:

For the samples in this document, the `DbContext` is initialized in the Startup.cs file.

The in memory database requires the `Microsoft.EntityFrameworkCore.InMemory` NuGet package.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CustomerDbContext>(options =>
                    options.UseInMemoryDatabase("name"));
    services.AddRazorPages();
}
```

The data model:

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Models
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(10)]
        public string Name { get; set; }
    }
}
```

The db context:

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Models;

namespace RazorPagesContacts.Data
{
    public class CustomerDbContext : DbContext
    {
        public CustomerDbContext(DbContextOptions options)
            : base(options)
        {
        }

        public DbSet<Customer> Customers { get; set; }
    }
}
```

The *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer.Name" />
    <input type="submit" />
</form>
```

The *Pages/Create.cshtml.cs* page model:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;
using RazorPagesContacts.Models;
using System.Threading.Tasks;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateModel : PageModel
    {
        private readonly CustomerDbContext _context;

        public CreateModel(CustomerDbContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Customers.Add(Customer);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

By convention, the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page.

The `PageModel` class allows separation of the logic of a page from its presentation. It defines page handlers for requests sent to the page and the data used to render the page. This separation allows:

- Managing of page dependencies through dependency injection.
- Unit testing

The page has an `OnPostAsync` *handler method*, which runs on `POST` requests (when a user posts the form). Handler methods for any HTTP verb can be added. The most common handlers are:

- `OnGet` to initialize state needed for the page. In the preceding code, the `OnGet` method displays the *CreateModel.cshtml* Razor Page.
- `OnPost` to handle form submissions.

The `Async` naming suffix is optional but is often used by convention for asynchronous functions. The preceding code is typical for Razor Pages.

If you're familiar with ASP.NET apps using controllers and views:

- The `OnPostAsync` code in the preceding example looks similar to typical controller code.
- Most of the MVC primitives like model binding, validation, and action results work the same with

Controllers and Razor Pages.

The previous `OnPostAsync` method:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Customers.Add(Customer);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

The basic flow of `OnPostAsync`:

Check for validation errors.

- If there are no errors, save the data and redirect.
- If there are errors, show the page again with validation messages. In many cases, validation errors would be detected on the client, and never submitted to the server.

The *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer.Name" />
    <input type="submit" />
</form>
```

The rendered HTML from *Pages/Create.cshtml*:

```
<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input type="text" data-val="true"
           data-val-length="The field Name must be a string with a maximum length of 10."
           data-val-length-max="10" data-val-required="The Name field is required."
           id="Customer_Name" maxlength="10" name="Customer.Name" value="" />
    <input type="submit" />
    <input name="__RequestVerificationToken" type="hidden"
           value="<Antiforgery token here>" />
</form>
```

In the previous code, posting the form:

- With valid data:
  - The `OnPostAsync` handler method calls the RedirectToPage helper method. `RedirectToPage` returns an instance of RedirectToPageResult. `RedirectToPage`:

    - Is an action result.
    - Is similar to `RedirectToAction` or `RedirectToRoute` (used in controllers and views).
    - Is customized for pages. In the preceding sample, it redirects to the root Index page (`/Index`). `RedirectToPage` is detailed in the URL generation for Pages section.
- With validation errors that are passed to the server:
    - The `OnPostAsync` handler method calls the Page helper method. `Page` returns an instance of PageResult. Returning `Page` is similar to how actions in controllers return `View` . `PageResult` is the default return type for a handler method. A handler method that returns `void` renders the page.
    - In the preceding example, posting the form with no value results in ModelState.IsValid returning false. In this sample, no validation errors are displayed on the client. Validation error handing is covered later in this document.

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Customers.Add(Customer);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

- With validation errors detected by client side validation:
    - Data is **not** posted to the server.
    - Client-side validation is explained later in this document.

The `Customer` property uses `[BindProperty]` attribute to opt in to model binding:

```
public class CreateModel : PageModel
{
    private readonly CustomerDbContext _context;

    public CreateModel(CustomerDbContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customers.Add(Customer);
        await _context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

`[BindProperty]` should **not** be used on models containing properties that should not be changed by the client. For more information, see Overposting.

Razor Pages, by default, bind properties only with non-`GET` verbs. Binding to properties removes the need to writing code to convert HTTP data to the model type. Binding reduces code by using the same property to render form fields (`<input asp-for="Customer.Name">`) and accept the input.

WARNING

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the `[BindProperty]` attribute's `SupportsGet` property to `true` :

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Reviewing the *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer.Name" />
    <input type="submit" />
</form>
```

- In the preceding code, the input tag helper `<input asp-for="Customer.Name" />` binds the HTML `<input>` element to the `Customer.Name` model expression.
- `@addTagHelper` makes Tag Helpers available.

The home page

Index.cshtml is the home page:

```
@page
@model RazorPagesContacts.Pages.Customers.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts home page</h1>
<form method="post">
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var contact in Model.Customer)
            {
                <tr>
                    <td> @contact.Id  </td>
                    <td>@contact.Name</td>
                    <td>
                        <a asp-page="./Edit" asp-route-id="@contact.Id">Edit</a> |
                        <button type="submit" asp-page-handler="delete"
                                asp-route-id="@contact.Id">delete
                        </button>
                    </td>
                </tr>
            }
        </tbody>
    </table>
    <a asp-page="Create">Create New</a>
</form>
```

The associated `PageModel` class (*Index.cshtml.cs*):

```
public class IndexModel : PageModel
{
    private readonly CustomerDbContext _context;

    public IndexModel(CustomerDbContext context)
    {
        _context = context;
    }

    public IList<Customer> Customer { get; set; }

    public async Task OnGetAsync()
    {
        Customer = await _context.Customers.ToListAsync();
    }

    public async Task<IActionResult> OnPostDeleteAsync(int id)
    {
        var contact = await _context.Customers.FindAsync(id);

        if (contact != null)
        {
            _context.Customers.Remove(contact);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage();
    }
}
```

The *Index.cshtml* file contains the following markup:

```
<a asp-page="./Edit" asp-route-id="@contact.Id">Edit</a> |
```

The `<a /a>` Anchor Tag Helper used the `asp-route-{value}` attribute to generate a link to the Edit page. The link contains route data with the contact ID. For example, `https://localhost:5001/Edit/1`. Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.

The *Index.cshtml* file contains markup to create a delete button for each customer contact:

```
<button type="submit" asp-page-handler="delete"
        asp-route-id="@contact.Id">delete
```

The rendered HTML:

```
<button type="submit" formaction="/Customers?id=1&amp;handler=delete">delete</button>
```

When the delete button is rendered in HTML, its formaction includes parameters for:

- The customer contact ID, specified by the `asp-route-id` attribute.
- The `handler`, specified by the `asp-page-handler` attribute.

When the button is selected, a form `POST` request is sent to the server. By convention, the name of the handler method is selected based on the value of the `handler` parameter according to the scheme `OnPost[handler]Async`.

Because the `handler` is `delete` in this example, the `OnPostDeleteAsync` handler method is used to process the `POST` request. If the `asp-page-handler` is set to a different value, such as `remove`, a handler

method with the name `OnPostRemoveAsync` is selected.

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _context.Customers.FindAsync(id);

    if (contact != null)
    {
        _context.Customers.Remove(contact);
        await _context.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

The `OnPostDeleteAsync` method:

- Gets the `id` from the query string.
- Queries the database for the customer contact with `FindAsync`.
- If the customer contact is found, it's removed and the database is updated.
- Calls RedirectToPage to redirect to the root Index page (`/Index`).

The Edit.cshtml file

```
@page "{id:int}"
@model RazorPagesContacts.Pages.Customers.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers


<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name"></span>
        </div>
    </div>

    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

The first line contains the `@page "{id:int}"` directive. The routing constraint `"{id:int}"` tells the page to accept requests to the page that contain `int` route data. If a request to the page doesn't contain route data that can be converted to an `int`, the runtime returns an HTTP 404 (not found) error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

The *Edit.cshtml.cs* file:

```
public class EditModel : PageModel
{
    private readonly CustomerDbContext _context;

    public EditModel(CustomerDbContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Customer = await _context.Customers.FindAsync(id);

        if (Customer == null)
        {
            return RedirectToPage("./Index");
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Customer).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            throw new Exception($"Customer {Customer.Id} not found!");
        }

        return RedirectToPage("./Index");
    }

}
```

Validation

Validation rules:

- Are declaratively specified in the model class.
- Are enforced everywhere in the app.

The System.ComponentModel.DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `[DataType]` that help with formatting and don't provide any validation.

Consider the `Customer` model:

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Models
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(10)]
        public string Name { get; set; }
    }
}
```

Using the following *Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Validation: customer name:</p>

<form method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <span asp-validation-for="Customer.Name"></span>
    Name:
    <input asp-for="Customer.Name" />
    <input type="submit" />
</form>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
```

The preceding code:

- Includes jQuery and jQuery validation scripts.
- Uses the `<div />` and `<span />` Tag Helpers to enable:
  - Client-side validation.
  - Validation error rendering.
- Generates the following HTML:

```
<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input type="text" data-val="true"
           data-val-length="The field Name must be a string with a maximum length of 10."
           data-val-length-max="10" data-val-required="The Name field is required."
           id="Customer_Name" maxlength="10" name="Customer.Name" value="" />
    <input type="submit" />
    <input name="__RequestVerificationToken" type="hidden"
           value="<Antiforgery token here>" />
</form>

<script src="/lib/jquery/dist/jquery.js"></script>
<script src="/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
```

Posting the Create form without a name value displays the error message "The Name field is required." on

the form. If JavaScript is enabled on the client, the browser displays the error without posting to the server.

The `[StringLength(10)]` attribute generates `data-val-length-max="10"` on the rendered HTML. `data-val-length-max` prevents browsers from entering more than the maximum length specified. If a tool such as Fiddler is used to edit and replay the post:

- With the name longer than 10.
- The error message "The field Name must be a string with a maximum length of 10." is returned.

Consider the following `Movie` model:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

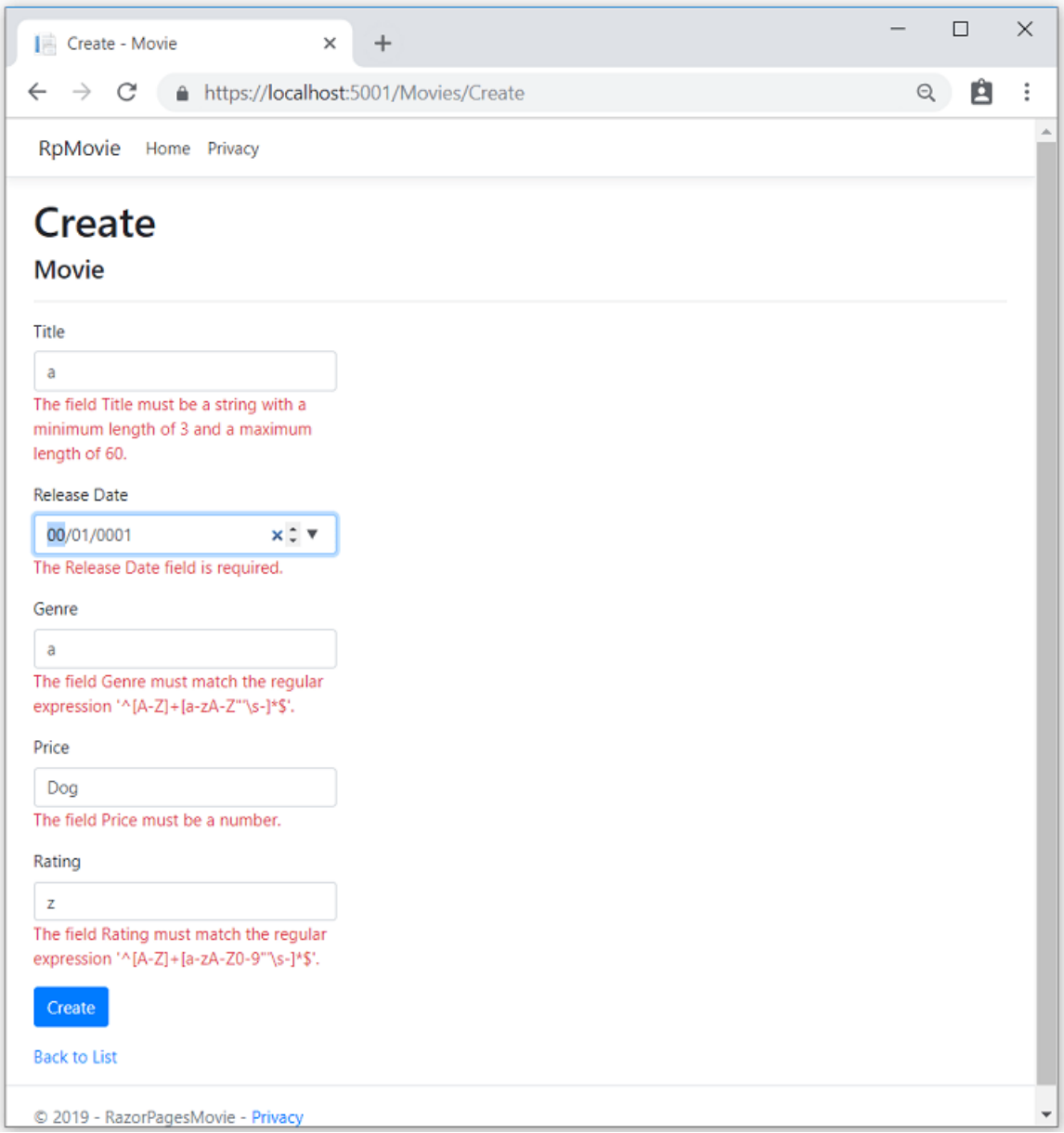
    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

The validation attributes specify behavior to enforce on the model properties they're applied to:

- The `Required` and `MinimumLength` attributes indicate that a property must have a value, but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White space, numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute sets the maximum length of a string property, and optionally its minimum length.

- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

The Create page for the `Movie` model shows displays errors with invalid values:



For more information, see:

- Add validation to the Movie app
- Model validation in ASP.NET Core.

Handle HEAD requests with an OnGet handler fallback

`HEAD` requests allow retrieving the headers for a specific resource. Unlike `GET` requests, `HEAD` requests don't return a response body.

Ordinarily, an `OnHead` handler is created and called for `HEAD` requests:

```
public void OnHead()
{
    HttpContext.Response.Headers.Add("Head Test", "Handled by OnHead!");
}
```

Razor Pages falls back to calling the `OnGet` handler if no `OnHead` handler is defined.

XSRF/CSRF and Razor Pages

Razor Pages are protected by Antiforgery validation. The FormTagHelper injects antiforgery tokens into HTML form elements.

Using Layouts, partials, templates, and Tag Helpers with Razor Pages

Pages work with all the capabilities of the Razor view engine. Layouts, partials, templates, Tag Helpers, *_ViewStart.cshtml*, and *_ViewImports.cshtml* work in the same way they do for conventional Razor views.

Let's declutter this page by taking advantage of some of those capabilities.

Add a layout page to *Pages/Shared/_Layout.cshtml*:

```
<!DOCTYPE html>
<html>
<head>
    <title>RP Sample</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <a asp-page="/Index">Home</a>
    <a asp-page="/Customers/Create">Create</a>
    <a asp-page="/Customers/Index">Customers</a> <br />

    @RenderBody()
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
</body>
</html>
```

The Layout:

- Controls the layout of each page (unless the page opts out of layout).
- Imports HTML structures such as JavaScript and stylesheets.
- The contents of the Razor page are rendered where `@RenderBody()` is called.

For more information, see layout page.

The Layout property is set in *Pages/_ViewStart.cshtml*:

```
@{
    Layout = "_Layout";
}
```

The layout is in the *Pages/Shared* folder. Pages look for other views (layouts, templates, partials) hierarchically, starting in the same folder as the current page. A layout in the *Pages/Shared* folder can be used from any Razor page under the *Pages* folder.

The layout file should go in the *Pages/Shared* folder.

We recommend you **not** put the layout file in the *Views/Shared* folder. *Views/Shared* is an MVC views pattern. Razor Pages are meant to rely on folder hierarchy, not path conventions.

View search from a Razor Page includes the *Pages* folder. The layouts, templates, and partials used with

MVC controllers and conventional Razor views *just work*.

Add a *Pages/_ViewImports.cshtml* file:

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

`@namespace` is explained later in the tutorial. The `@addTagHelper` directive brings in the built-in Tag Helpers to all the pages in the *Pages* folder.

The `@namespace` directive set on a page:

```
@page
@namespace RazorPagesIntro.Pages.Customers

@model NameSpaceModel

<h2>Name space</h2>
<p>
    @Model.Message
</p>
```

The `@namespace` directive sets the namespace for the page. The `@model` directive doesn't need to include the namespace.

When the `@namespace` directive is contained in *_ViewImports.cshtml*, the specified namespace supplies the prefix for the generated namespace in the Page that imports the `@namespace` directive. The rest of the generated namespace (the suffix portion) is the dot-separated relative path between the folder containing *_ViewImports.cshtml* and the folder containing the page.

For example, the `PageModel` class *Pages/Customers/Edit.cshtml.cs* explicitly sets the namespace:

```
namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        // Code removed for brevity.
```

The *Pages/_ViewImports.cshtml* file sets the following namespace:

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The generated namespace for the *Pages/Customers/Edit.cshtml* Razor Page is the same as the `PageModel` class.

`@namespace` *also works with conventional Razor views.*

Consider the *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Validation: customer name:</p>

<form method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <span asp-validation-for="Customer.Name"></span>
    Name:
    <input asp-for="Customer.Name" />
    <input type="submit" />
</form>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
```

The updated *Pages/Create.cshtml* view file with *_ViewImports.cshtml* and the preceding layout file:

```
@page
@model CreateModel

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer.Name" />
    <input type="submit" />
</form>
```

In the preceding code, the *_ViewImports.cshtml* imported the namespace and Tag Helpers. The layout file imported the JavaScript files.

The Razor Pages starter project contains the *Pages/_ValidationScriptsPartial.cshtml*, which hooks up client-side validation.

For more information on partial views, see Partial views in ASP.NET Core.

URL generation for Pages

The `Create` page, shown previously, uses `RedirectToPage`:

```
public class CreateModel : PageModel
{
    private readonly CustomerDbContext _context;

    public CreateModel(CustomerDbContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customers.Add(Customer);
        await _context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

The app has the following file/folder structure:

- */Pages*
  - *Index.cshtml*
  - *Privacy.cshtml*
  - */Customers*
    - *Create.cshtml*
    - *Edit.cshtml*
    - *Index.cshtml*

The *Pages/Customers/Create.cshtml* and *Pages/Customers/Edit.cshtml* pages redirect to *Pages/Customers/Index.cshtml* after success. The string `./Index` is a relative page name used to access the preceding page. It is used to generate URLs to the *Pages/Customers/Index.cshtml* page. For example:

- `Url.Page("./Index", ...)`
- `<a asp-page="./Index">Customers Index Page</a>`
- `RedirectToPage("./Index")`

The absolute page name `/Index` is used to generate URLs to the *Pages/Index.cshtml* page. For example:

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">Home Index Page</a>`
- `RedirectToPage("/Index")`

The page name is the path to the page from the root */Pages* folder including a leading `/` (for example, `/Index`). The preceding URL generation samples offer enhanced options and functional capabilities over

hard-coding a URL. URL generation uses routing and can generate and encode parameters according to how the route is defined in the destination path.

URL generation for pages supports relative names. The following table shows which Index page is selected using different `RedirectToPage` parameters in *Pages/Customers/Create.cshtml*.

REDIRECTTOPAGE(X)	PAGE
RedirectToPage("/Index")	*Pages/Index*
RedirectToPage("./Index");	*Pages/Customers/Index*
RedirectToPage("../Index")	*Pages/Index*
RedirectToPage("Index")	*Pages/Customers/Index*

`RedirectToPage("Index")`, `RedirectToPage("./Index")`, and `RedirectToPage("../Index")` are *relative names*. The `RedirectToPage` parameter is *combined* with the path of the current page to compute the name of the destination page.

Relative name linking is useful when building sites with a complex structure. When relative names are used to link between pages in a folder:

- Renaming a folder doesn't break the relative links.
- Links are not broken because they don't include the folder name.

To redirect to a page in a different Area, specify the area:

```
RedirectToPage("/Index", new { area = "Services" });
```

For more information, see Areas in ASP.NET Core and Razor Pages route and app conventions in ASP.NET Core.

ViewData attribute

Data can be passed to a page with ViewDataAttribute. Properties with the `[ViewData]` attribute have their values stored and loaded from the ViewDataDictionary.

In the following example, the `AboutModel` applies the `[ViewData]` attribute to the `Title` property:

```
public class AboutModel : PageModel
{
    [ViewData]
    public string Title { get; } = "About";

    public void OnGet()
    {
    }
}
```

In the About page, access the `Title` property as a model property:

```
<h1>@Model.Title</h1>
```

In the layout, the title is read from the ViewData dictionary:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

TempData

ASP.NET Core exposes the TempData. This property stores data until it's read. The Keep and Peek methods can be used to examine the data without deletion. `TempData` is useful for redirection, when data is needed for more than a single request.

The following code sets the value of `Message` using `TempData`:

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

The following markup in the *Pages/Customers/Index.cshtml* file displays the value of `Message` using `TempData`.

```
<h3>Msg: @Model.Message</h3>
```

The *Pages/Customers/Index.cshtml.cs* page model applies the `[TempData]` attribute to the `Message` property.

```
[TempData]
public string Message { get; set; }
```

For more information, see TempData.

Multiple handlers per page

The following page generates markup for two handlers using the `asp-page-handler` Tag Helper:

```
@page
@model CreateFATHModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

The form in the preceding example has two submit buttons, each using the `FormActionTagHelper` to submit to a different URL. The `asp-page-handler` attribute is a companion to `asp-page`. `asp-page-handler` generates URLs that submit to each of the handler methods defined by a page. `asp-page` isn't specified because the sample is linking to the current page.

The page model:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpperInvariant();
            return await OnPostJoinListAsync();
        }
    }
}
```

The preceding code uses *named handler methods*. Named handler methods are created by taking the text in the name after `On<HTTP Verb>` and before `Async` (if present). In the preceding example, the page methods are OnPost**JoinList**Async and OnPost**JoinListUC**Async. With *OnPost* and *Async* removed, the handler names are `JoinList` and `JoinListUC`.

```
<input type="submit" asp-page-handler="JoinList" value="Join" />
<input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinListUC`.

Custom routes

Use the `@page` directive to:

- Specify a custom route to a page. For example, the route to the About page can be set to `/Some/Other/Path` with `@page "/Some/Other/Path"`.

- Append segments to a page's default route. For example, an "item" segment can be added to a page's default route with `@page "item"`.
- Append parameters to a page's default route. For example, an ID parameter, `id`, can be required for a page with `@page "{id}"`.

A root-relative path designated by a tilde (`~`) at the beginning of the path is supported. For example, `@page "~/Some/Other/Path"` is the same as `@page "/Some/Other/Path"`.

If you don't like the query string `?handler=JoinList` in the URL, change the route to put the handler name in the path portion of the URL. The route can be customized by adding a route template enclosed in double quotes after the `@page` directive.

```
@page "{handler?}"
@model CreateRouteModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH/JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH/JoinListUC`.

The `?` following `handler` means the route parameter is optional.

Advanced configuration and settings

The configuration and settings in following sections is not required by most apps.

To configure advanced options, use the AddRazorPages overload that configures RazorPagesOptions:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
    {
        options.RootDirectory = "/MyPages";
        options.Conventions.AuthorizeFolder("/MyPages/Admin");
    });
}
```

Use the RazorPagesOptions to set the root directory for pages, or add application model conventions for pages. For more information on conventions, see Razor Pages authorization conventions.

To precompile views, see Razor view compilation.

Specify that Razor Pages are at the content root

By default, Razor Pages are rooted in the */Pages* directory. Add WithRazorPagesAtContentRoot to specify that your Razor Pages are at the content root (ContentRootPath) of the app:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
        {
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        })
        .WithRazorPagesAtContentRoot();
}
```

Specify that Razor Pages are at a custom root directory

Add WithRazorPagesRoot to specify that Razor Pages are at a custom root directory in the app (provide a relative path):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
        {
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        })
        .WithRazorPagesRoot("/path/to/razor/pages");
}
```

Additional resources

- See Get started with Razor Pages, which builds on this introduction.
- Authorize attribute and Razor Pages
- Download or view sample code
- Introduction to ASP.NET Core
- razor syntax reference for ASP.NET Core
- Areas in ASP.NET Core
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Razor Pages authorization conventions in ASP.NET Core
- Razor Pages route and app conventions in ASP.NET Core
- Razor Pages unit tests in ASP.NET Core
- Partial views in ASP.NET Core
- Integrate ASP.NET Core Razor components into Razor Pages and MVC apps

By Rick Anderson and Ryan Nowak

Razor Pages is a new aspect of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

If you're looking for a tutorial that uses the Model-View-Controller approach, see Get started with ASP.NET Core MVC.

This document provides an introduction to Razor Pages. It's not a step by step tutorial. If you find some of the sections too advanced, see Get started with Razor Pages. For an overview of ASP.NET Core, see the Introduction to ASP.NET Core.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
>
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

Create a Razor Pages project

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

See Get started with Razor Pages for detailed instructions on how to create a Razor Pages project.

Razor Pages

Razor Pages is enabled in *Startup.cs*:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Includes support for Razor Pages and controllers.
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
    }
}
```

Consider a basic page:

```
@page

<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

The preceding code looks a lot like a Razor view file used in an ASP.NET Core app with controllers and views. What makes it different is the `@page` directive. `@page` makes the file into an MVC action - which means that it handles requests directly, without going through a controller. `@page` must be the first Razor directive on a page. `@page` affects the behavior of other Razor constructs.

A similar page, using a `PageModel` class, is shown in the following two files. The *Pages/Index2.cshtml* file:

```
@page
@using RazorPagesIntro.Pages
@model IndexModel2

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>
```

The *Pages/Index2.cshtml.cs* page model:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using System;

namespace RazorPagesIntro.Pages
{
    public class IndexModel2 : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}
```

By convention, the `PageModel` class file has the same name as the Razor Page file with *.cs* appended. For example, the previous Razor Page is *Pages/Index2.cshtml*. The file containing the `PageModel` class is named *Pages/Index2.cshtml.cs*.

The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Page path and the matching URL:

FILE NAME AND PATH	MATCHING URL
/Pages/Index.cshtml	`/` or `/Index`
/Pages/Contact.cshtml	`/Contact`
/Pages/Store/Contact.cshtml	`/Store/Contact`
/Pages/Store/Index.cshtml	`/Store` or `/Store/Index`

Notes:

- The runtime looks for Razor Pages files in the *Pages* folder by default.
- `Index` is the default page when a URL doesn't include a page.

Write a basic form

Razor Pages is designed to make common patterns used with web browsers easy to implement when building an app. Model binding, Tag Helpers, and HTML helpers all *just work* with the properties defined in a Razor Page class. Consider a page that implements a basic "contact us" form for the `Contact` model:

For the samples in this document, the `DbContext` is initialized in the Startup.cs file.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesContacts.Data;

namespace RazorPagesContacts
{
    public class Startup
    {
        public IHostingEnvironment HostingEnvironment { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<AppDbContext>(options =>
                                options.UseInMemoryDatabase("name"));
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }
    }
}
```

The data model:

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Data
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
        public string Name { get; set; }
    }
}
```

The db context:

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions options)
            : base(options)
        {
        }

        public DbSet<Customer> Customers { get; set; }
    }
}
```

The *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

The *Pages/Create.cshtml.cs* page model:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class CreateModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }
    }
}
```

By convention, the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page.

The `PageModel` class allows separation of the logic of a page from its presentation. It defines page handlers for requests sent to the page and the data used to render the page. This separation allows:

- Managing of page dependencies through dependency injection.
- Unit testing the pages.

The page has an `OnPostAsync` *handler method*, which runs on `POST` requests (when a user posts the form). You can add handler methods for any HTTP verb. The most common handlers are:

- `OnGet` to initialize state needed for the page. OnGet sample.
- `OnPost` to handle form submissions.

The `Async` naming suffix is optional but is often used by convention for asynchronous functions. The preceding code is typical for Razor Pages.

If you're familiar with ASP.NET apps using controllers and views:

- The `OnPostAsync` code in the preceding example looks similar to typical controller code.
- Most of the MVC primitives like model binding, validation, Validation, and action results are shared.

The previous `OnPostAsync` method:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}
```

The basic flow of `OnPostAsync`:

Check for validation errors.

- If there are no errors, save the data and redirect.
- If there are errors, show the page again with validation messages. Client-side validation is identical to traditional ASP.NET Core MVC applications. In many cases, validation errors would be detected on the client, and never submitted to the server.

When the data is entered successfully, the `OnPostAsync` handler method calls the `RedirectToPage` helper method to return an instance of `RedirectToPageResult`. `RedirectToPage` is a new action result, similar to `RedirectToAction` or `RedirectToRoute`, but customized for pages. In the preceding sample, it redirects to the root Index page (`/Index`). `RedirectToPage` is detailed in the URL generation for Pages section.

When the submitted form has validation errors (that are passed to the server), the `OnPostAsync` handler method calls the `Page` helper method. `Page` returns an instance of `PageResult`. Returning `Page` is similar to how actions in controllers return `View`. `PageResult` is the default return type for a handler method. A handler method that returns `void` renders the page.

The `Customer` property uses `[BindProperty]` attribute to opt in to model binding.

```
public class CreateModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateModel(AppDbContext db)
    {
        _db = db;
    }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        return RedirectToPage("/Index");
    }
}
```

Razor Pages, by default, bind properties only with non-`GET` verbs. Binding to properties can reduce the amount of code you have to write. Binding reduces code by using the same property to render form fields (`<input asp-for="Customer.Name">`) and accept the input.

WARNING

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the `[BindProperty]` attribute's `SupportsGet` property to `true`:

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

The home page (*Index.cshtml*):

```
@page
@model RazorPagesContacts.Pages.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts</h1>
<form method="post">
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var contact in Model.Customers)
            {
                <tr>
                    <td>@contact.Id</td>
                    <td>@contact.Name</td>
                    <td>
                        <a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
                        <button type="submit" asp-page-handler="delete"
                                asp-route-id="@contact.Id">delete</button>
                    </td>
                </tr>
            }
        </tbody>
    </table>

    <a asp-page="./Create">Create</a>
</form>
```

The associated `PageModel` class (*Index.cshtml.cs*):

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Pages
{
    public class IndexModel : PageModel
    {
        private readonly AppDbContext _db;

        public IndexModel(AppDbContext db)
        {
            _db = db;
        }

        public IList<Customer> Customers { get; private set; }

        public async Task OnGetAsync()
        {
            Customers = await _db.Customers.AsNoTracking().ToListAsync();
        }

        public async Task<IActionResult> OnPostDeleteAsync(int id)
        {
            var contact = await _db.Customers.FindAsync(id);

            if (contact != null)
            {
                _db.Customers.Remove(contact);
                await _db.SaveChangesAsync();
            }

            return RedirectToPage();
        }
    }
}
```

The *Index.cshtml* file contains the following markup to create an edit link for each contact:

```
<a asp-page="./Edit" asp-route-id="@contact.Id">edit</a>
```

The `<a asp-page="./Edit" asp-route-id="@contact.Id">Edit</a>` Anchor Tag Helper used the `asp-route-{value}` attribute to generate a link to the Edit page. The link contains route data with the contact ID. For example, `https://localhost:5001/Edit/1`. Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. Tag Helpers are enabled by `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`

The *Pages/Edit.cshtml* file:

```
@page "{id:int}"
@model RazorPagesContacts.Pages.EditModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@{
    ViewData["Title"] = "Edit Customer";
}

<h1>Edit Customer - @Model.Customer.Id</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <input asp-for="Customer.Id" type="hidden" />
    <div>
        <label asp-for="Customer.Name"></label>
        <div>
            <input asp-for="Customer.Name" />
            <span asp-validation-for="Customer.Name" ></span>
        </div>
    </div>

    <div>
        <button type="submit">Save</button>
    </div>
</form>
```

The first line contains the `@page "{id:int}"` directive. The routing constraint `"{id:int}"` tells the page to accept requests to the page that contain `int` route data. If a request to the page doesn't contain route data that can be converted to an `int`, the runtime returns an HTTP 404 (not found) error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

The *Pages/Edit.cshtml.cs* file:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Customer = await _db.Customers.FindAsync(id);

            if (Customer == null)
            {
                return RedirectToPage("/Index");
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Attach(Customer).State = EntityState.Modified;

            try
            {
                await _db.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                throw new Exception($"Customer {Customer.Id} not found!");
            }

            return RedirectToPage("/Index");
        }
    }
}
```

The *Index.cshtml* file also contains markup to create a delete button for each customer contact:

```
<button type="submit" asp-page-handler="delete"
        asp-route-id="@contact.Id">delete</button>
```

When the delete button is rendered in HTML, its `formaction` includes parameters for:

- The customer contact ID specified by the `asp-route-id` attribute.

- The `handler` specified by the `asp-page-handler` attribute.

Here is an example of a rendered delete button with a customer contact ID of `1`:

```
<button type="submit" formaction="/?id=1&amp;handler=delete">delete</button>
```

When the button is selected, a form `POST` request is sent to the server. By convention, the name of the handler method is selected based on the value of the `handler` parameter according to the scheme `OnPost[handler]Async`.

Because the `handler` is `delete` in this example, the `OnPostDeleteAsync` handler method is used to process the `POST` request. If the `asp-page-handler` is set to a different value, such as `remove`, a handler method with the name `OnPostRemoveAsync` is selected. The following code shows the `OnPostDeleteAsync` handler:

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _db.Customers.FindAsync(id);

    if (contact != null)
    {
        _db.Customers.Remove(contact);
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

The `OnPostDeleteAsync` method:

- Accepts the `id` from the query string. If the *Index.cshtml* page directive contained routing constraint `"{id:int?}"`, `id` would come from route data. The route data for `id` is specified in the URI such as `https://localhost:5001/Customers/2`.
- Queries the database for the customer contact with `FindAsync`.
- If the customer contact is found, they're removed from the list of customer contacts. The database is updated.
- Calls `RedirectToPage` to redirect to the root Index page (`/Index`).

Mark page properties as required

Properties on a `PageModel` can be marked with the Required attribute:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        [Required(ErrorMessage = "Color is required")]
        public string Color { get; set; }

        public IActionResult OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            // Process color.

            return RedirectToPage("./Index");
        }
    }
}
```

For more information, see Model validation.

Handle HEAD requests with an OnGet handler fallback

`HEAD` requests allow you to retrieve the headers for a specific resource. Unlike `GET` requests, `HEAD` requests don't return a response body.

Ordinarily, an `OnHead` handler is created and called for `HEAD` requests:

```
public void OnHead()
{
    HttpContext.Response.Headers.Add("HandledBy", "Handled by OnHead!");
}
```

In ASP.NET Core 2.1 or later, Razor Pages falls back to calling the `OnGet` handler if no `OnHead` handler is defined. This behavior is enabled by the call to SetCompatibilityVersion in `Startup.ConfigureServices`:

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

The default templates generate the `SetCompatibilityVersion` call in ASP.NET Core 2.1 and 2.2. `SetCompatibilityVersion` effectively sets the Razor Pages option `AllowMappingHeadRequestsToGetHandler` to `true`.

Rather than opting in to all behaviors with `SetCompatibilityVersion`, you can explicitly opt in to *specific* behaviors. The following code opts in to allowing `HEAD` requests to be mapped to the `OnGet` handler:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.AllowMappingHeadRequestsToGetHandler = true;
    });
```

XSRF/CSRF and Razor Pages

You don't have to write any code for antiforgery validation. Antiforgery token generation and validation are automatically included in Razor Pages.

Using Layouts, partials, templates, and Tag Helpers with Razor Pages

Pages work with all the capabilities of the Razor view engine. Layouts, partials, templates, Tag Helpers, *_ViewStart.cshtml*, *_ViewImports.cshtml* work in the same way they do for conventional Razor views.

Let's declutter this page by taking advantage of some of those capabilities.

Add a layout page to *Pages/Shared/_Layout.cshtml*:

```
<!DOCTYPE html>
<html>
<head>
    <title>Razor Pages Sample</title>
</head>
<body>
   <a asp-page="/Index">Home</a>
    @RenderBody()
    <a asp-page="/Customers/Create">Create</a> <br />
</body>
</html>
```

The Layout:

- Controls the layout of each page (unless the page opts out of layout).
- Imports HTML structures such as JavaScript and stylesheets.

See layout page for more information.

The Layout property is set in *Pages/_ViewStart.cshtml*:

```
@{
    Layout = "_Layout";
}
```

The layout is in the *Pages/Shared* folder. Pages look for other views (layouts, templates, partials) hierarchically, starting in the same folder as the current page. A layout in the *Pages/Shared* folder can be used from any Razor page under the *Pages* folder.

The layout file should go in the *Pages/Shared* folder.

We recommend you **not** put the layout file in the *Views/Shared* folder. *Views/Shared* is an MVC views pattern. Razor Pages are meant to rely on folder hierarchy, not path conventions.

View search from a Razor Page includes the *Pages* folder. The layouts, templates, and partials you're using with MVC controllers and conventional Razor views *just work*.

Add a *Pages/_ViewImports.cshtml* file:

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

`@namespace` is explained later in the tutorial. The `@addTagHelper` directive brings in the built-in Tag Helpers to all the pages in the *Pages* folder.

When the `@namespace` directive is used explicitly on a page:

```
@page
@namespace RazorPagesIntro.Pages.Customers

@model NameSpaceModel

<h2>Name space</h2>
<p>
    @Model.Message
</p>
```

The directive sets the namespace for the page. The `@model` directive doesn't need to include the namespace.

When the `@namespace` directive is contained in *_ViewImports.cshtml*, the specified namespace supplies the prefix for the generated namespace in the Page that imports the `@namespace` directive. The rest of the generated namespace (the suffix portion) is the dot-separated relative path between the folder containing *_ViewImports.cshtml* and the folder containing the page.

For example, the `PageModel` class *Pages/Customers/Edit.cshtml.cs* explicitly sets the namespace:

```
namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        // Code removed for brevity.
```

The *Pages/_ViewImports.cshtml* file sets the following namespace:

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The generated namespace for the *Pages/Customers/Edit.cshtml* Razor Page is the same as the `PageModel` class.

`@namespace` *also works with conventional Razor views.*

The original *Pages/Create.cshtml* view file:

```
@page
@model RazorPagesContacts.Pages.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

The updated *Pages/Create.cshtml* view file:

```
@page
@model CreateModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" />
    </form>
</body>
</html>
```

The Razor Pages starter project contains the *Pages/_ValidationScriptsPartial.cshtml*, which hooks up client-side validation.

For more information on partial views, see Partial views in ASP.NET Core.

URL generation for Pages

The `Create` page, shown previously, uses `RedirectToPage`:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _db.Customers.Add(Customer);
    await _db.SaveChangesAsync();
    return RedirectToPage("/Index");
}
```

The app has the following file/folder structure:

- */Pages*
    - *Index.cshtml*

- */Customers*
  - *Create.cshtml*
  - *Edit.cshtml*
  - *Index.cshtml*

The *Pages/Customers/Create.cshtml* and *Pages/Customers/Edit.cshtml* pages redirect to *Pages/Index.cshtml* after success. The string `/Index` is part of the URI to access the preceding page. The string `/Index` can be used to generate URIs to the *Pages/Index.cshtml* page. For example:

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">My Index Page</a>`
- `RedirectToPage("/Index")`

The page name is the path to the page from the root */Pages* folder including a leading `/` (for example, `/Index`). The preceding URL generation samples offer enhanced options and functional capabilities over hardcoding a URL. URL generation uses routing and can generate and encode parameters according to how the route is defined in the destination path.

URL generation for pages supports relative names. The following table shows which Index page is selected with different `RedirectToPage` parameters from *Pages/Customers/Create.cshtml*:

REDIRECTTOPAGE(X)	PAGE
RedirectToPage("/Index")	*Pages/Index*
RedirectToPage("./Index");	*Pages/Customers/Index*
RedirectToPage("../Index")	*Pages/Index*
RedirectToPage("Index")	*Pages/Customers/Index*

`RedirectToPage("Index")`, `RedirectToPage("./Index")`, and `RedirectToPage("../Index")` are *relative names*. The `RedirectToPage` parameter is *combined* with the path of the current page to compute the name of the destination page.

Relative name linking is useful when building sites with a complex structure. If you use relative names to link between pages in a folder, you can rename that folder. All the links still work (because they didn't include the folder name).

To redirect to a page in a different Area, specify the area:

```
RedirectToPage("/Index", new { area = "Services" });
```

For more information, see Areas in ASP.NET Core.

ViewData attribute

Data can be passed to a page with ViewDataAttribute. Properties on controllers or Razor Page models with the `[ViewData]` attribute have their values stored and loaded from the ViewDataDictionary.

In the following example, the `AboutModel` contains a `Title` property marked with `[ViewData]`. The `Title` property is set to the title of the About page:

```
public class AboutModel : PageModel
{
    [ViewData]
    public string Title { get; } = "About";

    public void OnGet()
    {
    }
}
```

In the About page, access the `Title` property as a model property:

```
<h1>@Model.Title</h1>
```

In the layout, the title is read from the ViewData dictionary:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

TempData

ASP.NET Core exposes the TempData property on a controller. This property stores data until it's read. The `Keep` and `Peek` methods can be used to examine the data without deletion. `TempData` is useful for redirection, when data is needed for more than a single request.

The following code sets the value of `Message` using `TempData`:

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

The following markup in the *Pages/Customers/Index.cshtml* file displays the value of `Message` using `TempData`.

```
<h3>Msg: @Model.Message</h3>
```

The *Pages/Customers/Index.cshtml.cs* page model applies the `[TempData]` attribute to the `Message` property.

```
[TempData]
public string Message { get; set; }
```

For more information, see TempData .

Multiple handlers per page

The following page generates markup for two handlers using the `asp-page-handler` Tag Helper:

```
@page
@model CreateFATHModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

The form in the preceding example has two submit buttons, each using the `FormActionTagHelper` to submit to a different URL. The `asp-page-handler` attribute is a companion to `asp-page`. `asp-page-handler` generates URLs that submit to each of the handler methods defined by a page. `asp-page` isn't specified because the sample is linking to the current page.

The page model:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpperInvariant();
            return await OnPostJoinListAsync();
        }
    }
}
```

The preceding code uses *named handler methods*. Named handler methods are created by taking the text in the name after `On<HTTP Verb>` and before `Async` (if present). In the preceding example, the page methods are OnPost**JoinList**Async and OnPost**JoinListUC**Async. With *OnPost* and *Async* removed, the handler names are `JoinList` and `JoinListUC`.

```
<input type="submit" asp-page-handler="JoinList" value="Join" />
<input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinListUC`.

Custom routes

Use the `@page` directive to:

- Specify a custom route to a page. For example, the route to the About page can be set to `/Some/Other/Path` with `@page "/Some/Other/Path"`.

- Append segments to a page's default route. For example, an "item" segment can be added to a page's default route with `@page "item"`.
- Append parameters to a page's default route. For example, an ID parameter, `id`, can be required for a page with `@page "{id}"`.

A root-relative path designated by a tilde (`~`) at the beginning of the path is supported. For example, `@page "~/Some/Other/Path"` is the same as `@page "/Some/Other/Path"`.

If you don't like the query string `?handler=JoinList` in the URL, change the route to put the handler name in the path portion of the URL. The route can be customized by adding a route template enclosed in double quotes after the `@page` directive.

```
@page "{handler?}"
@model CreateRouteModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div>Name: <input asp-for="Customer.Name" /></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
    </form>
</body>
</html>
```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH/JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH/JoinListUC`.

The `?` following `handler` means the route parameter is optional.

Configuration and settings

To configure advanced options, use the extension method `AddRazorPagesOptions` on the MVC builder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.RootDirectory = "/MyPages";
            options.Conventions.AuthorizeFolder("/MyPages/Admin");
        });
}
```

Currently you can use the `RazorPagesOptions` to set the root directory for pages, or add application model conventions for pages. We'll enable more extensibility this way in the future.

To precompile views, see Razor view compilation .

Download or view sample code.

See Get started with Razor Pages, which builds on this introduction.

Specify that Razor Pages are at the content root

By default, Razor Pages are rooted in the */Pages* directory. Add WithRazorPagesAtContentRoot to AddMvc to specify that your Razor Pages are at the content root (ContentRootPath) of the app:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesAtContentRoot();
```

Specify that Razor Pages are at a custom root directory

Add WithRazorPagesRoot to AddMvc to specify that your Razor Pages are at a custom root directory in the app (provide a relative path):

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        ...
    })
    .WithRazorPagesRoot("/path/to/razor/pages");
```

Additional resources

- Authorize attribute and Razor Pages
- Introduction to ASP.NET Core
- razor syntax reference for ASP.NET Core
- Areas in ASP.NET Core
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Razor Pages authorization conventions in ASP.NET Core
- Razor Pages route and app conventions in ASP.NET Core
- Razor Pages unit tests in ASP.NET Core
- Partial views in ASP.NET Core

Tutorial: Create a Razor Pages web app with ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

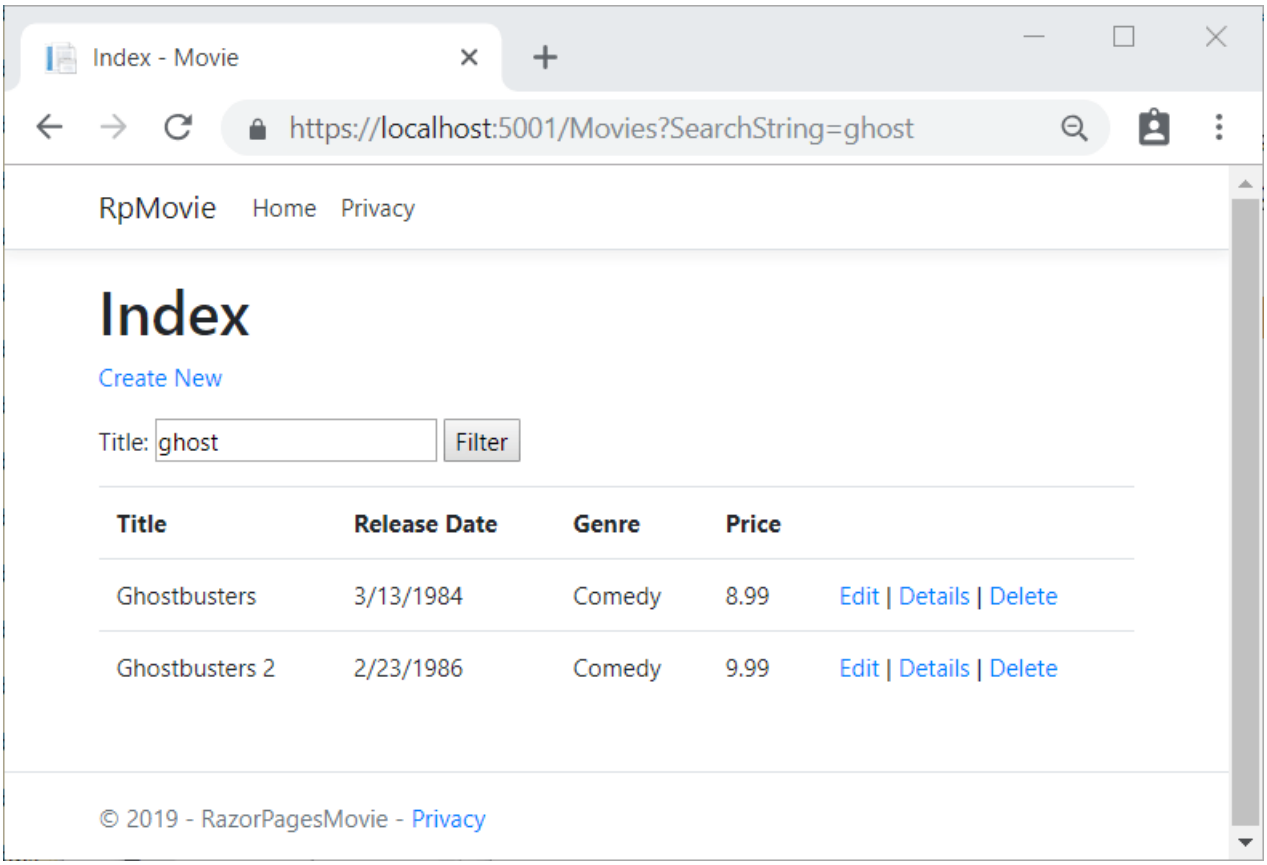
This series of tutorials explains the basics of building a Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages.

This series includes the following tutorials:

1. Create a Razor Pages web app
2. Add a model to a Razor Pages app
3. Scaffold (generate) Razor pages
4. Work with a database
5. Update Razor pages
6. Add search
7. Add a new field
8. Add validation

At the end, you'll have an app that can display and manage a database of movies.



Tutorial: Get started with Razor Pages in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

This is the first tutorial of a series that teaches the basics of building an ASP.NET Core Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages.

At the end of the series, you'll have an app that manages a database of movies.

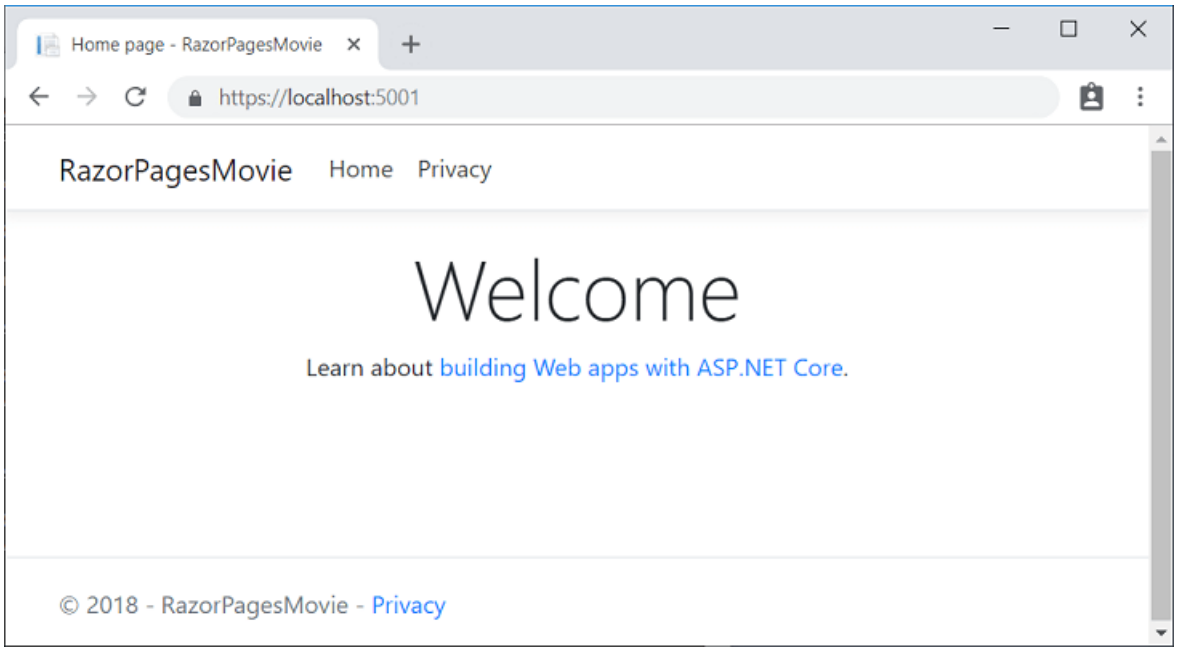
View or download sample code (how to download).

View or download sample code (how to download).

In this tutorial, you:

- Create a Razor Pages web app.
- Run the app.
- Examine the project files.

At the end of this tutorial, you'll have a working Razor Pages web app that you'll build on in later tutorials.



Prerequisites

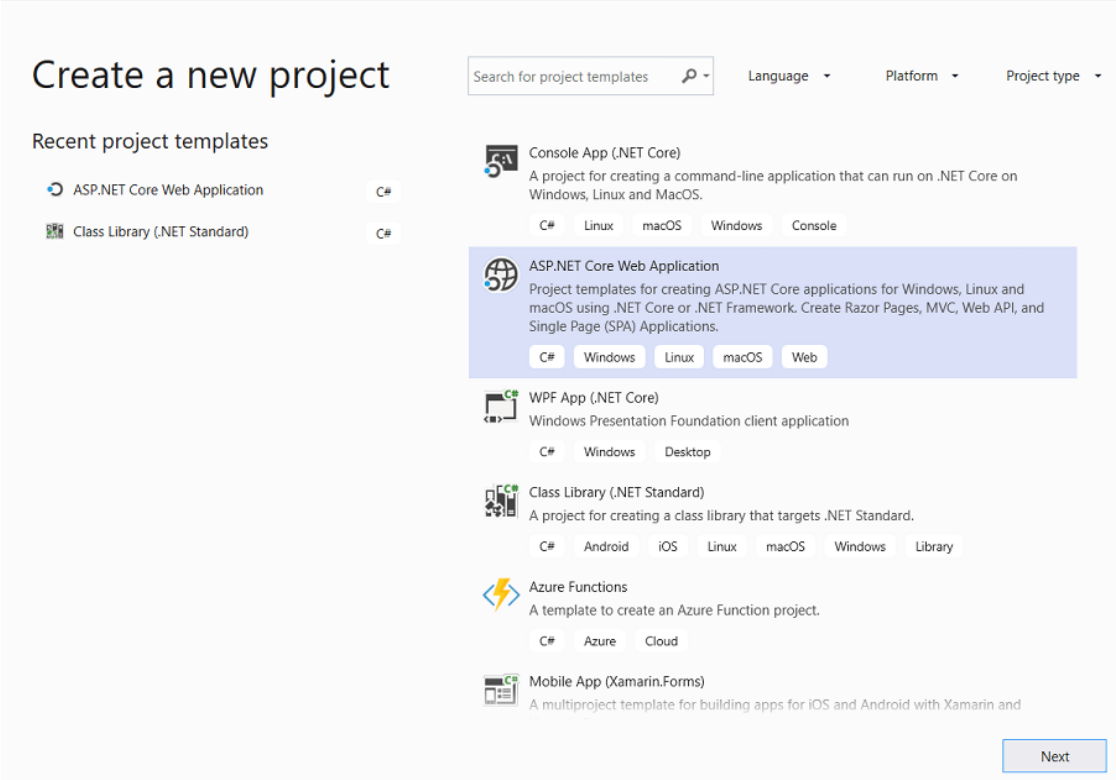
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

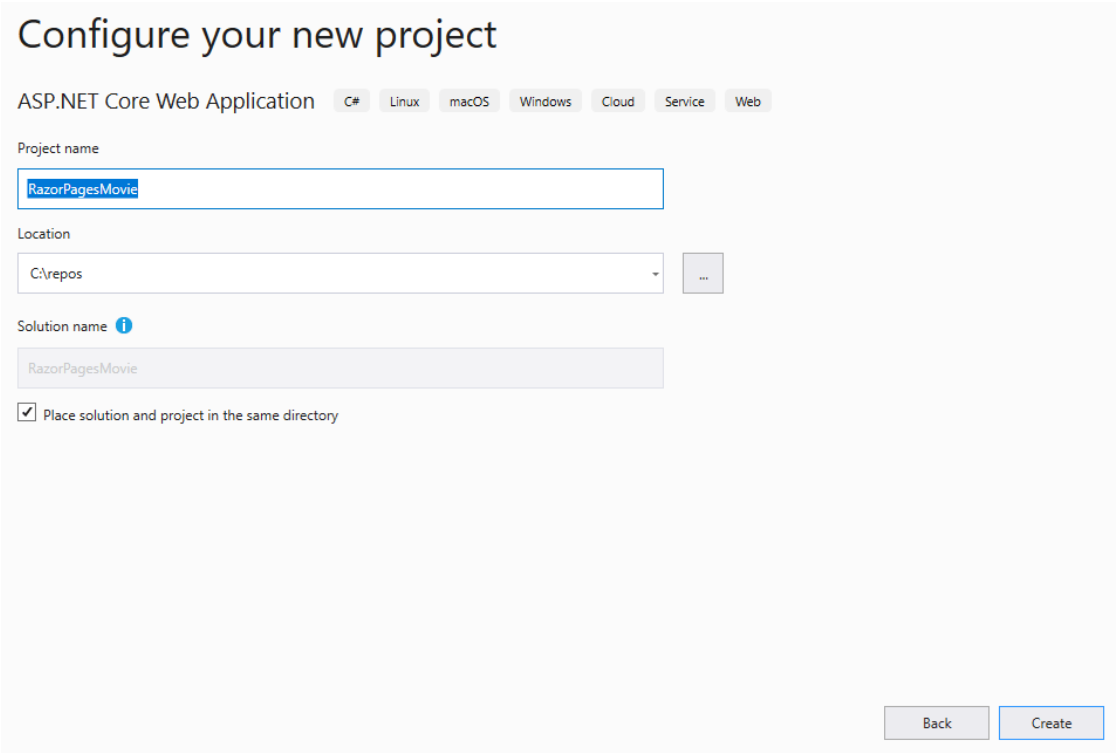
Create a Razor Pages web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

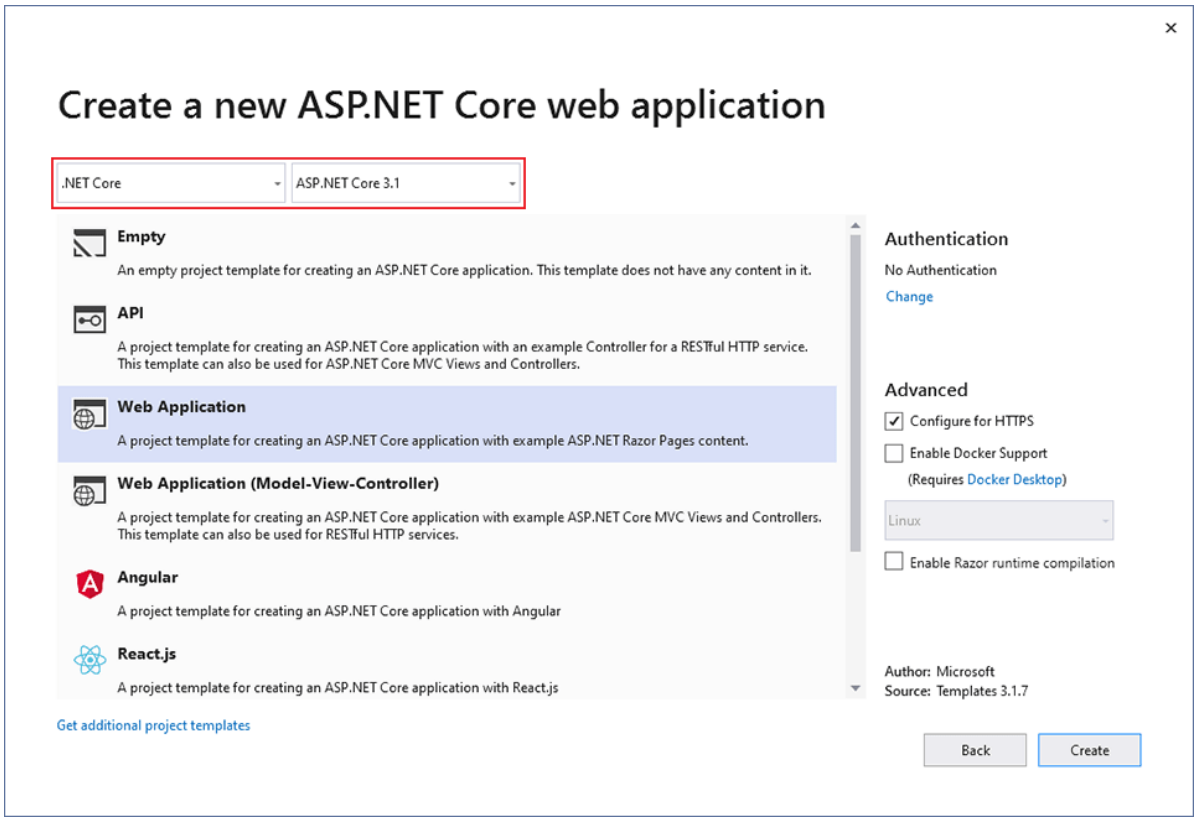
- From the Visual Studio **File** menu, select **New** > **Project**.
- Create a new ASP.NET Core Web Application and select **Next**.



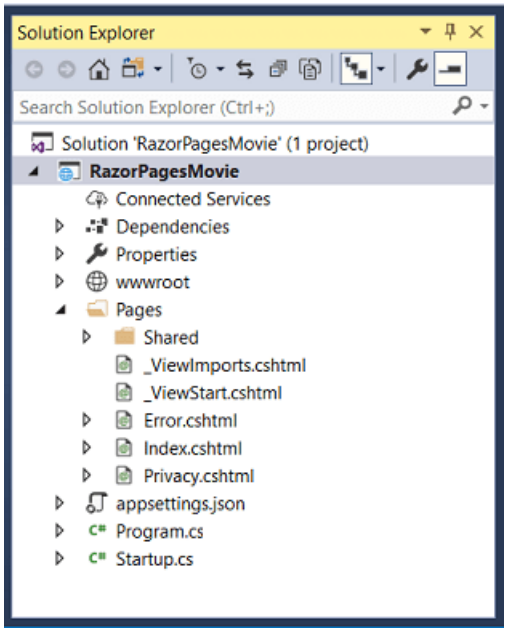
- Name the project **RazorPagesMovie**. It's important to name the project *RazorPagesMovie* so the namespaces will match when you copy and paste code.



- Select **ASP.NET Core 3.1** in the dropdown, **Web Application**, and then select **Create**.



The following starter project is created:

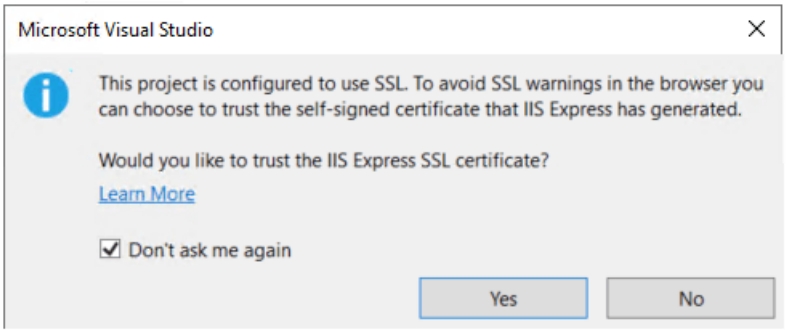


Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

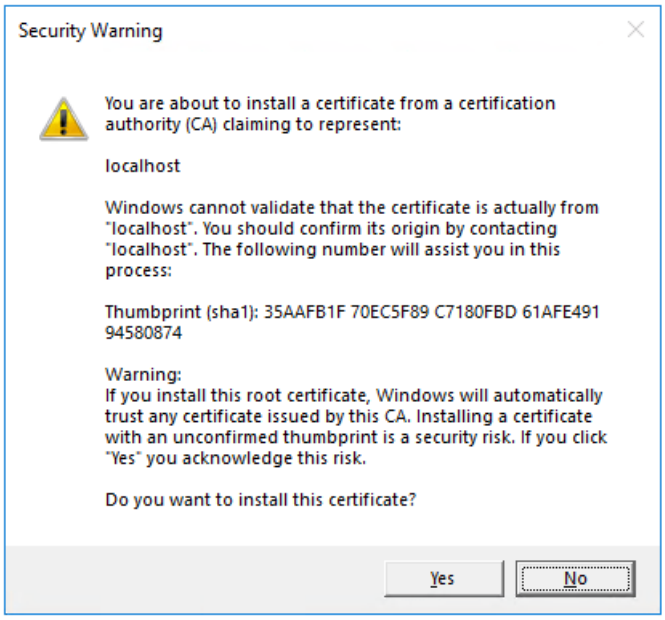
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

Visual Studio starts IIS Express and runs the app. The address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for the local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

Examine the project files

Here's an overview of the main project folders and files that you'll work with in later tutorials.

Pages folder

Contains Razor pages and supporting files. Each Razor page is a pair of files:

- A *.cshtml* file that contains HTML markup with C# code using Razor syntax.
- A *.cshtml.cs* file that contains C# code that handles page events.

Supporting files have names that begin with an underscore. For example, the *_Layout.cshtml* file configures UI elements common to all pages. This file sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page. For more information, see Layout in ASP.NET Core.

wwwroot folder

Contains static files, such as HTML files, JavaScript files, and CSS files. For more information, see Static files in ASP.NET Core.

appSettings.json

Contains configuration data, such as connection strings. For more information, see Configuration in ASP.NET Core.

Program.cs

Contains the entry point for the program. For more information, see .NET Generic Host.

Startup.cs

Contains code that configures app behavior. For more information, see App startup in ASP.NET Core.

Next steps

Advance to the next tutorial in the series:

ADD A MODEL

This is the first tutorial of a series. The series teaches the basics of building an ASP.NET Core Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages.

At the end of the series, you'll have an app that manages a database of movies.

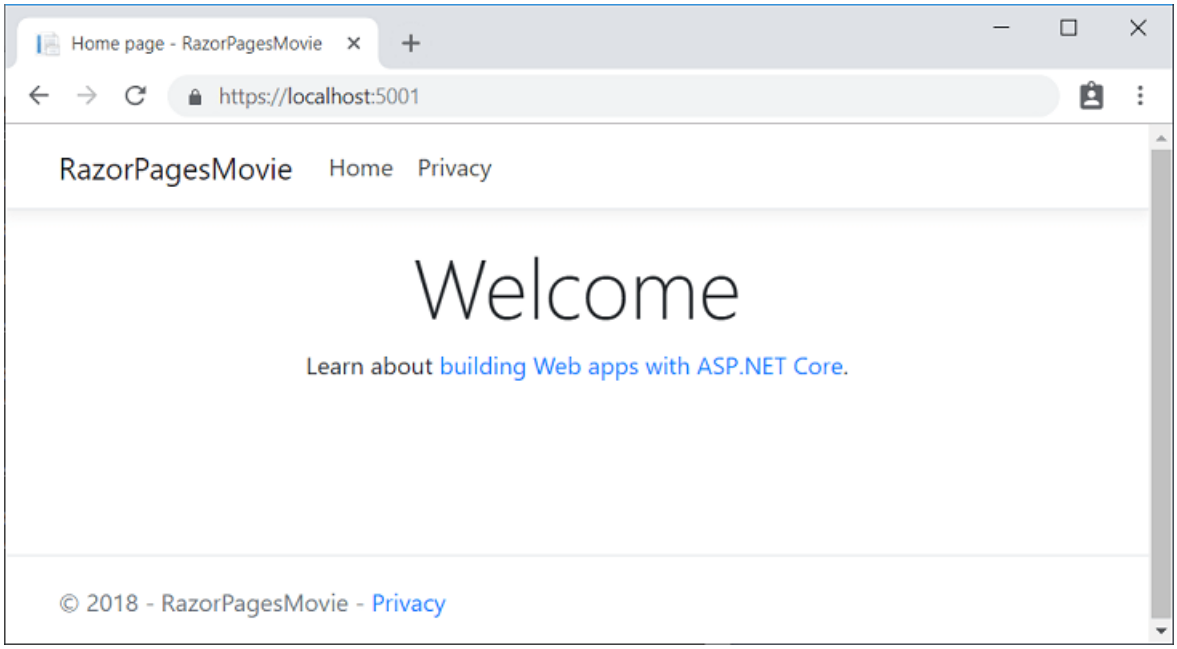
View or download sample code (how to download).

View or download sample code (how to download).

In this tutorial, you:

- Create a Razor Pages web app.
- Run the app.
- Examine the project files.

At the end of this tutorial, you'll have a working Razor Pages web app that you'll build on in later tutorials.



Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

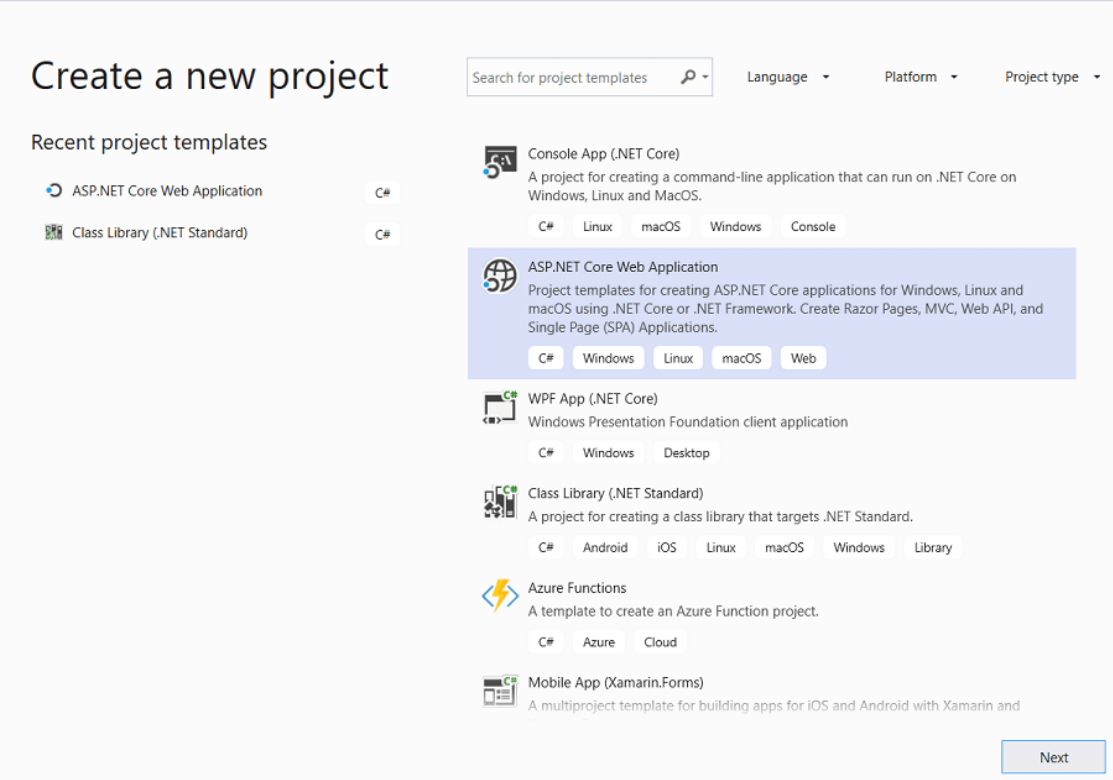
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

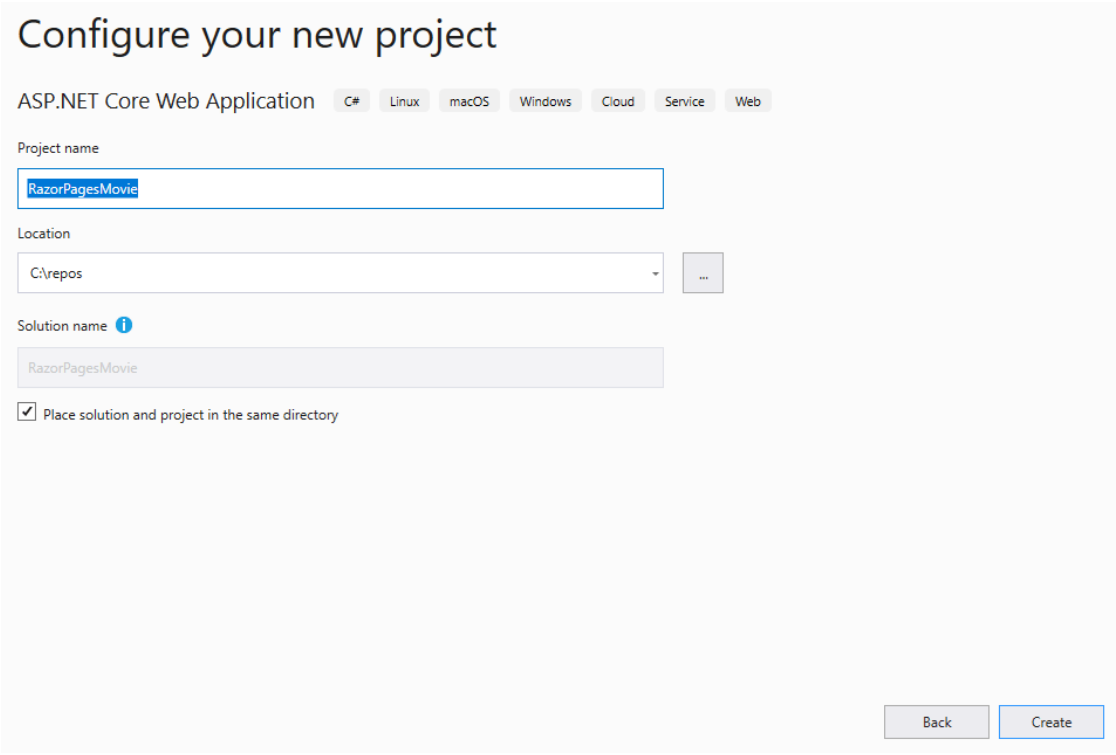
Create a Razor Pages web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

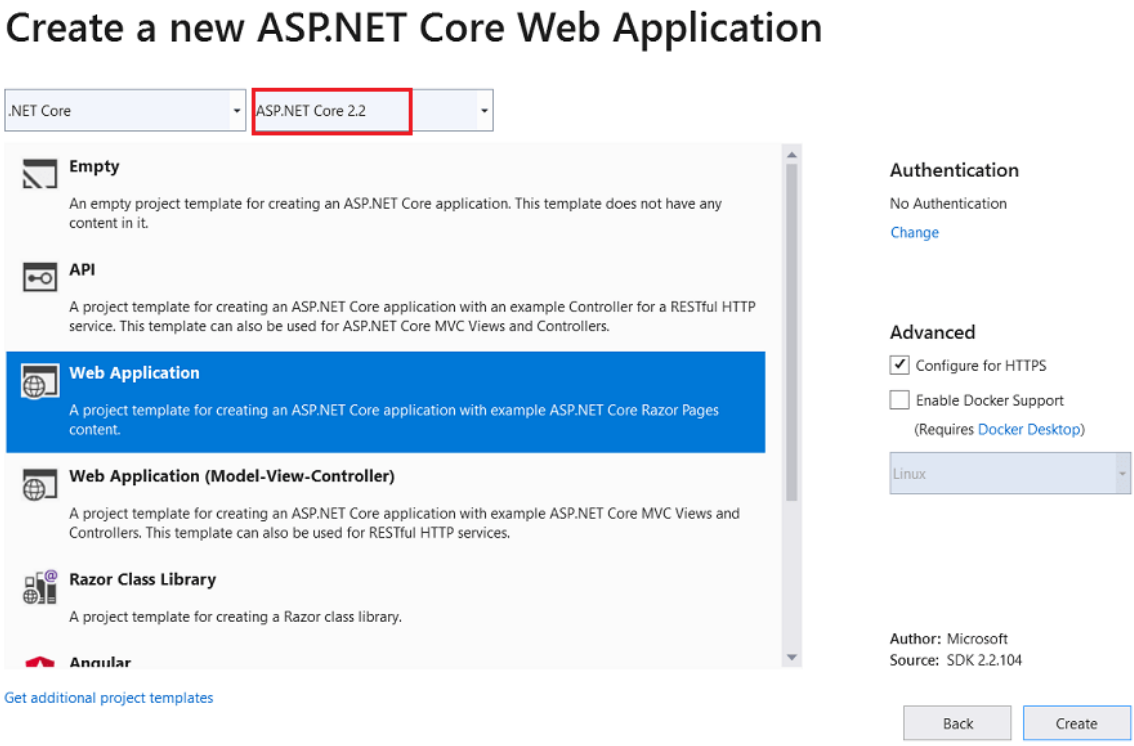
- From the Visual Studio **File** menu, select **New** > **Project**.
- Create a new ASP.NET Core Web Application and select **Next**.



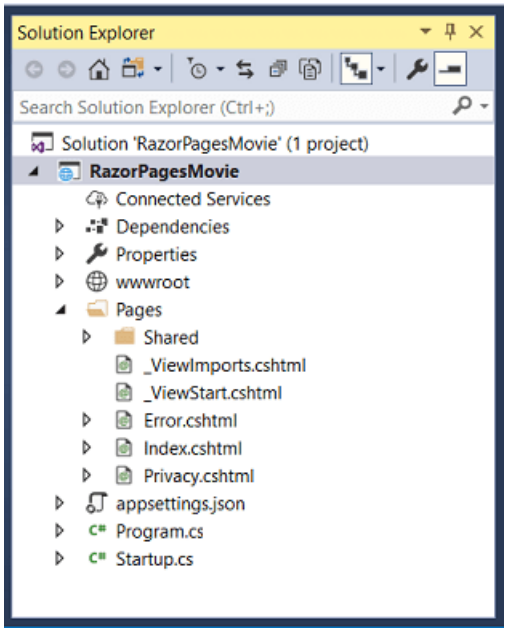
- Name the project **RazorPagesMovie**. It's important to name the project *RazorPagesMovie* so the namespaces will match when you copy and paste code.



- Select **ASP.NET Core 2.2** in the dropdown, **Web Application**, and then select **Create**.



The following starter project is created:

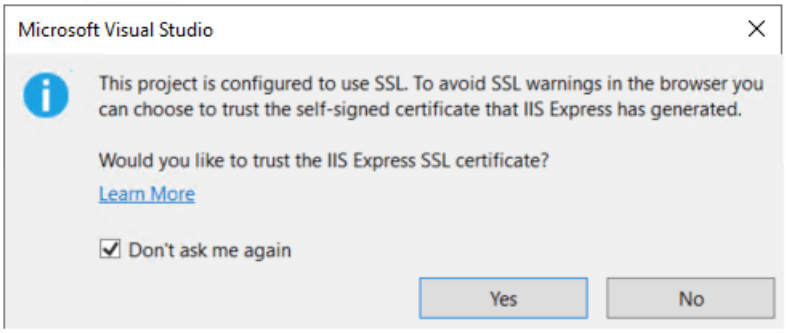


Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

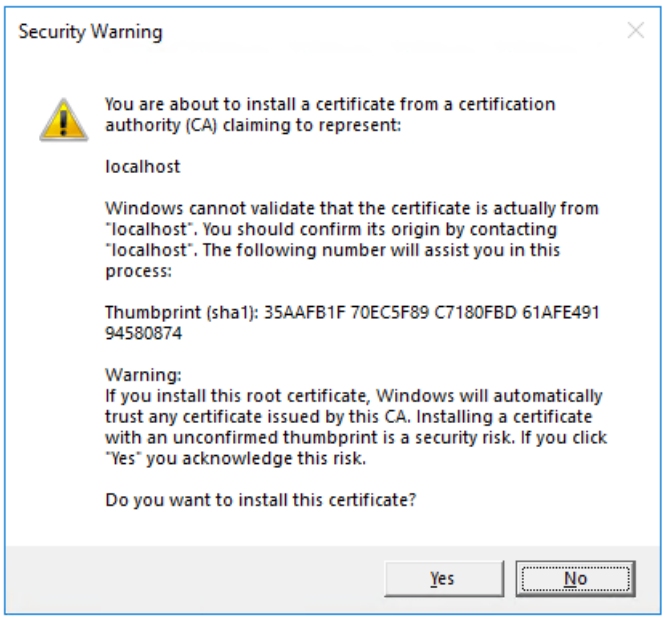
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog:



  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:

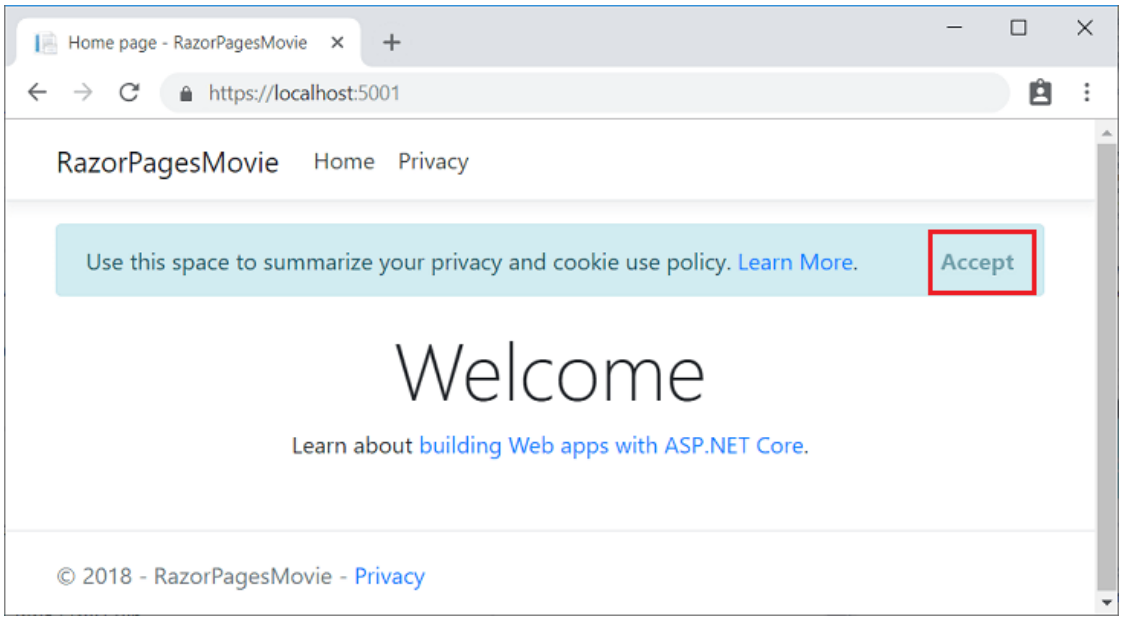


Select **Yes** if you agree to trust the development certificate.

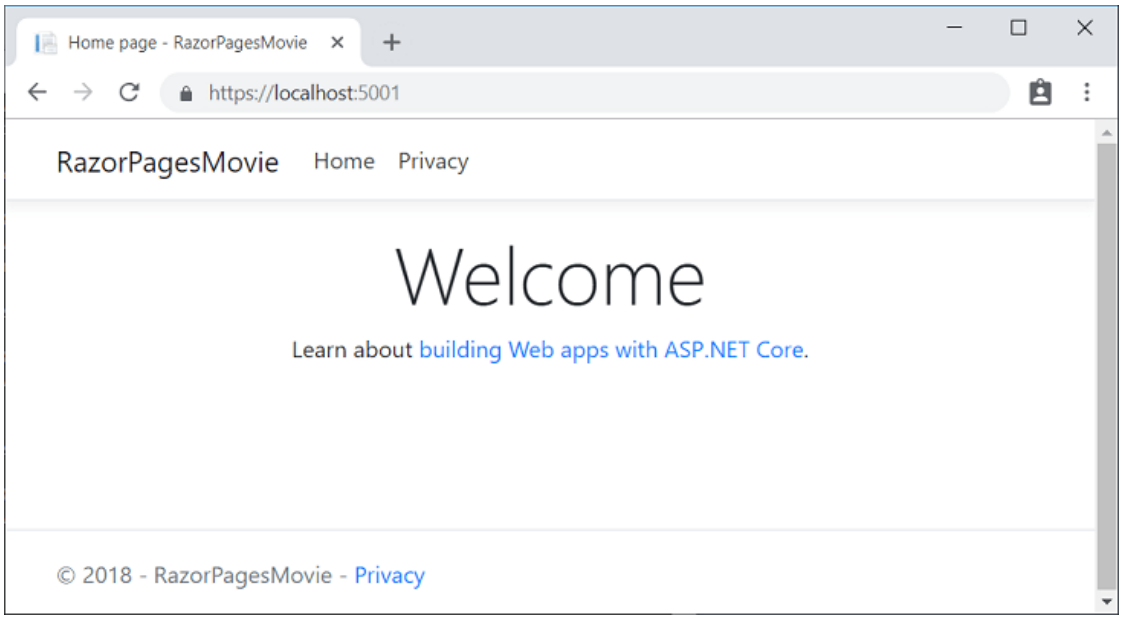
Visual Studio starts IIS Express and runs the app. The address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for the local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

- On the app's home page, select **Accept** to consent to tracking.

  This app doesn't track personal information, but the project template includes the consent feature in case you need it to comply with the European Union's General Data Protection Regulation (GDPR).



The following image shows the app after you give consent to tracking:



Examine the project files

Here's an overview of the main project folders and files that you'll work with in later tutorials.

Pages folder

Contains Razor pages and supporting files. Each Razor page is a pair of files:

- A *.cshtml* file that contains HTML markup with C# code using Razor syntax.
- A *.cshtml.cs* file that contains C# code that handles page events.

Supporting files have names that begin with an underscore. For example, the *_Layout.cshtml* file configures UI elements common to all pages. This file sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page. For more information, see Layout in ASP.NET Core.

wwwroot folder

Contains static files, such as HTML files, JavaScript files, and CSS files. For more information, see Static files in ASP.NET Core.

appSettings.json

Contains configuration data, such as connection strings. For more information, see Configuration in ASP.NET Core.

Program.cs

Contains the entry point for the program. For more information, see .NET Generic Host.

Startup.cs

Contains code that configures app behavior, such as whether it requires consent for cookies. For more information, see App startup in ASP.NET Core.

Additional resources

- Youtube version of this tutorial

Next steps

Advance to the next tutorial in the series:

ADD A MODEL

Part 2, add a model to a Razor Pages app in ASP.NET Core

9/22/2020 • 22 minutes to read • Edit Online

By Rick Anderson

In this section, classes are added for managing movies. The app's model classes use Entity Framework Core (EF Core) to work with the database. EF Core is an object-relational mapper (O/RM) that simplifies data access.

The model classes are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They define the properties of the data that are stored in the database.

View or download sample code (how to download).

View or download sample code (how to download).

Add a data model

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Right-click the **RazorPagesMovie** project > **Add** > **New Folder**. Name the folder *Models*.

Right click the *Models* folder. Select **Add** > **Class**. Name the class **Movie**.

Add the following properties to the `Movie` class:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains:

- The `ID` field is required by the database for the primary key.
- `[DataType(DataType.Date)]`: The DataType attribute specifies the type of the data (Date). With this attribute:
  - The user is not required to enter time information in the date field.
  - Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

Build the project to verify there are no compilation errors.

Scaffold the movie model

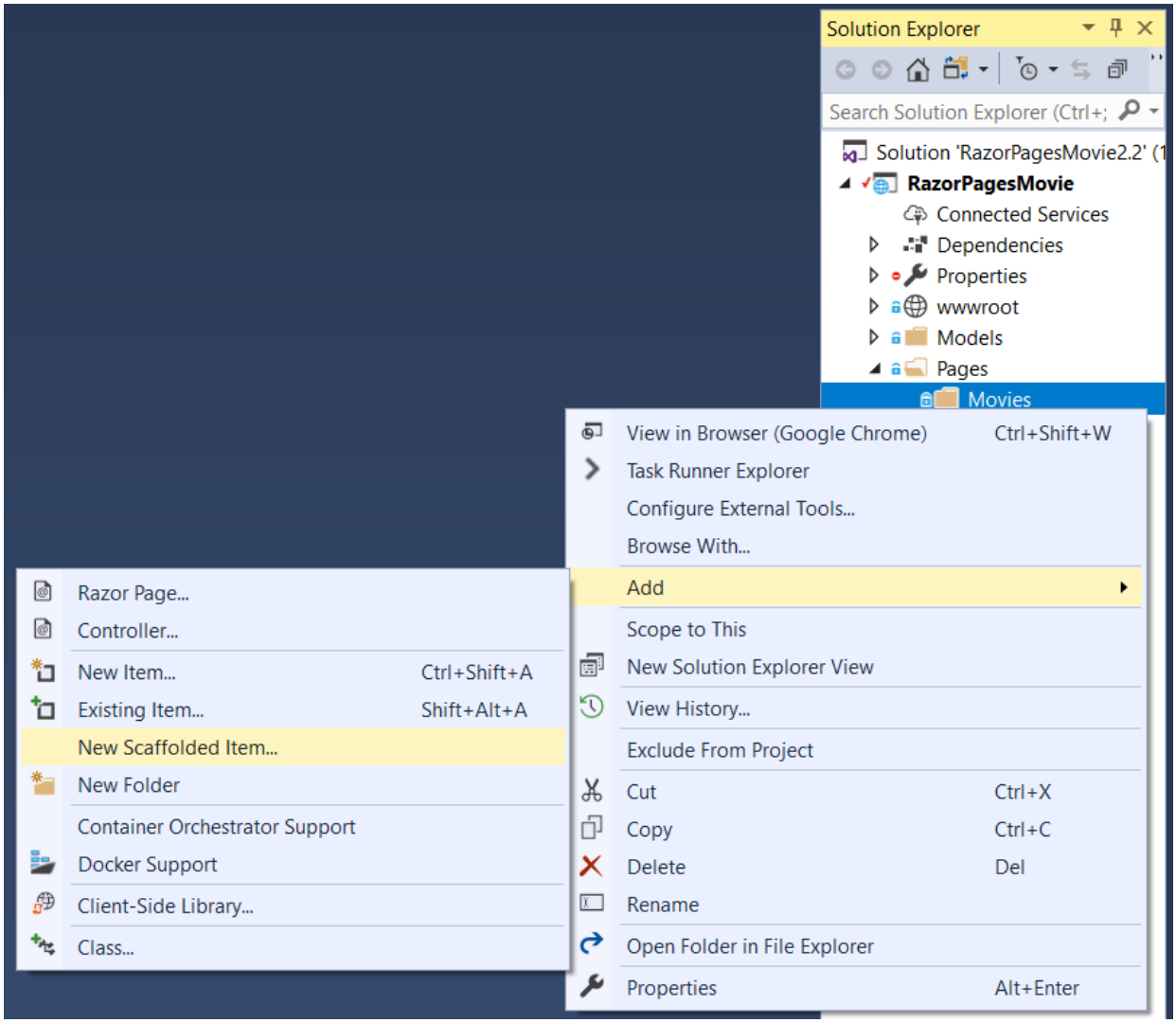
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

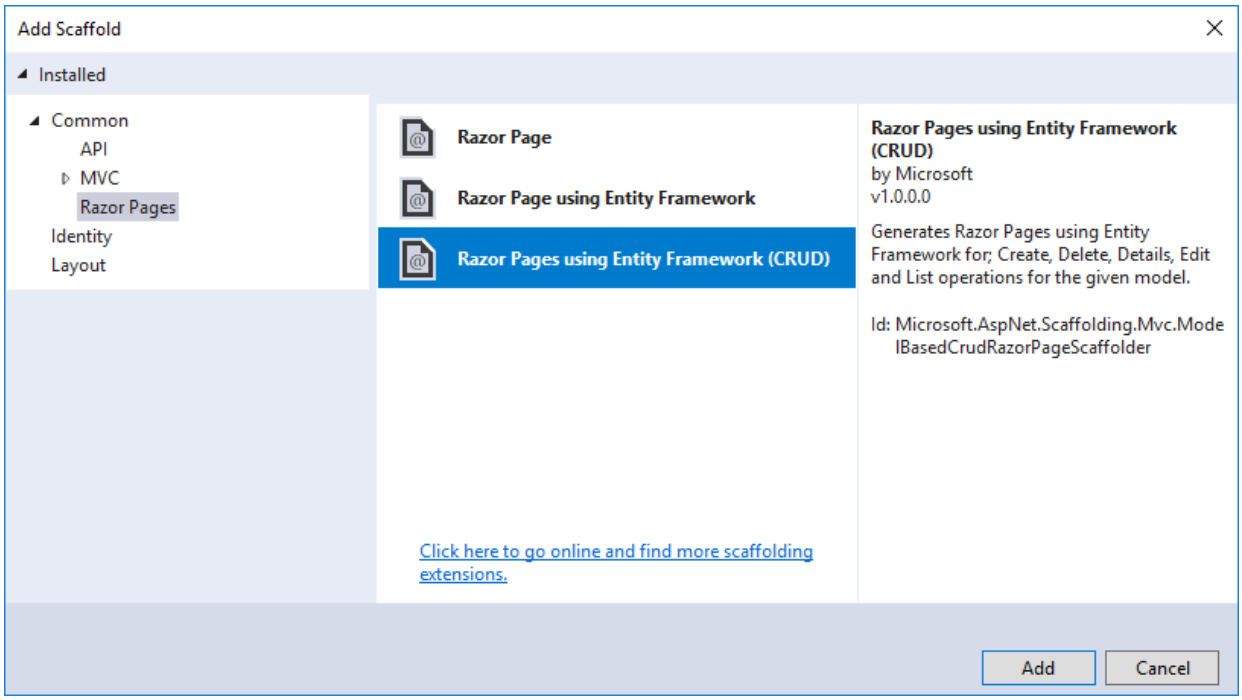
Create a *Pages/Movies* folder:

- Right click on the *Pages* folder > **Add** > **New Folder**.
- Name the folder *Movies*

Right click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.

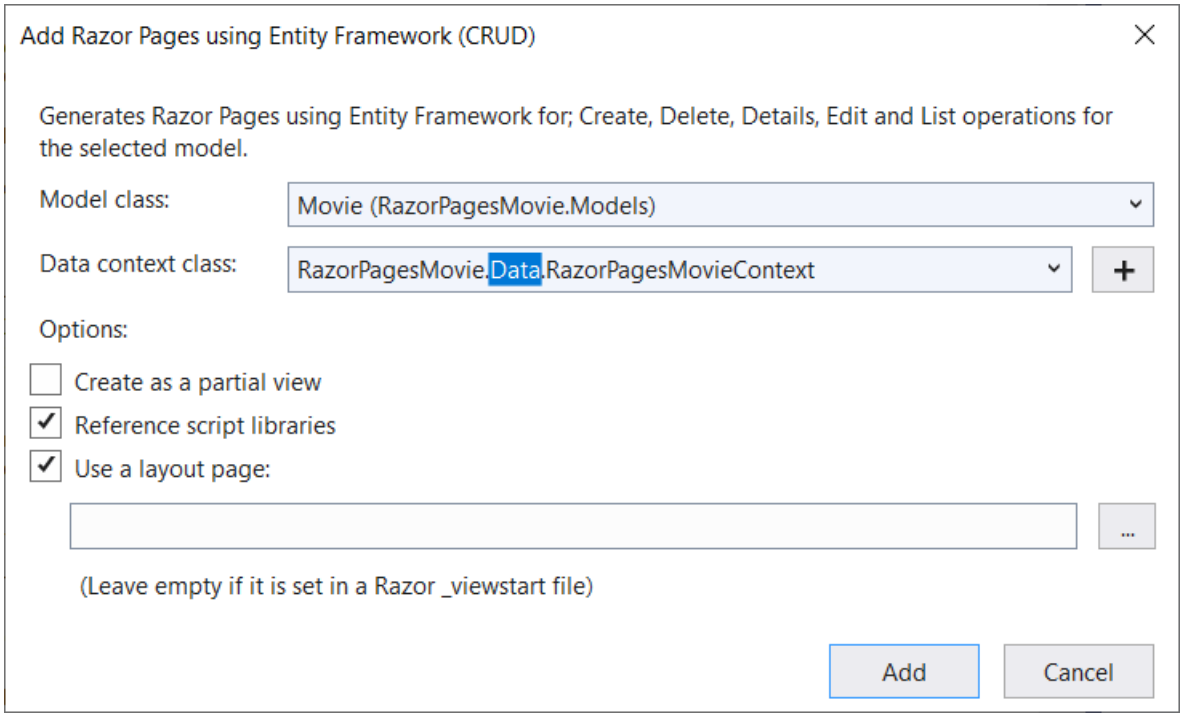


In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **Add**.



Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:

- In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.
- In the **Data context class** row, select the + (plus) sign and change the generated name from RazorPagesMovie.**Models**.RazorPagesMovieContext to RazorPagesMovie.**Data**.RazorPagesMovieContext. This change is not required. It creates the database context class with the correct namespace.
- Select **Add**.



The *appsettings.json* file is updated with the connection string used to connect to a local database.

Files created

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

The scaffold process creates and updates the following files:

- *Pages/Movies*: Create, Delete, Details, Edit, and Index.
- *Data/RazorPagesMovieContext.cs*

Updated

- *Startup.cs*

The created and updated files are explained in the next section.

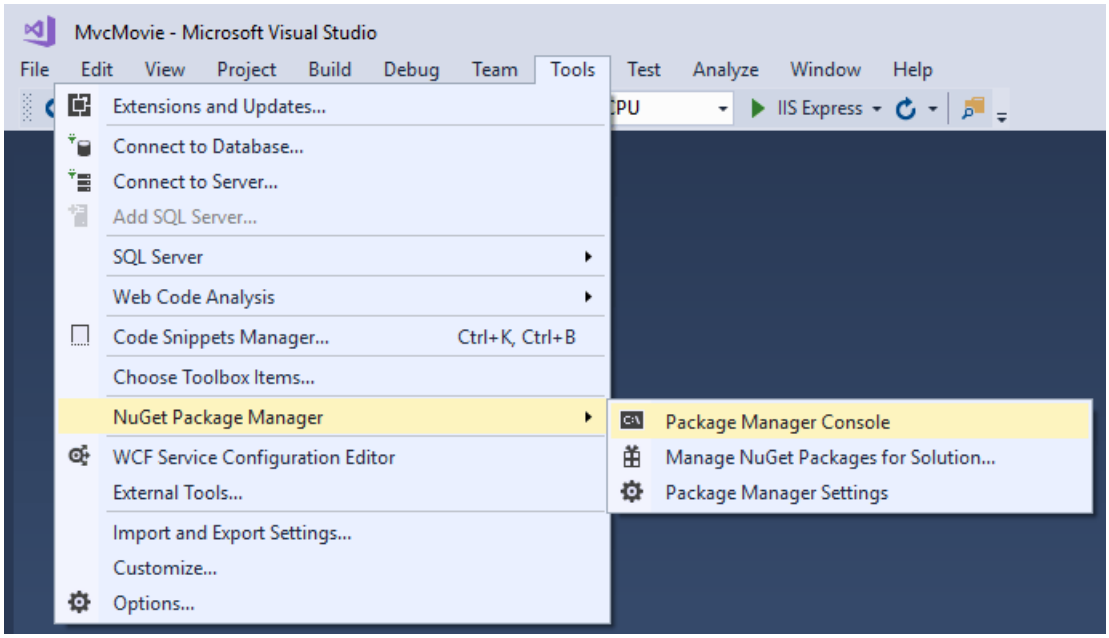
Initial migration

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In this section, the Package Manager Console (PMC) is used to:

- Add an initial migration.
- Update the database with the initial migration.

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration InitialCreate
Update-Database
```

The preceding commands generate the following warning: "No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'."

You can ignore that warning, it will be fixed in a later tutorial.

The migrations command generates code to create the initial database schema. The schema is based on the model specified in `DbContext`. The `InitialCreate` argument is used to name the migrations. Any name can be used, but by convention a name is selected that describes the migration.

The `update` command runs the `Up` method in migrations that have not been applied. In this case, `update` runs

the `Up` method in *Migrations/<time-stamp>_InitialCreate.cs* file, which creates the database.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

The scaffolding tool automatically created a DB context and registered it with the dependency injection container.

Examine the `Startup.ConfigureServices` method. The highlighted line was added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

The `RazorPagesMovieContext` coordinates EF Core functionality (Create, Read, Update, Delete, etc.) for the `Movie` model. The data context (`RazorPagesMovieContext`) is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model.

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Test the app

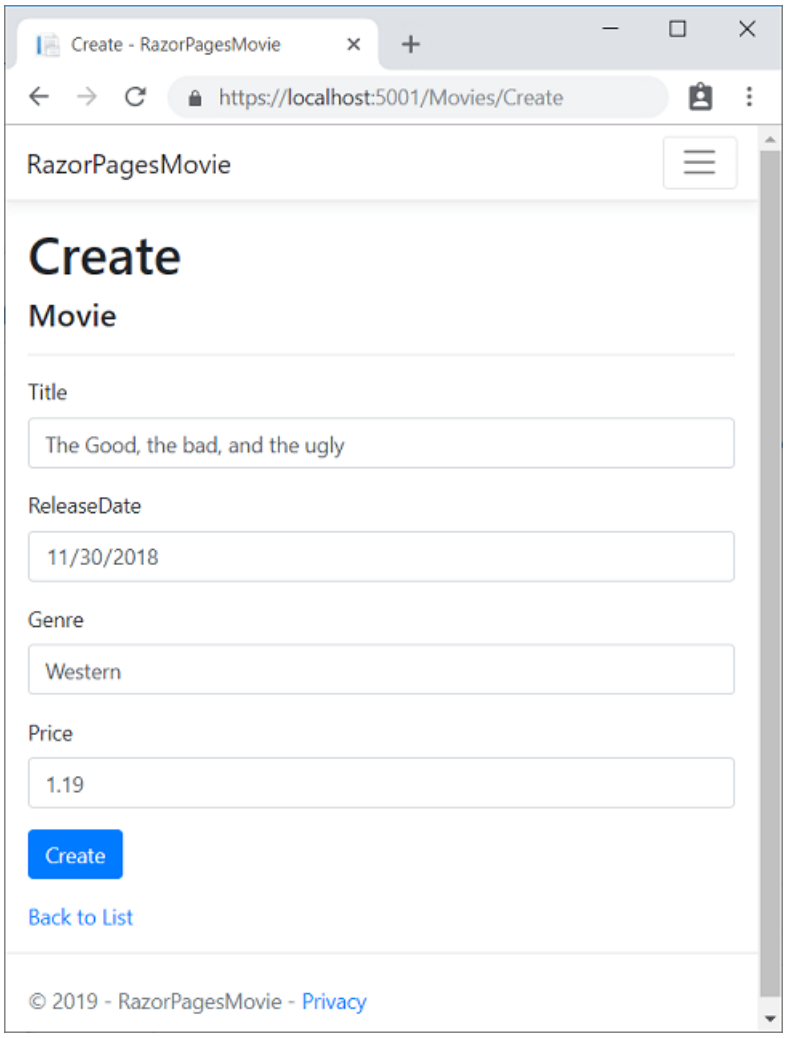
- Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

If you get the error:

```
SqlException: Cannot open database "RazorPagesMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

- Test the **Create** link.



> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** links.

The next tutorial explains the files created by scaffolding.

Additional resources

PREVIOUS: GET STARTED
NEXT: SCAFFOLDED RAZOR PAGES

In this section, classes are added for managing movies in a cross-platform SQLite database. Apps created from an ASP.NET Core template use a SQLite database. The app's model classes are used with Entity Framework Core

(EF Core) (SQLite EF Core Database Provider) to work with the database. EF Core is an object-relational mapping (ORM) framework that simplifies data access.

The model classes are known as POCO classes (from "plain-old CLR objects") because they don't have any dependency on EF Core. They define the properties of the data that are stored in the database.

View or download sample code (how to download).

View or download sample code (how to download).

Add a data model

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Right-click the **RazorPagesMovie** project > **Add** > **New Folder**. Name the folder *Models*.

Right click the *Models* folder. Select **Add** > **Class**. Name the class **Movie**.

Add the following properties to the `Movie` class:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains:

- The `ID` field is required by the database for the primary key.
- `[DataType(DataType.Date)]`: The DataType attribute specifies the type of the data (Date). With this attribute:
  - The user is not required to enter time information in the date field.
  - Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

Build the project to verify there are no compilation errors.

Scaffold the movie model

In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

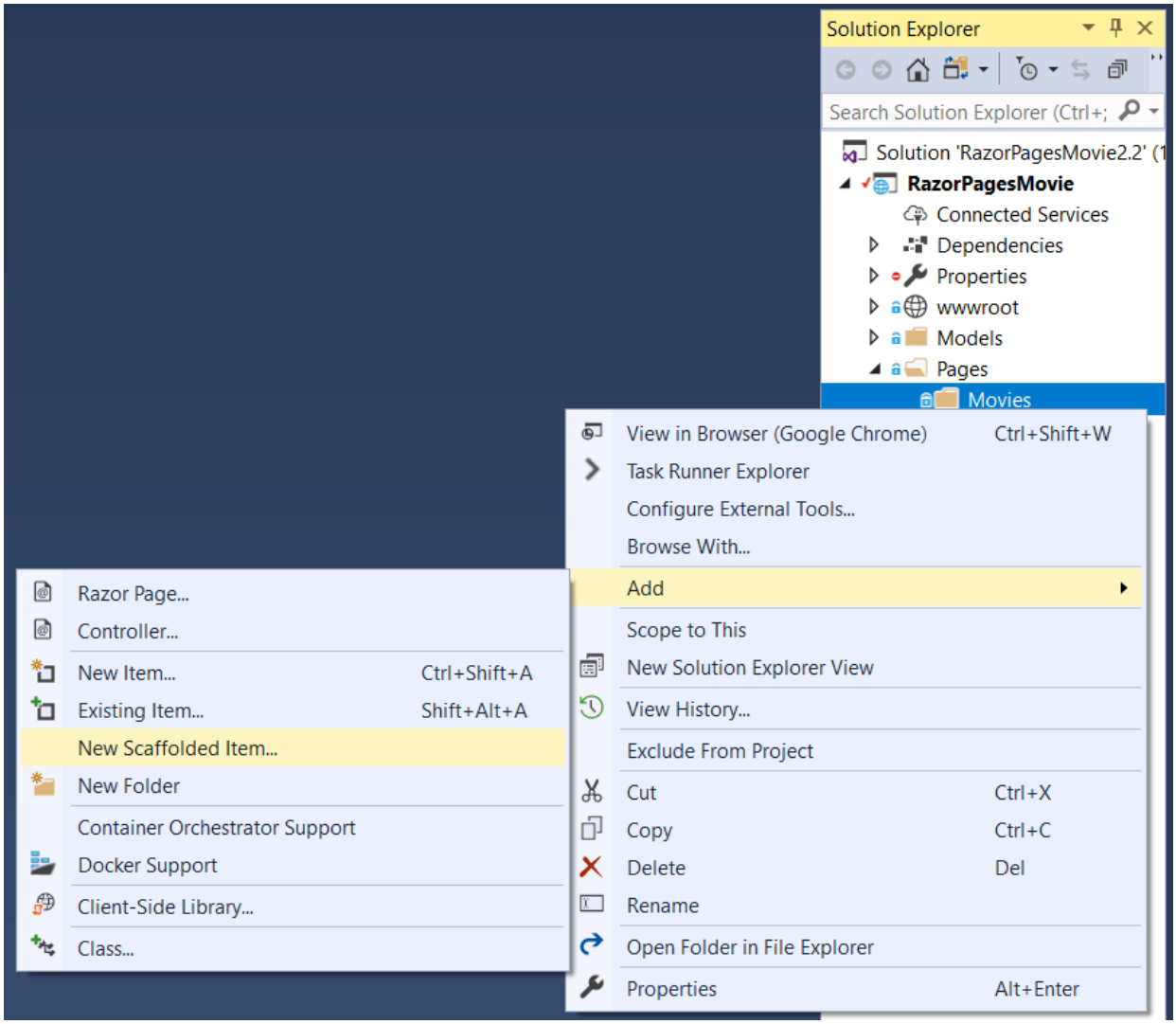
- Visual Studio
- Visual Studio Code

- Visual Studio for Mac

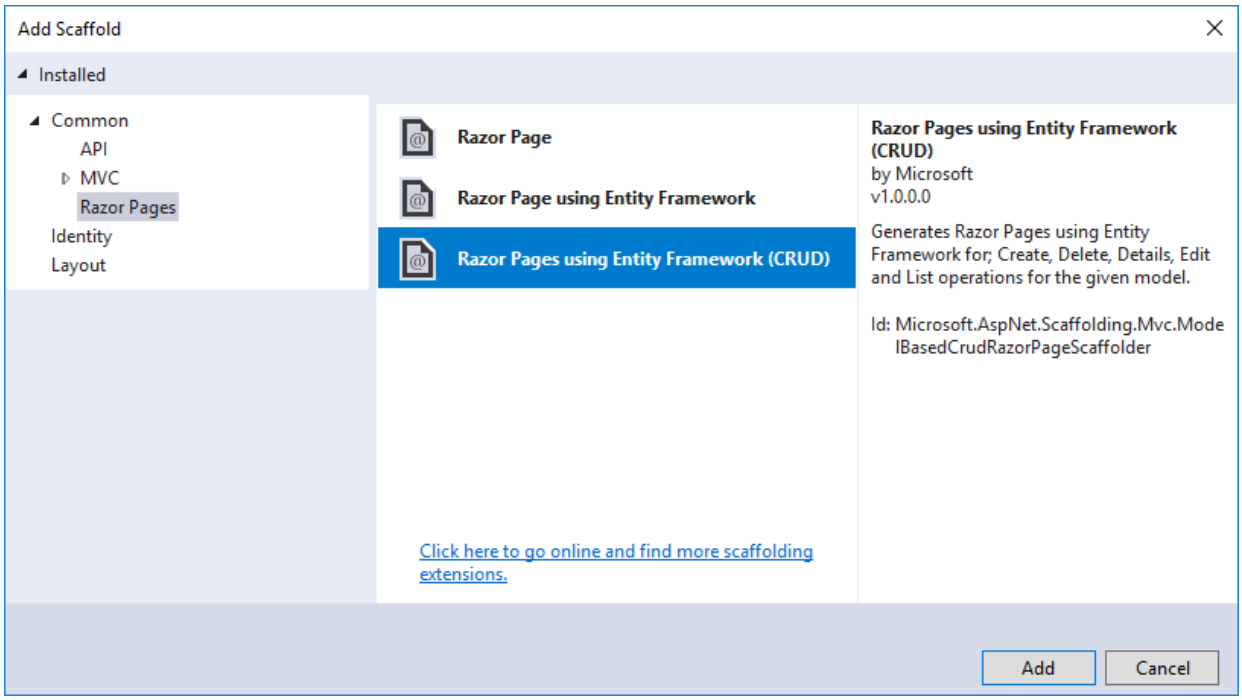
Create a *Pages/Movies* folder:

- Right click on the *Pages* folder > **Add** > **New Folder**.
- Name the folder *Movies*

Right click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.

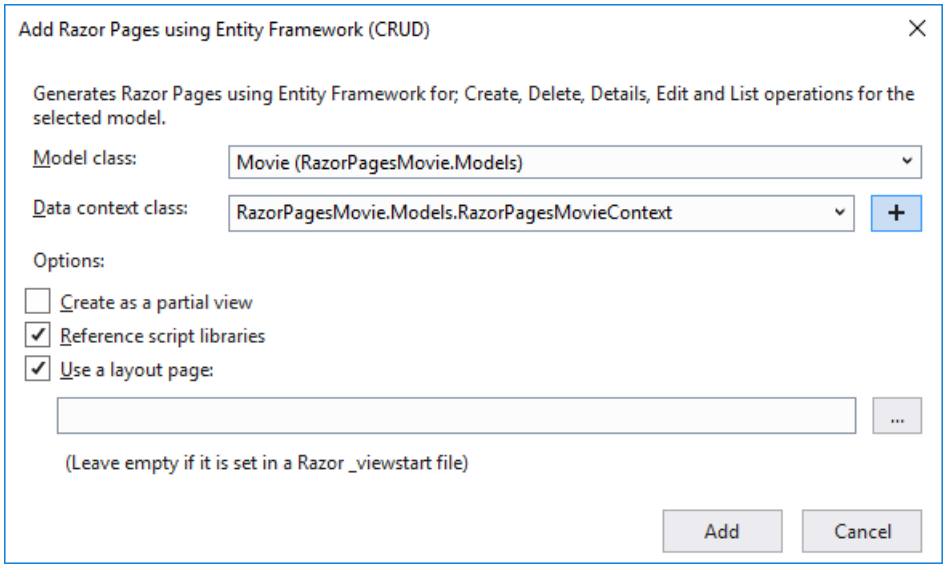


In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **Add**.



Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:

- In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.
- In the **Data context class** row, select the **+** (plus) sign and accept the generated name **RazorPagesMovie.Models.RazorPagesMovieContext**.
- Select **Add**.



The *appsettings.json* file is updated with the connection string used to connect to a local database.

The scaffold process creates and updates the following files:

Files created

- *Pages/Movies*: Create, Delete, Details, Edit, and Index.
- *Data/RazorPagesMovieContext.cs*

File updated

- *Startup.cs*

The created and updated files are explained in the next section.

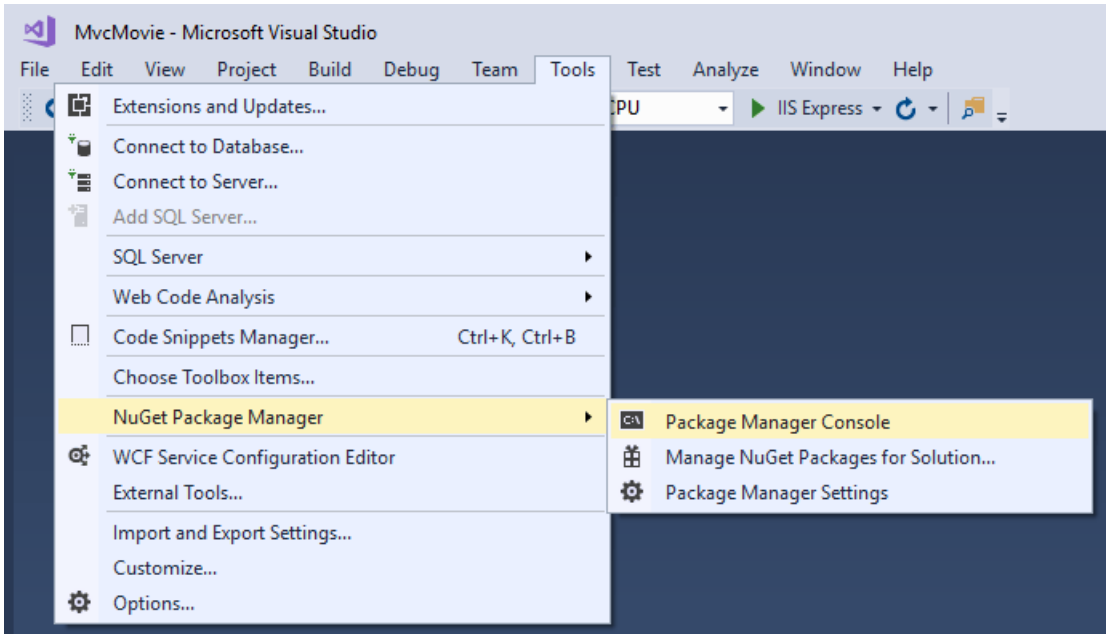
Initial migration

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In this section, the Package Manager Console (PMC) is used to:

- Add an initial migration.
- Update the database with the initial migration.

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration Initial
Update-Database
```

The `Add-Migration` command generates code to create the initial database schema. The schema is based on the model specified in the `DbContext` (In the *RazorPagesMovieContext.cs* file). The `InitialCreate` argument is used to name the migration. Any name can be used, but by convention a name that describes the migration is used. For more information, see Tutorial: Using the migrations feature - ASP.NET MVC with EF Core.

The `Update-Database` command runs the `Up` method in the *Migrations/<time-stamp>_InitialCreate.cs* file. The `Up` method creates the database.

> **NOTE**
>
> The preceding commands generate the following warning: "*No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.*" You can ignore that warning, it will be fixed in a later tutorial.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

The scaffolding tool automatically created a DB context and registered it with the dependency injection container.

Examine the `Startup.ConfigureServices` method. The highlighted line was added by the scaffolder:

```
// This method gets called by the runtime.
// Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is
        // needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

The `RazorPagesMovieContext` coordinates EF Core functionality (Create, Read, Update, Delete, etc.) for the `Movie` model. The data context (`RazorPagesMovieContext`) is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model.

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie.Models
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext (DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Test the app

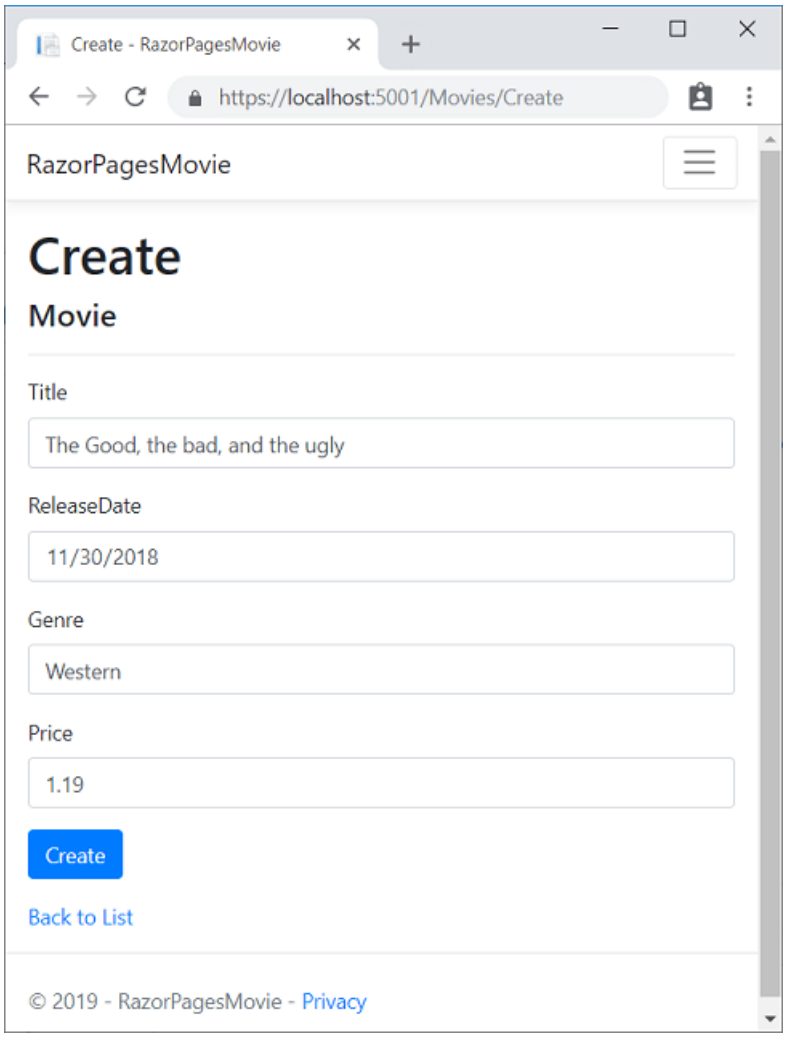
- Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

If you get the error:

```
SqlException: Cannot open database "RazorPagesMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

- Test the **Create** link.



> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** links.

The next tutorial explains the files created by scaffolding.

Additional resources

PREVIOUS: GET STARTED
NEXT: SCAFFOLDED RAZOR PAGES

Part 3, scaffolded Razor Pages in ASP.NET Core

9/22/2020 • 16 minutes to read • Edit Online

By Rick Anderson

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial.

View or download sample code (how to download).

View or download sample code (how to download).

The Create, Delete, Details, and Edit pages

Examine the *Pages/Movies/Index.cshtml.cs* Page Model:

```
// Unused usings removed.
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

        public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get;set; }

        public async Task OnGetAsync()
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor Pages are derived from `PageModel`. By convention, the `PageModel`-derived class is called `<PageName>Model`. The constructor uses dependency injection to add the `RazorPagesMovieContext` to the page. All the scaffolded pages follow this pattern. See Asynchronous code for more information on asynchronous programming with Entity Framework.

When a request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. `OnGetAsync` or `OnGet` is called to initialize the state of the page. In this case, `OnGetAsync` gets a list of movies and displays them.

When `OnGet` returns `void` or `OnGetAsync` returns `Task`, no return statement is used. When the return type is `IActionResult` or `Task<IActionResult>`, a return statement must be provided. For example, the *Pages/Movies/Create.cshtml.cs* `OnPostAsync` method:

```
    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Movie.Add(Movie);
        await _context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Examine the *Pages/Movies/Index.cshtml* Razor Page:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup, otherwise it transitions into C#.

The @page directive

The `@page` Razor directive makes the file an MVC action, which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` is an example of transitioning into Razor-specific markup. See Razor syntax for more information.

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movie[0].Title)
```

The `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` is `null` or empty. When the lambda expression is evaluated (for example, with `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

The @model directive

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel`-derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayFor` HTML Helpers on the page.

The layout page

Select the menu links (**RazorPagesMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Pages/Shared/_Layout.cshtml* file. Open the *Pages/Shared/_Layout.cshtml* file.

Layout templates allow the HTML container layout to be:

- Specified in one place.
- Applied in multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the page-specific views show up, *wrapped* in the layout page. For example, select the **Privacy** link and the *Pages/Privacy.cshtml* view is rendered inside the `RenderBody` method.

ViewData and layout

Consider the following markup from the *Pages/Movies/Index.cshtml* file:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

The preceding highlighted markup is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class contains a `ViewData` dictionary property that can be used to pass data to a View. Objects are added to the `ViewData` dictionary using a key/value pattern. In the preceding sample, the `"Title"` property is added to the `ViewData` dictionary.

The `"Title"` property is used in the *Pages/Shared/_Layout.cshtml* file. The following markup shows the first few lines of the *_Layout.cshtml* file.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>

    @*Markup removed for brevity.*@
```

The line `@*Markup removed for brevity.*@` is a Razor comment. Unlike HTML comments (`<!-- -->`), Razor comments are not sent to the client.

Update the layout

Change the `<title>` element in the *Pages/Shared/_Layout.cshtml* file to display **Movie** rather than **RazorPagesMovie**.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie</title>
```

Find the following anchor element in the *Pages/Shared/_Layout.cshtml* file.

```
<a class="navbar-brand" asp-area="" asp-page="/Index">RazorPagesMovie</a>
```

Replace the preceding element with the following markup:

```
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

The preceding anchor element is a Tag Helper. In this case, it's the Anchor Tag Helper. The `asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page. The `asp-area` attribute value is empty, so the area isn't used in the link. See Areas for more information.

Save your changes, and test the app by clicking on the **RpMovie** link. See the _Layout.cshtml file in GitHub if you have any problems.

Test the other links (**Home**, **RpMovie**, **Create**, **Edit**, and **Delete**). Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark.

> **NOTE**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

The `Layout` property is set in the *Pages/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The preceding markup sets the layout file to *Pages/Shared/_Layout.cshtml* for all Razor files under the *Pages*

folder. See Layout for more information.

The Create page model

Examine the *Pages/Movies/Create.cshtml.cs* page model:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

        public CreateModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize, so `Page` is returned. Later in the tutorial, an example of `OnGet` initializing state is shown. The `Page` method creates a `PageResult` object that renders the *Create.cshtml* page.

The `Movie` property uses the `[BindProperty]` attribute to opt-in to model binding. When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date. Client-side validation and model validation are discussed later in the tutorial.

If there are no model errors, the data is saved, and the browser is redirected to the Index page.

The Create Razor Page

Examine the *Pages/Movies/Create.cshtml* Razor Page file:

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

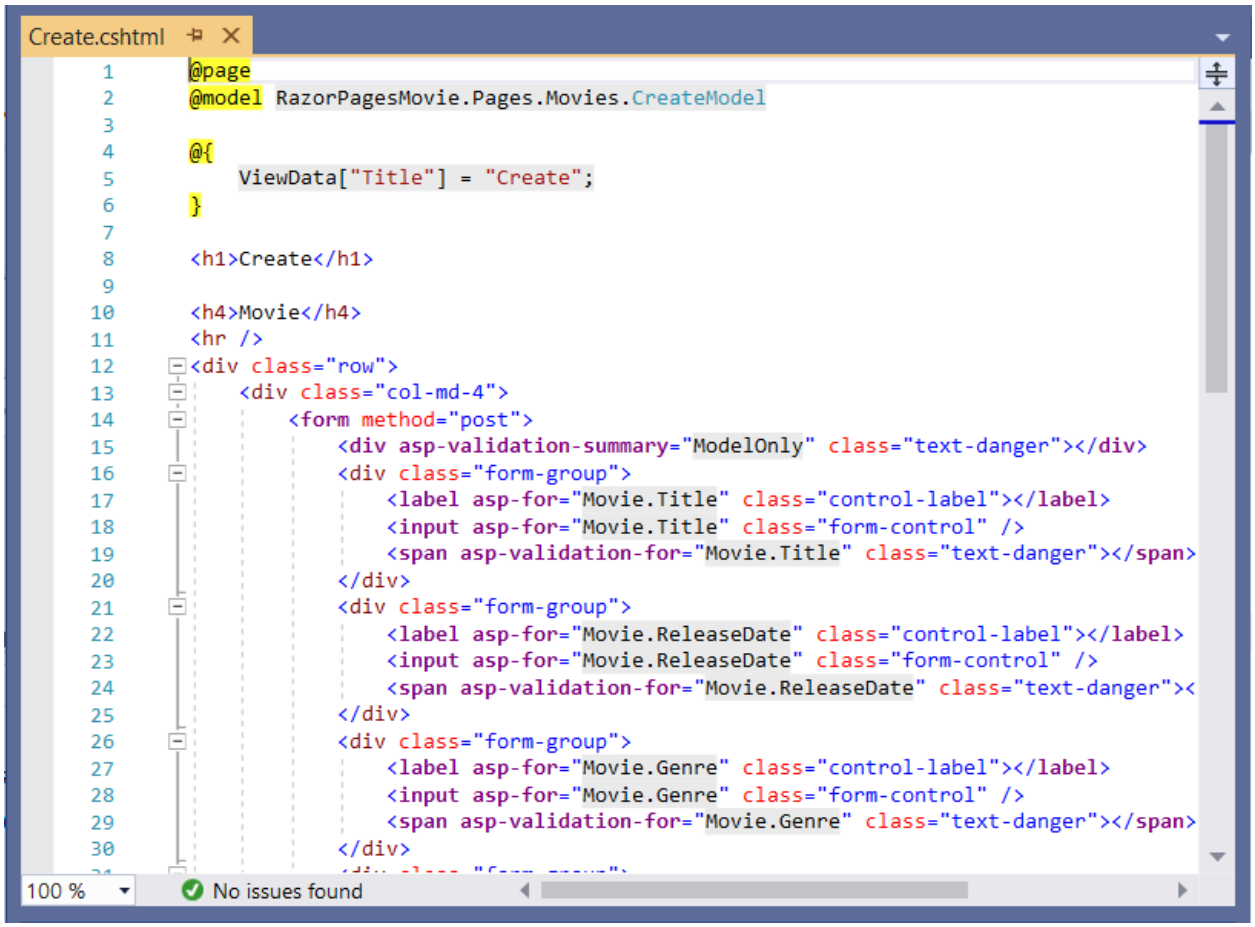
<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio displays the following tags in a distinctive bold font used for Tag Helpers:

- `<form method="post">`
- `<div asp-validation-summary="ModelOnly" class="text-danger"></div>`
- `<label asp-for="Movie.Title" class="control-label"></label>`
- `<input asp-for="Movie.Title" class="form-control" />`
- `<span asp-validation-for="Movie.Title" class="text-danger"></span>`



The `<form method="post">` element is a Form Tag Helper. The Form Tag Helper automatically includes an antiforgery token.

The scaffolding engine creates Razor markup for each field in the model (except the ID) similar to the following:

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

The Validation Tag Helpers (`<div asp-validation-summary` and `<span asp-validation-for`) display validation errors. Validation is covered in more detail later in this series.

The Label Tag Helper (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `for` attribute for the `Title` property.

The Input Tag Helper (`<input asp-for="Movie.Title" class="form-control">`) uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side.

For more information on Tag Helpers such as `<form method="post">`, see Tag Helpers in ASP.NET Core.

Additional resources

PREVIOUS: ADDING A MODEL
NEXT: DATABASE

By Rick Anderson

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial.

View or download sample.

The Create, Delete, Details, and Edit pages

Examine the *Pages/Movies/Index.cshtml.cs* Page Model:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Movies
{
    public class IndexModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public IndexModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IList<Movie> Movie { get;set; }

        public async Task OnGetAsync()
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor Pages are derived from `PageModel`. By convention, the `PageModel`-derived class is called `<PageName>Model`. The constructor uses dependency injection to add the `RazorPagesMovieContext` to the page. All the scaffolded pages follow this pattern. See Asynchronous code for more information on asynchronous programming with Entity Framework.

When a request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. `OnGetAsync` or `OnGet` is called on a Razor Page to initialize the state for the page. In this case, `OnGetAsync` gets a list of movies and displays them.

When `OnGet` returns `void` or `OnGetAsync` returns `Task`, no return method is used. When the return type is `IActionResult` or `Task<IActionResult>`, a return statement must be provided. For example, the *Pages/Movies/Create.cshtml.cs* `OnPostAsync` method:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Examine the *Pages/Movies/Index.cshtml* Razor Page:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup, otherwise it transitions into C#.

The `@page` Razor directive makes the file into an MVC action, which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` is an example of transitioning into Razor-specific markup. See Razor syntax for more information.

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movie[0].Title)
```

The `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` are `null` or empty. When the lambda expression is evaluated (for example, with `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

The @model directive

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel`-derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayFor` HTML Helpers on the page.

The layout page

Select the menu links (**RazorPagesMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Pages/Shared/_Layout.cshtml* file. Open the *Pages/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the page-specific views you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the **Pages/Privacy.cshtml** view is rendered inside the `RenderBody` method.

ViewData and layout

Consider the following code from the *Pages/Movies/Index.cshtml* file:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

The preceding highlighted code is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class has a `ViewData` dictionary property that can be used to add data that you want to pass to a View. You add objects into the `ViewData` dictionary using a key/value pattern. In the preceding sample, the "Title" property is added to the `ViewData` dictionary.

The "Title" property is used in the *Pages/Shared/_Layout.cshtml* file. The following markup shows the first few lines of the *_Layout.cshtml* file.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>

    @*Markup removed for brevity.*@
```

The line `@*Markup removed for brevity.*@` is a Razor comment which doesn't appear in your layout file. Unlike HTML comments (`<!-- -->`), Razor comments are not sent to the client.

Update the layout

Change the `<title>` element in the *Pages/Shared/_Layout.cshtml* file to display **Movie** rather than **RazorPagesMovie**.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie</title>
```

Find the following anchor element in the *Pages/Shared/_Layout.cshtml* file.

```
<a class="navbar-brand" asp-area="" asp-page="/Index">RazorPagesMovie</a>
```

Replace the preceding element with the following markup.

```
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

The preceding anchor element is a Tag Helper. In this case, it's the Anchor Tag Helper. The `asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page. The `asp-area` attribute value is empty, so the area isn't used in the link. See Areas for more information.

Save your changes, and test the app by clicking on the **RpMovie** link. See the _Layout.cshtml file in GitHub if you have any problems.

Test the other links (**Home**, **RpMovie**, **Create**, **Edit**, and **Delete**). Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark.

> **NOTE**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. This GitHub issue 4076 for instructions on adding decimal comma.

The `Layout` property is set in the *Pages/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The preceding markup sets the layout file to *Pages/Shared/_Layout.cshtml* for all Razor files under the *Pages*

folder. See Layout for more information.

The Create page model

Examine the *Pages/Movies/Create.cshtml.cs* page model:

```
// Unused usings removed.
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
using System;
using System.Threading.Tasks;

namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

        public CreateModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize, so `Page` is returned. Later in the tutorial you see `OnGet` method initialize state. The `Page` method creates a `PageResult` object that renders the *Create.cshtml* page.

The `Movie` property uses the `[BindProperty]` attribute to opt-in to model binding. When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date. Client-side validation and model validation are discussed later in the tutorial.

If there are no model errors, the data is saved, and the browser is redirected to the Index page.

The Create Razor Page

Examine the *Pages/Movies/Create.cshtml* Razor Page file:

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

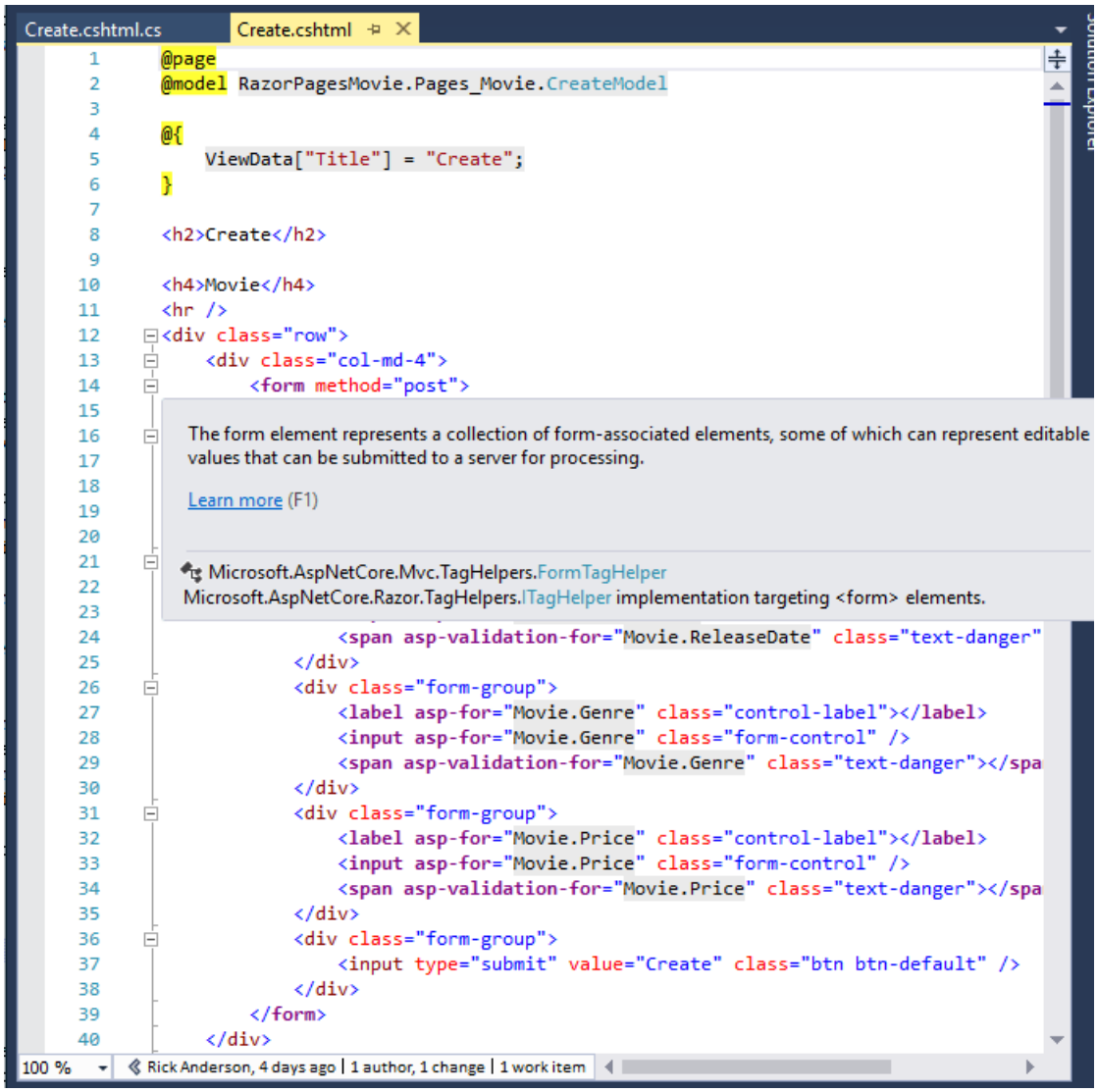
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio displays the `<form method="post">` tag in a distinctive bold font used for Tag Helpers:



The `<form method="post">` element is a Form Tag Helper. The Form Tag Helper automatically includes an antiforgery token.

The scaffolding engine creates Razor markup for each field in the model (except the ID) similar to the following:

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

The Validation Tag Helpers (`<div asp-validation-summary` and `<span asp-validation-for`) display validation errors. Validation is covered in more detail later in this series.

The Label Tag Helper (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `for` attribute for the `Title` property.

The Input Tag Helper (`<input asp-for="Movie.Title" class="form-control">`) uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side.

Additional resources

- YouTube version of this tutorial

PREVIOUS: ADDING A MODEL

NEXT: DATABASE

Part 4, with a database and ASP.NET Core

9/22/2020 • 12 minutes to read • Edit Online

By Rick Anderson and Joe Audette

View or download sample code (how to download).

View or download sample code (how to download).

The `RazorPagesMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in *Startup.cs*:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

The name value for the database (`Database={Database name}`) will be different for your generated code. The name value is arbitrary.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
bc;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

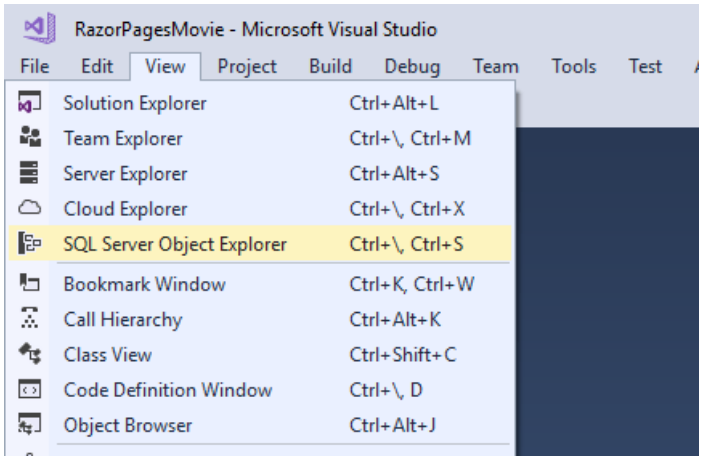
When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a real database server. See Configuration for more information.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

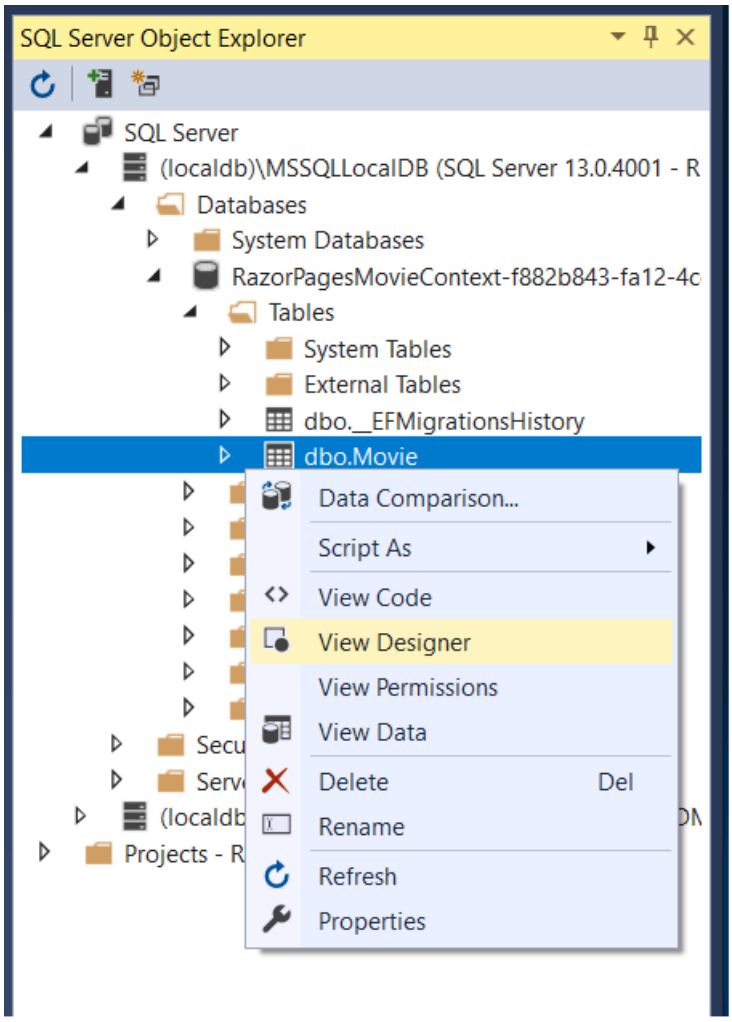
SQL Server Express LocalDB

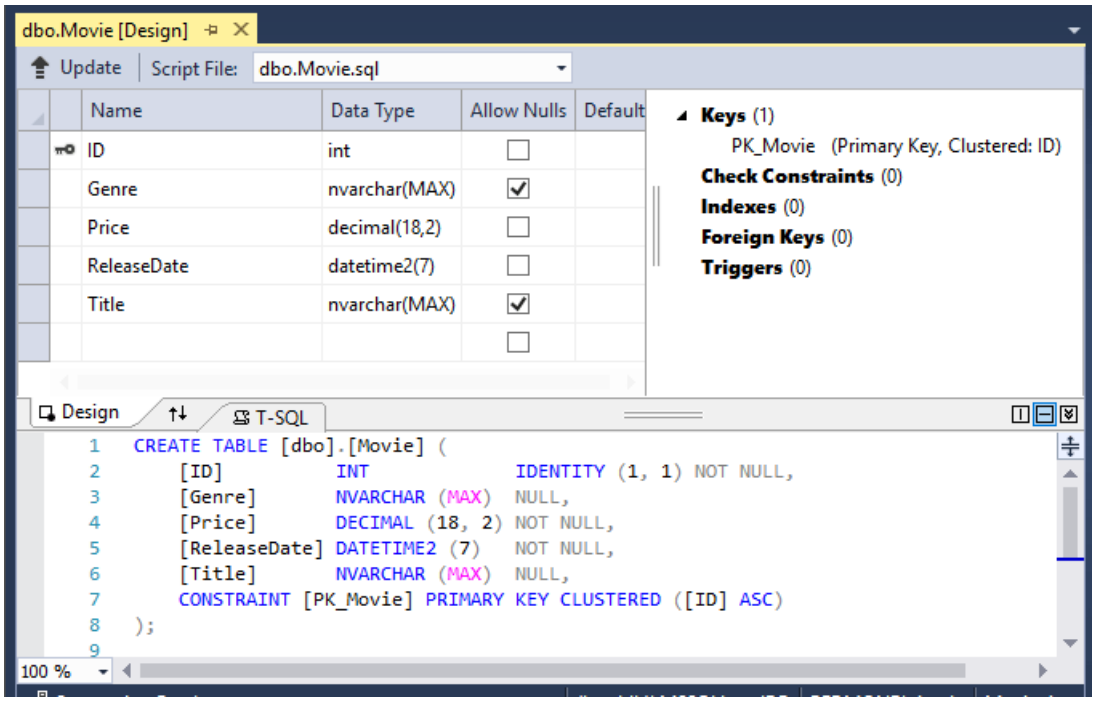
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:\Users\<user>\` directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



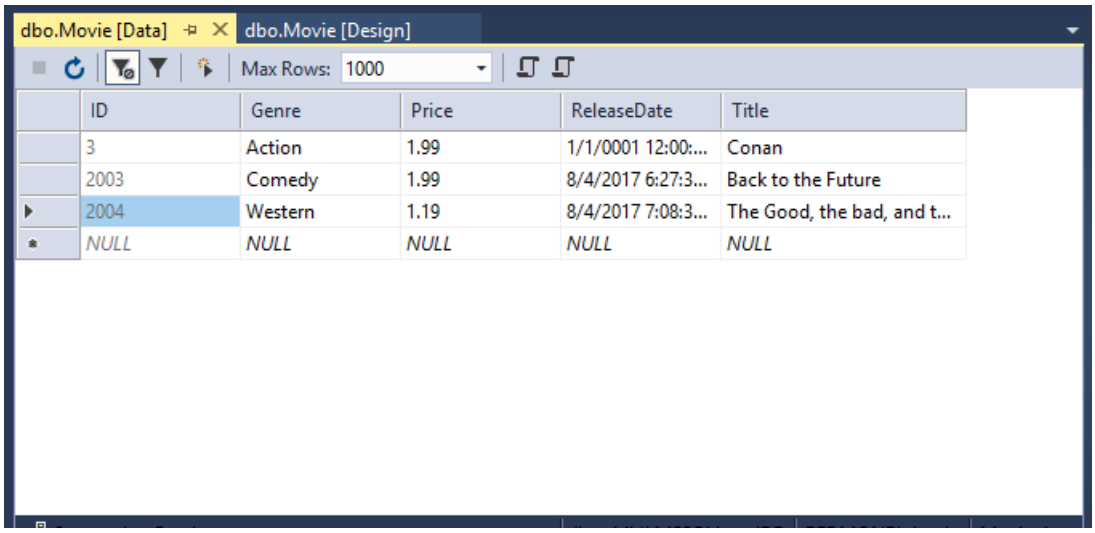
- Right click on the `Movie` table and select **View Designer**:





Note the key icon next to `ID`. By default, EF creates a property named `ID` for the primary key.

- Right click on the `Movie` table and select **View Data**:



Seed the database

Create a new class named `SeedData` in the *Models* folder with the following code:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new RazorPagesMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<RazorPagesMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

In *Program.cs*, modify the `Main` method to do the following:

- Get a DB context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method completes.

The following code shows the updated *Program.cs* file.

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();

        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

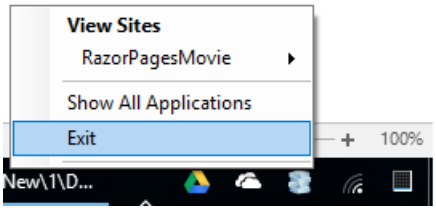
The following exception occurs when `Update-Database` has not been run:

```
SqlException: Cannot open database "RazorPagesMovieContext-" requested by the login. The login failed.
Login failed for user 'user name'.
```

Test the app

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:
  - Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**:



  - If you were running VS in non-debug mode, press F5 to run in debug mode.
  - If you were running VS in debug mode, stop the debugger and press F5.

The next tutorial will improve the presentation of the data.

Additional resources

PREVIOUS: SCAFFOLDED RAZOR PAGES

NEXT: UPDATING THE PAGES

View or download sample code (how to download).

View or download sample code (how to download).

The `RazorPagesMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in *Startup.cs*:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
// This method gets called by the runtime.
// Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is
        // needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<RazorPagesMovieContext>(options =>
      options.UseSqlServer(
          Configuration.GetConnectionString("RazorPagesMovieContext")));
}
```

For more information on the methods used in `ConfigureServices`, see:

- EU General Data Protection Regulation (GDPR) support in ASP.NET Core for `CookiePolicyOptions`.
- SetCompatibilityVersion

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

The name value for the database (`Database={Database name}`) will be different for your generated code. The name value is arbitrary.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
1234;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

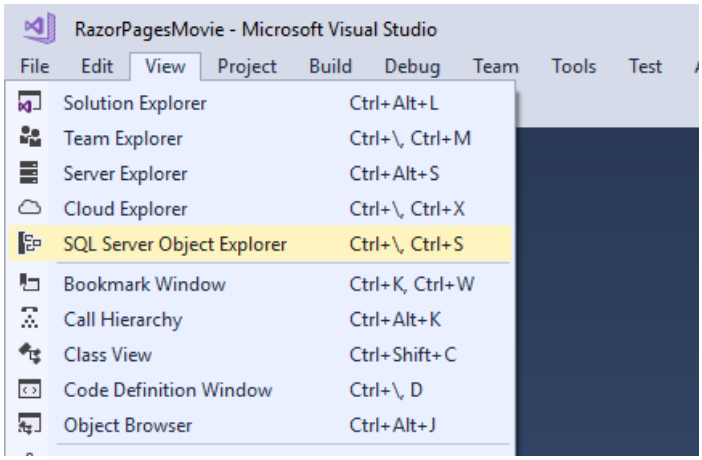
When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a real database server. See Configuration for more information.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

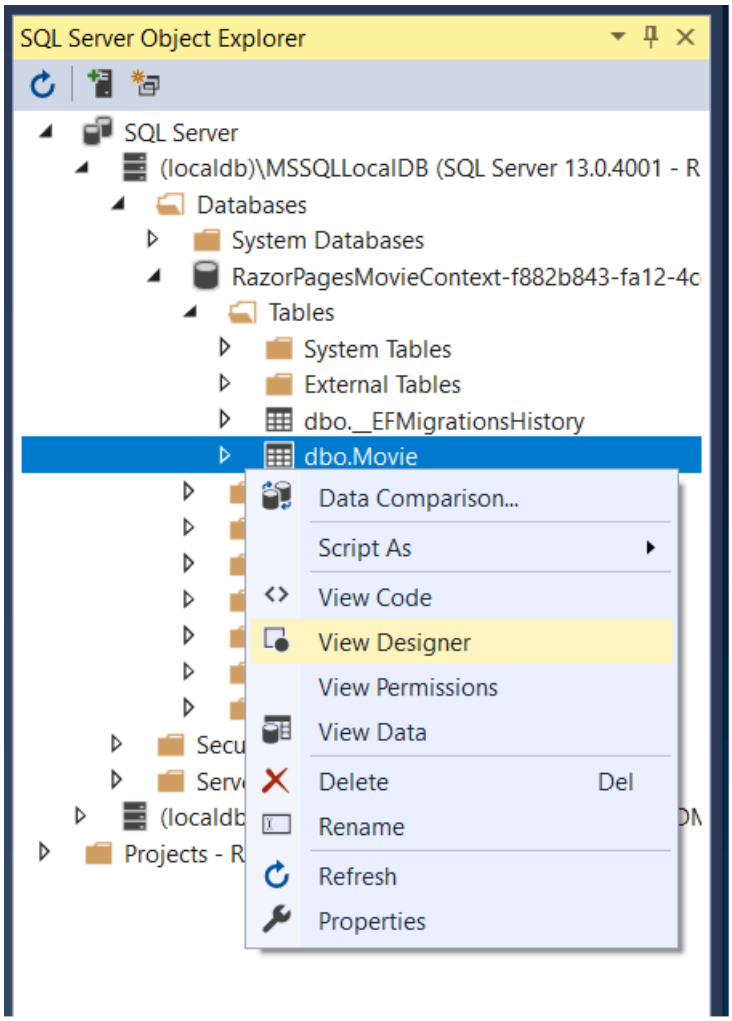
SQL Server Express LocalDB

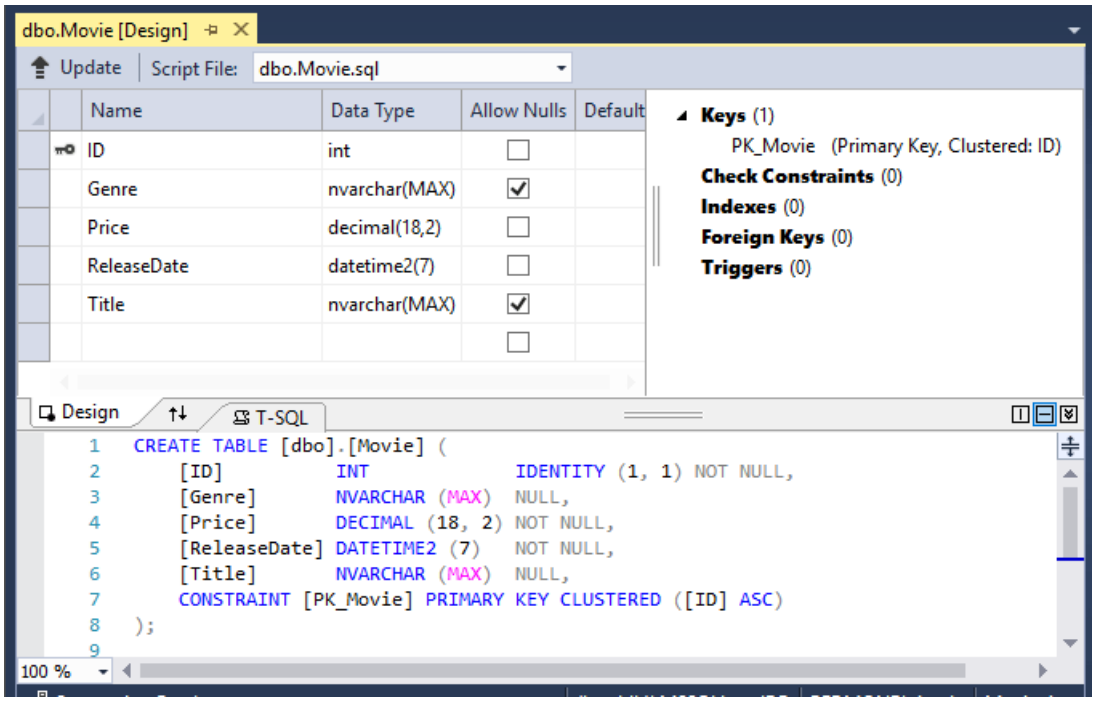
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:/Users/<user/>` directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



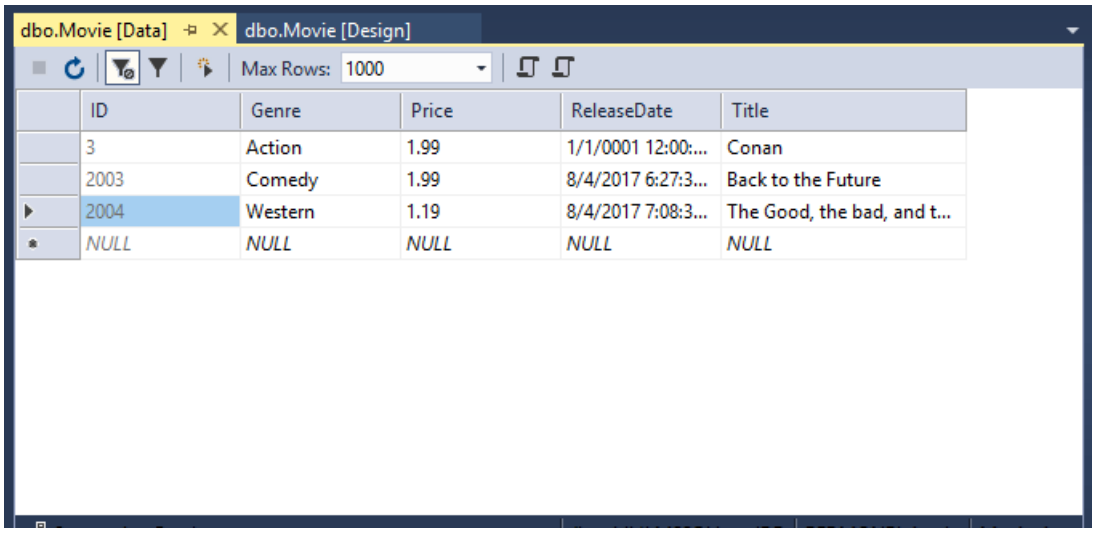
- Right click on the `Movie` table and select **View Designer**:





Note the key icon next to `ID`. By default, EF creates a property named `ID` for the primary key.

- Right click on the `Movie` table and select **View Data**:



Seed the database

Create a new class named `SeedData` in the *Models* folder with the following code:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace RazorPagesMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new RazorPagesMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<RazorPagesMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

In *Program.cs*, modify the `Main` method to do the following:

- Get a DB context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method completes.

The following code shows the updated *Program.cs* file.

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using RazorPagesMovie.Models;
using System;
using Microsoft.EntityFrameworkCore;

namespace RazorPagesMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context=services.
                        GetRequiredService<RazorPagesMovieContext>();
                    context.Database.Migrate();
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

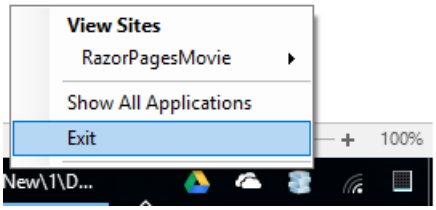
A production app would not call `Database.Migrate`. It's added to the preceding code to prevent the following exception when `Update-Database` has not been run:

SqlException: Cannot open database "RazorPagesMovieContext-21" requested by the login. The login failed.
Login failed for user 'user name'.

Test the app

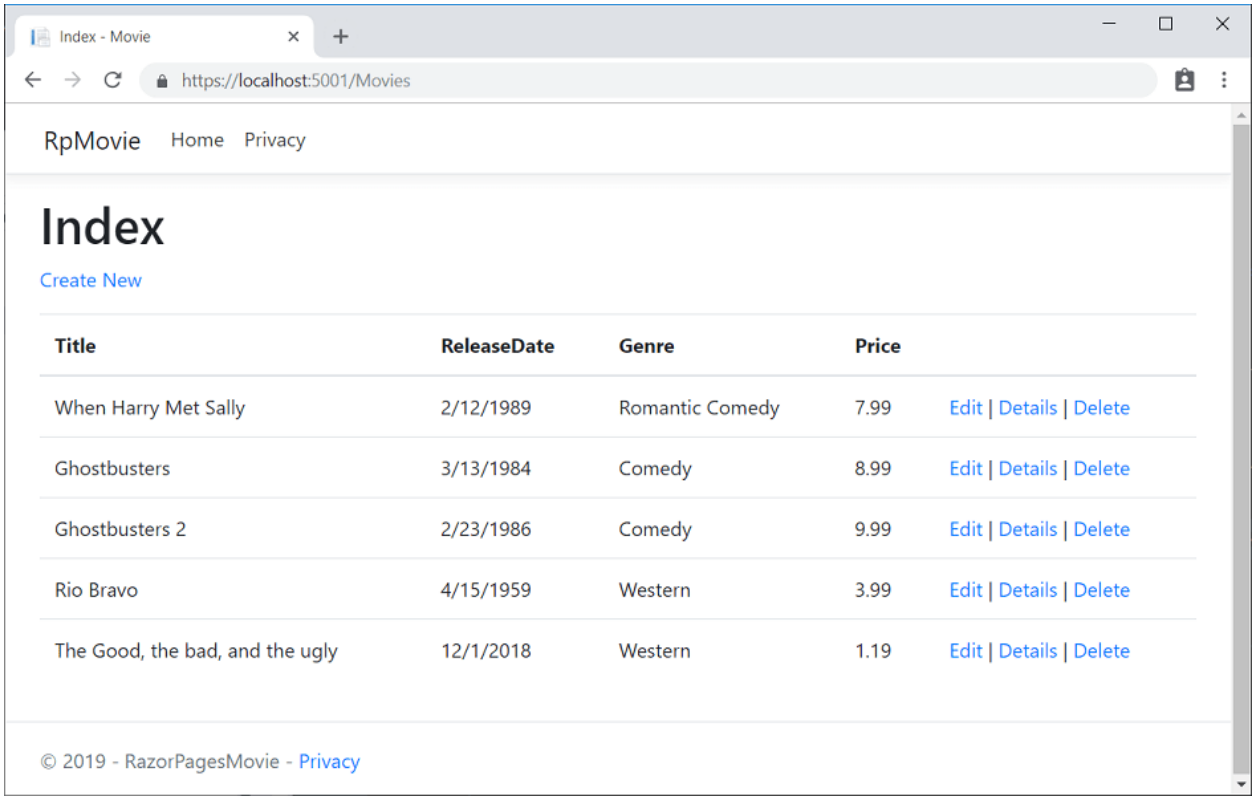
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:
  - Right-click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**:



  - If you were running VS in non-debug mode, press F5 to run in debug mode.
  - If you were running VS in debug mode, stop the debugger and press F5.

The app shows the seeded data:



The next tutorial will clean up the presentation of the data.

Additional resources

- YouTube version of this tutorial

PREVIOUS: SCAFFOLDED RAZOR PAGES

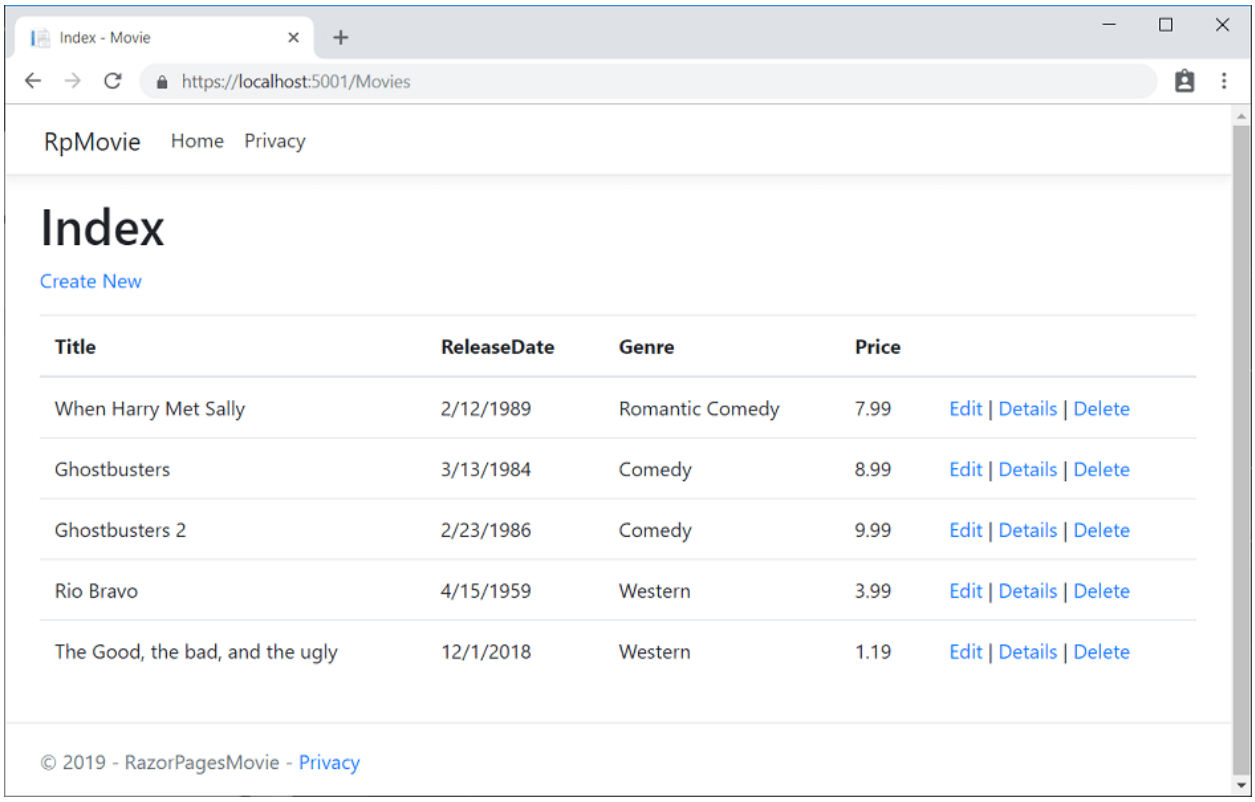
NEXT: UPDATING THE PAGES

Part 5, update the generated pages in an ASP.NET Core app

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

The scaffolded movie app has a good start, but the presentation isn't ideal. **ReleaseDate** should be **Release Date** (two words).



Update the generated code

Open the *Models/Movie.cs* file and add the highlighted lines shown in the following code:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

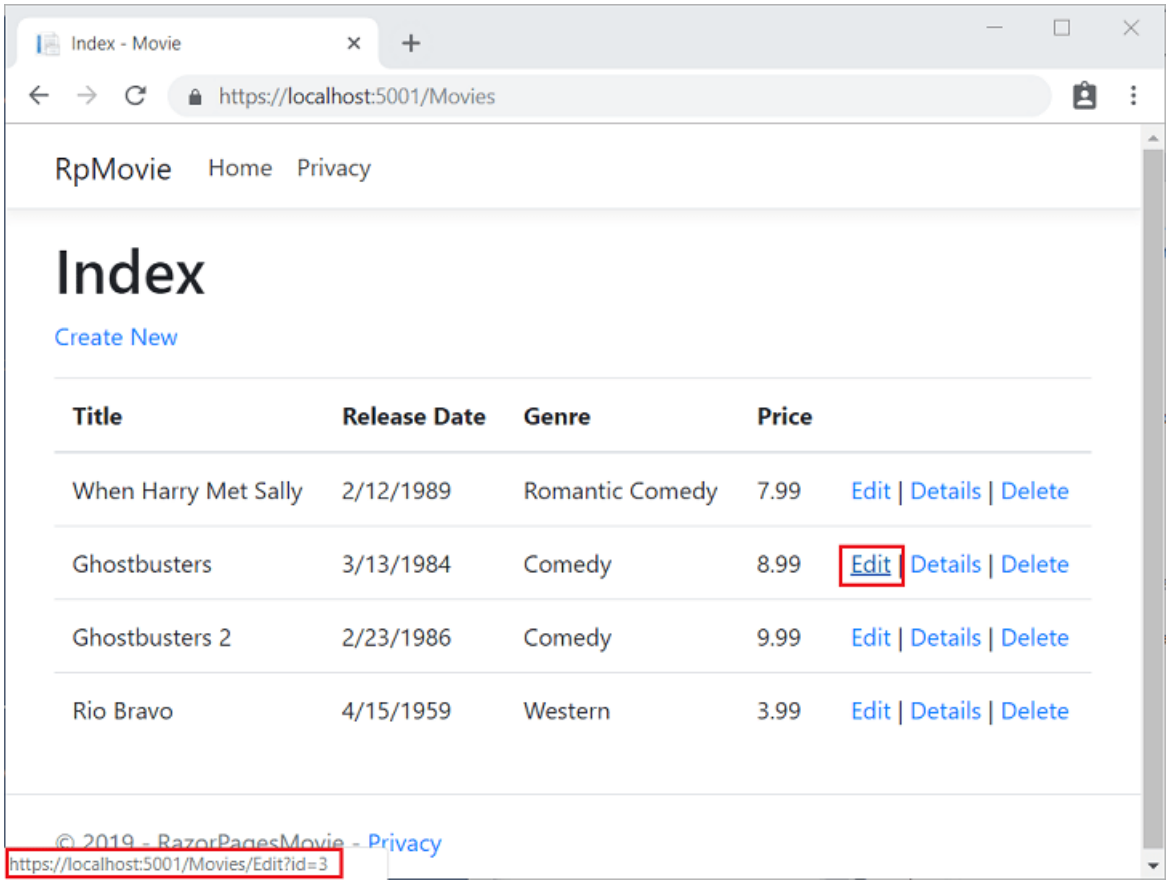
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

The `[Column(TypeName = "decimal(18, 2)")]` data annotation enables Entity Framework Core to correctly map `Price` to currency in the database. For more information, see Data Types.

DataAnnotations is covered in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

Browse to Pages/Movies and hover over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the *Pages/Movies/Index.cshtml* file.

```
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the preceding code, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route id (`asp-route-id`). See URL generation for Pages for more information.

Use **View Source** from your favorite browser to examine the generated markup. A portion of the generated HTML is shown below:

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically-generated links pass the movie ID with a query string (for example, the `?id=1` in `https://localhost:5001/Movies/Details?id=1`).

Add route template

Update the Edit, Details, and Delete Razor Pages to use the "{id:int}" route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`. Run the app and then view source. The generated HTML adds the ID to the path portion of the URL:

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the "{id:int}" route template that does **not** include the integer will return an HTTP 404 (not found) error. For example, `http://localhost:5000/Movies/Details` will return a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

To test the behavior of `@page "{id:int?}"`:

- Set the page directive in *Pages/Movies/Details.cshtml* to `@page "{id:int?}"`.
- Set a break point in `public async Task<IActionResult> OnGetAsync(int? id)` (in *Pages/Movies/Details.cshtml.cs*).
- Navigate to `https://localhost:5001/Movies/Details/`.

With the `@page "{id:int}"` directive, the break point is never hit. The routing engine returns HTTP 404. Using `@page "{id:int?}"`, the `OnGetAsync` method returns `NotFound` (HTTP 404).

Review concurrency exception handling

Review the `OnPostAsync` method in the *Pages/Movies/Edit.cshtml.cs* file:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.ID == id);
}
```

The previous code detects concurrency exceptions when the one client deletes the movie and the other client posts changes to the movie.

To test the `catch` block:

- Set a breakpoint on `catch (DbUpdateConcurrencyException)`
- Select **Edit** for a movie, make changes, but don't enter **Save**.
- In another browser window, select the **Delete** link for the same movie, and then delete the movie.
- In the previous browser window, post changes to the movie.

Production code may want to detect concurrency conflicts. See Handle concurrency conflicts for more information.

Posting and binding review

Examine the *Pages/Movies/Edit.cshtml.cs* file:

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Movie = await _context.Movie.FirstOrDefaultAsync(m => m.ID == id);

        if (Movie == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(Movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return RedirectToPage("./Index");
    }

    private bool MovieExists(int id)
    {
        return _context.Movie.Any(e => e.ID == id);
    }
```

When an HTTP GET request is made to the Movies/Edit page (for example, `http://localhost:5000/Movies/Edit/2`):

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the *Pages/Movies/Edit.cshtml* Razor Page. The *Pages/Movies/Edit.cshtml* file

contains the model directive (`@model RazorPagesMovie.Pages.Movies.EditModel`), which makes the movie model available on the page.

- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables Model binding.

```
[BindProperty]
public Movie Movie { get; set; }
```

- If there are errors in the model state (for example, `ReleaseDate` cannot be converted to a date), the form is redisplayed with the submitted values.
- If there are no model errors, the movie is saved.

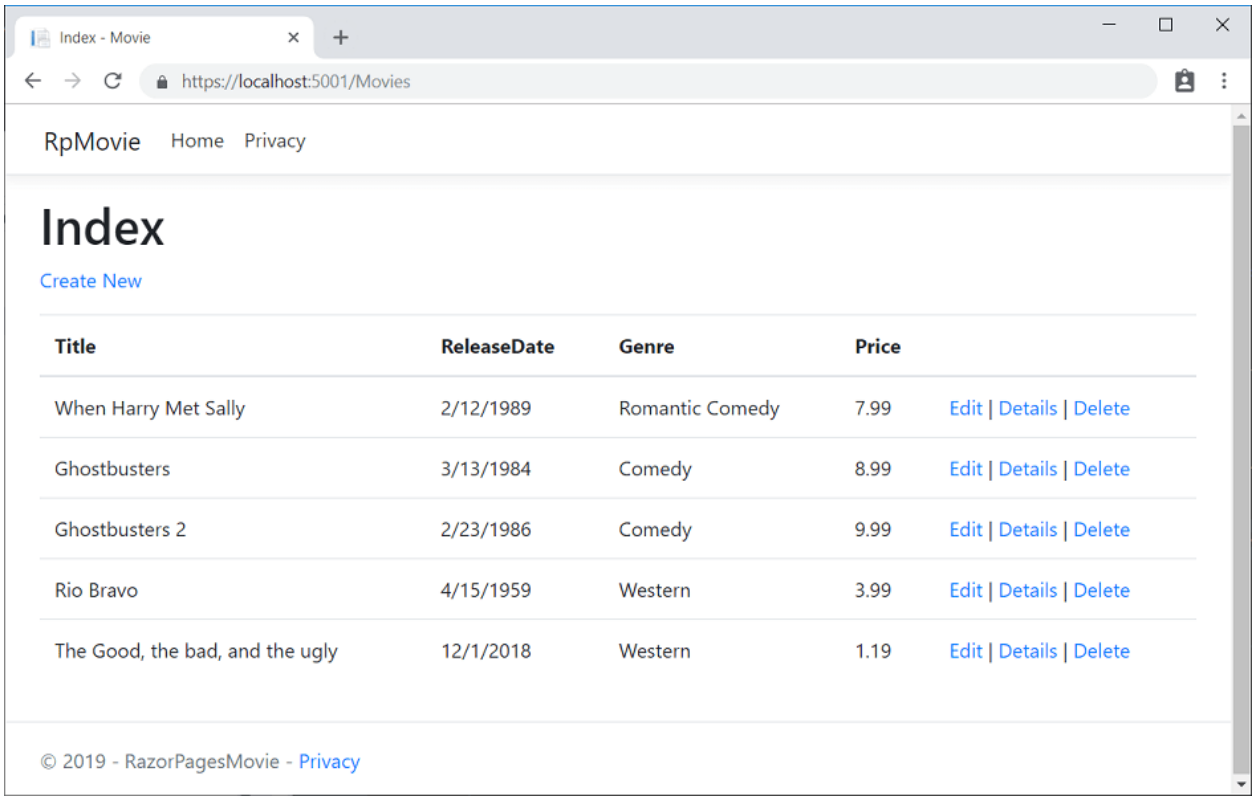
The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Additional resources

PREVIOUS: WORKING WITH A DATABASE

NEXT: ADD SEARCH

The scaffolded movie app has a good start, but the presentation isn't ideal. **ReleaseDate** should be **Release Date** (two words).



Update the generated code

Open the *Models/Movie.cs* file and add the highlighted lines shown in the following code:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

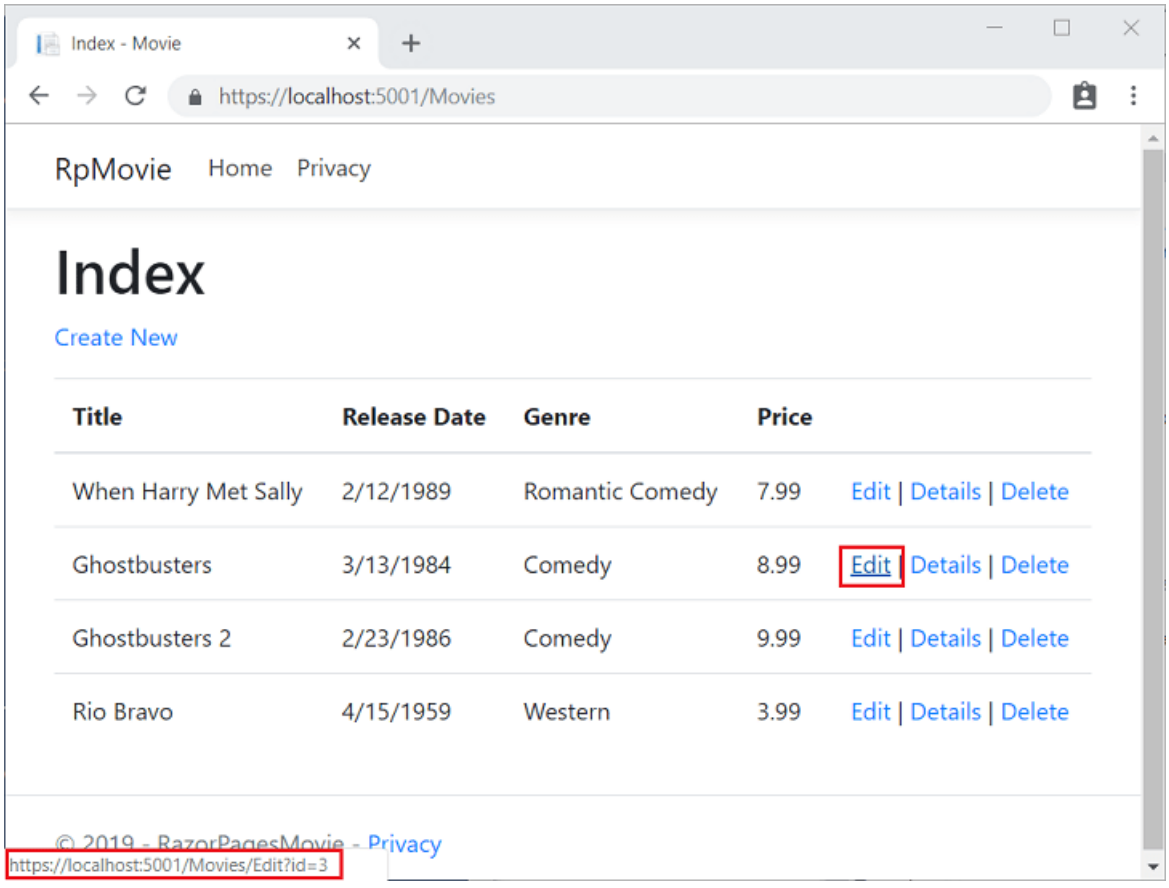
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

The `[Column(TypeName = "decimal(18, 2)")]` data annotation enables Entity Framework Core to correctly map `Price` to currency in the database. For more information, see Data Types.

DataAnnotations is covered in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

Browse to Pages/Movies and hover over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the *Pages/Movies/Index.cshtml* file.

```
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the preceding code, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route id (`asp-route-id`). See URL generation for Pages for more information.

Use **View Source** from your favorite browser to examine the generated markup. A portion of the generated HTML is shown below:

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically-generated links pass the movie ID with a query string (for example, the `?id=1` in `https://localhost:5001/Movies/Details?id=1`).

Update the Edit, Details, and Delete Razor Pages to use the "{id:int}" route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"` . Run the app and then view source. The generated HTML adds the ID to the path portion of the URL:

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the "{id:int}" route template that does **not** include the integer will return an HTTP 404 (not found) error. For example, `http://localhost:5000/Movies/Details` will return a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

To test the behavior of `@page "{id:int?}"` :

- Set the page directive in *Pages/Movies/Details.cshtml* to `@page "{id:int?}"`.
- Set a break point in `public async Task<IActionResult> OnGetAsync(int? id)` (in *Pages/Movies/Details.cshtml.cs*).
- Navigate to `https://localhost:5001/Movies/Details/`.

With the `@page "{id:int}"` directive, the break point is never hit. The routing engine returns HTTP 404. Using `@page "{id:int?}"`, the `OnGetAsync` method returns `NotFound` (HTTP 404).

Review concurrency exception handling

Review the `OnPostAsync` method in the *Pages/Movies/Edit.cshtml.cs* file:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.ID))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
    return _context.Movie.Any(e => e.ID == id);
}
```

The previous code detects concurrency exceptions when the one client deletes the movie and the other client posts changes to the movie.

To test the `catch` block:

- Set a breakpoint on `catch (DbUpdateConcurrencyException)`
- Select **Edit** for a movie, make changes, but don't enter **Save**.
- In another browser window, select the **Delete** link for the same movie, and then delete the movie.
- In the previous browser window, post changes to the movie.

Production code may want to detect concurrency conflicts. See Handle concurrency conflicts for more information.

Posting and binding review

Examine the *Pages/Movies/Edit.cshtml.cs* file:

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);

        if (Movie == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Attach(Movie).State = EntityState.Modified;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!_context.Movie.Any(e => e.ID == Movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }

        return RedirectToPage("./Index");
    }
}
```

When an HTTP GET request is made to the Movies/Edit page (for example, `http://localhost:5000/Movies/Edit/2`):

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the *Pages/Movies/Edit.cshtml* Razor Page. The *Pages/Movies/Edit.cshtml* file contains the model directive (`@model RazorPagesMovie.Pages.Movies.EditModel`), which makes the movie model available on the page.
- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables Model binding.

```
[BindProperty]
public Movie Movie { get; set; }
```

- If there are errors in the model state (for example, `ReleaseDate` cannot be converted to a date), the form is displayed with the submitted values.
- If there are no model errors, the movie is saved.

The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Search is added in the next tutorial.

Additional resources

- YouTube version of this tutorial

PREVIOUS: WORKING WITH A DATABASE

NEXT: ADD SEARCH

Part 6, add search to ASP.NET Core Razor Pages

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

View or download sample code (how to download).

View or download sample code (how to download).

In the following sections, searching movies by *genre* or *name* is added.

Add the following highlighted properties to *Pages/Movies/Index.cshtml.cs*:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    [BindProperty(SupportsGet = true)]
    public string SearchString { get; set; }
    // Requires using Microsoft.AspNetCore.Mvc.Rendering;
    public SelectList Genres { get; set; }
    [BindProperty(SupportsGet = true)]
    public string MovieGenre { get; set; }
```

- `SearchString`: contains the text users enter in the search text box. `SearchString` has the `[BindProperty]` attribute. `[BindProperty]` binds form values and query strings with the same name as the property. `(SupportsGet = true)` is required for binding on GET requests.
- `Genres`: contains the list of genres. `Genres` allows the user to select a genre from the list. `SelectList` requires `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`: contains the specific genre the user selects (for example, "Western").
- `Genres` and `MovieGenre` are used later in this tutorial.

WARNING

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the `[BindProperty]` attribute's `SupportsGet` property to `true`:

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Update the Index page's `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a LINQ query to select the movies:

```
// using System.Linq;
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

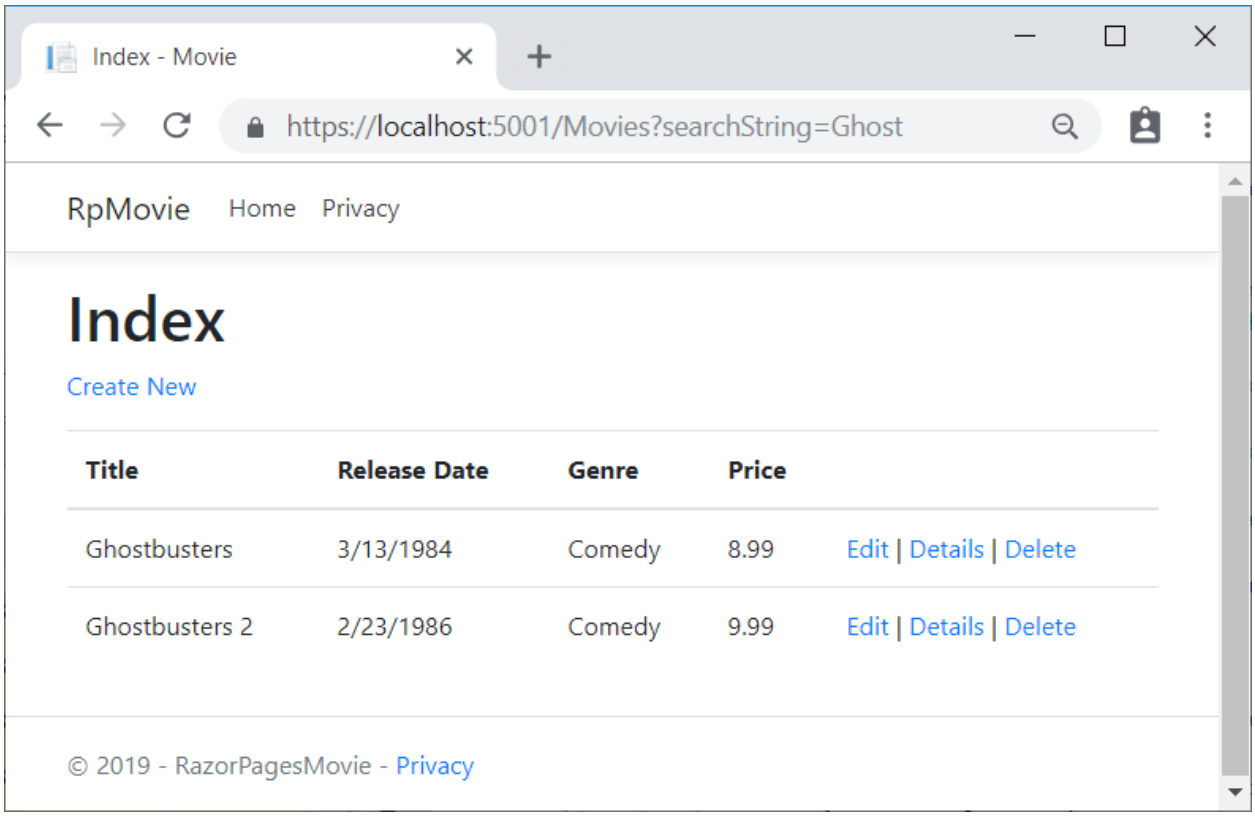
If the `SearchString` property is not null or empty, the movies query is modified to filter on the search string:

```
if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}
```

The `s => s.Title.Contains()` code is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the preceding code). LINQ queries are not executed when they're defined or when they're modified by calling a method (such as `Where`, `Contains` or `OrderBy`). Rather, query execution is deferred. That means the evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See Query Execution for more information.

> **NOTE**
>
> The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

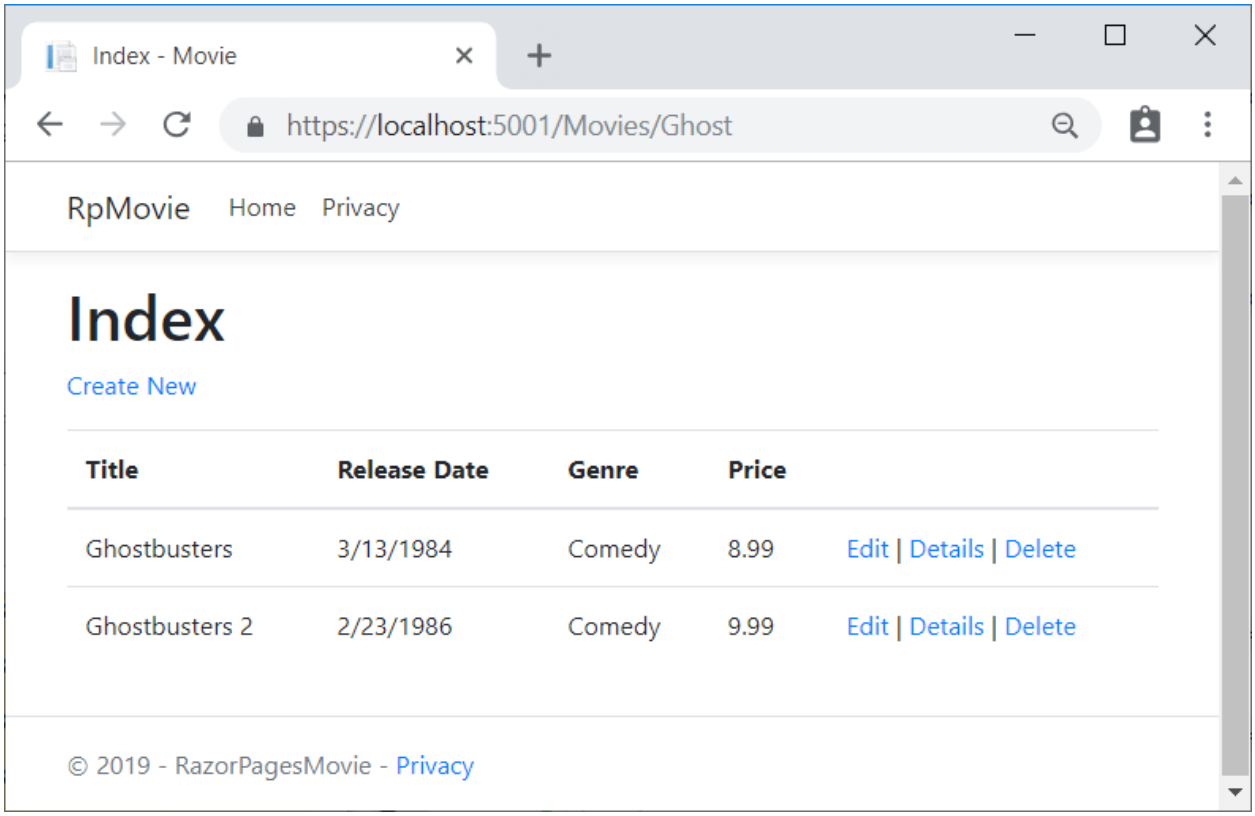
Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL (for example, `https://localhost:5001/Movies?searchString=Ghost`). The filtered movies are displayed.



If the following route template is added to the Index page, the search string can be passed as a URL segment (for example, `https://localhost:5001/Movies/Ghost`).

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



The ASP.NET Core runtime uses model binding to set the value of the `SearchString` property from the query string (`?searchString=Ghost`) or route data (`https://localhost:5001/Movies/Ghost`). Model binding is not case

sensitive.

However, you can't expect users to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the *Pages/Movies/Index.cshtml* file, and add the `<form>` markup highlighted in the following code:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

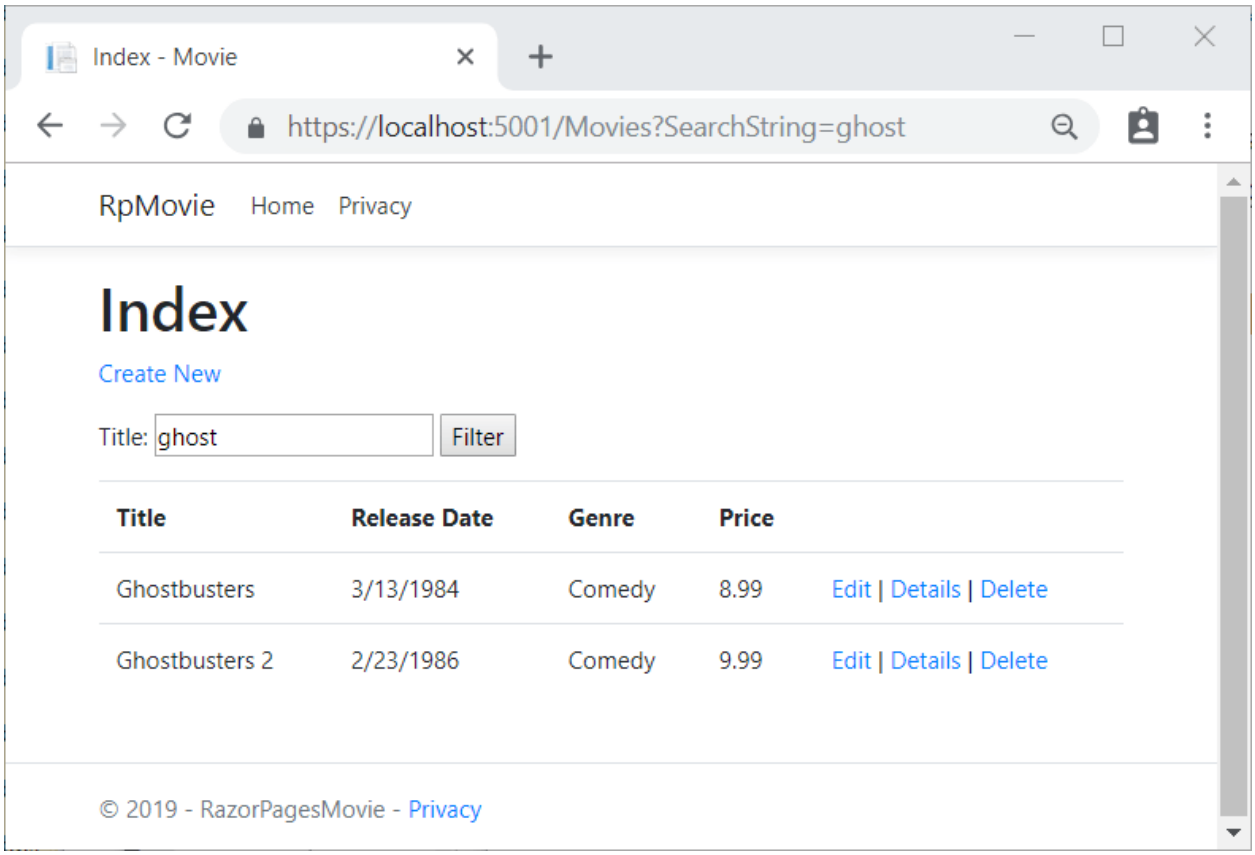
<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

The HTML `<form>` tag uses the following Tag Helpers:

- Form Tag Helper. When the form is submitted, the filter string is sent to the *Pages/Movies/Index* page via query string.
- Input Tag Helper

Save the changes and test the filter.



Search by genre

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

The following code is a LINQ query that retrieves all the genres from the database.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres.

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Add search by genre to the Razor Page

Update *Index.cshtml* as follows:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

Test the app by searching by genre, by movie title, and by both.

Additional resources

- YouTube version of this tutorial

PREVIOUS: UPDATING THE PAGES

NEXT: ADDING A NEW FIELD

View or download sample code (how to download).

View or download sample code (how to download).

In the following sections, searching movies by *genre* or *name* is added.

Add the following highlighted properties to *Pages/Movies/Index.cshtml.cs*:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Models.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Models.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get; set; }
    [BindProperty(SupportsGet = true)]
    public string SearchString { get; set; }
    // Requires using Microsoft.AspNetCore.Mvc.Rendering;
    public SelectList Genres { get; set; }
    [BindProperty(SupportsGet = true)]
    public string MovieGenre { get; set; }
```

- `SearchString`: contains the text users enter in the search text box. `SearchString` has the `[BindProperty]` attribute. `[BindProperty]` binds form values and query strings with the same name as the property. `(SupportsGet = true)` is required for binding on GET requests.
- `Genres`: contains the list of genres. `Genres` allows the user to select a genre from the list. `SelectList` requires `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`: contains the specific genre the user selects (for example, "Western").
- `Genres` and `MovieGenre` are used later in this tutorial.

WARNING

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the `[BindProperty]` attribute's `SupportsGet` property to `true`:

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Update the Index page's `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a LINQ query to select the movies:

```
// using System.Linq;
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

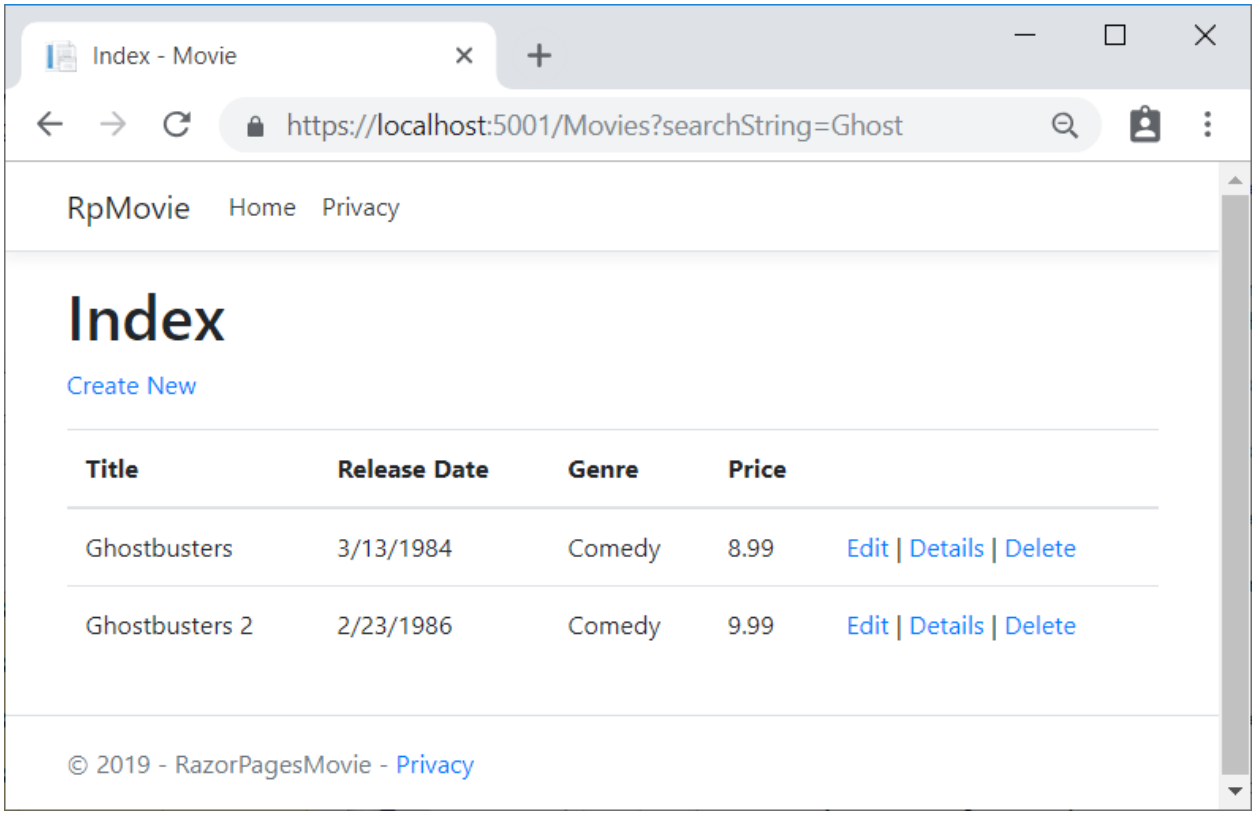
If the `SearchString` property is not null or empty, the movies query is modified to filter on the search string:

```
if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}
```

The `s => s.Title.Contains()` code is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the preceding code). LINQ queries are not executed when they're defined or when they're modified by calling a method (such as `Where`, `Contains` or `OrderBy`). Rather, query execution is deferred. That means the evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See Query Execution for more information.

Note: The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

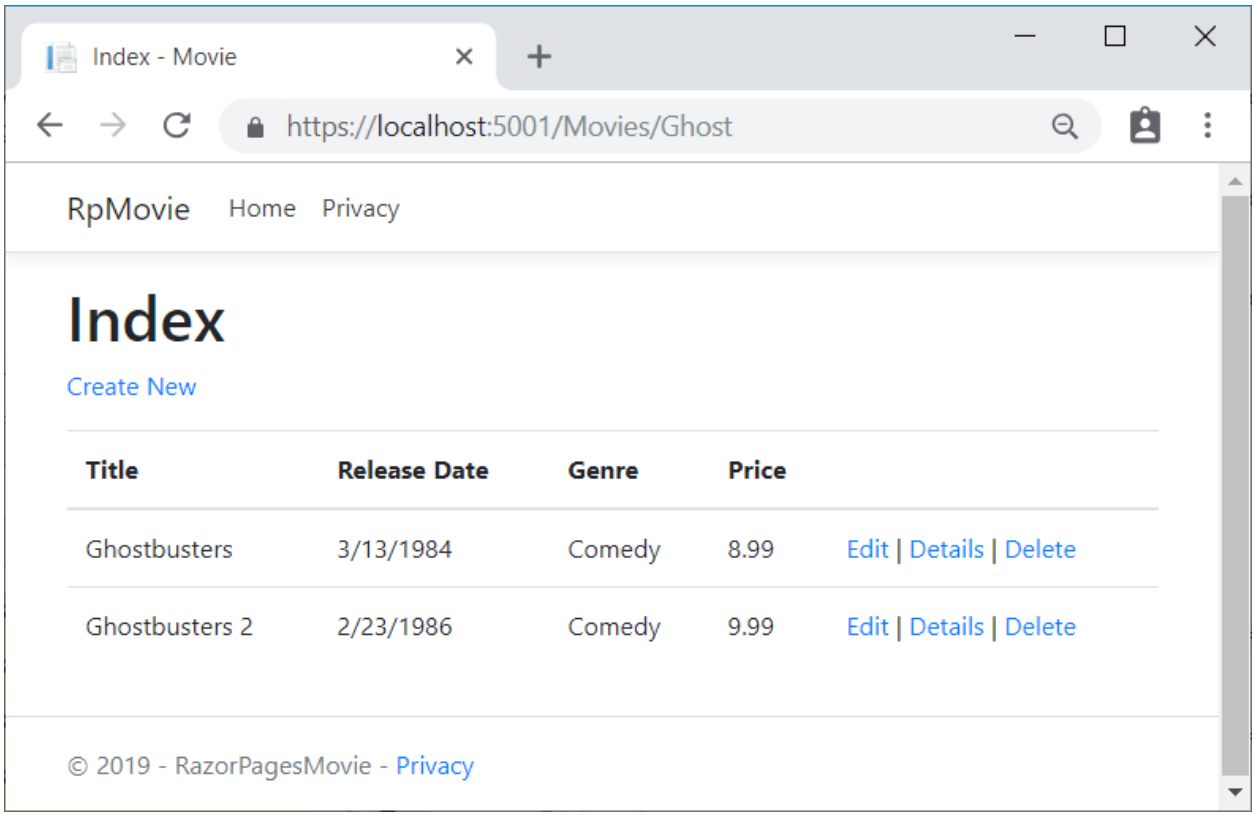
Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL (for example, `https://localhost:5001/Movies?searchString=Ghost`). The filtered movies are displayed.



If the following route template is added to the Index page, the search string can be passed as a URL segment (for example, `https://localhost:5001/Movies/Ghost`).

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



The ASP.NET Core runtime uses model binding to set the value of the `SearchString` property from the query string (`?searchString=Ghost`) or route data (`https://localhost:5001/Movies/Ghost`). Model binding is not case sensitive.

However, you can't expect users to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the *Pages/Movies/Index.cshtml* file, and add the `<form>` markup highlighted in the following code:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

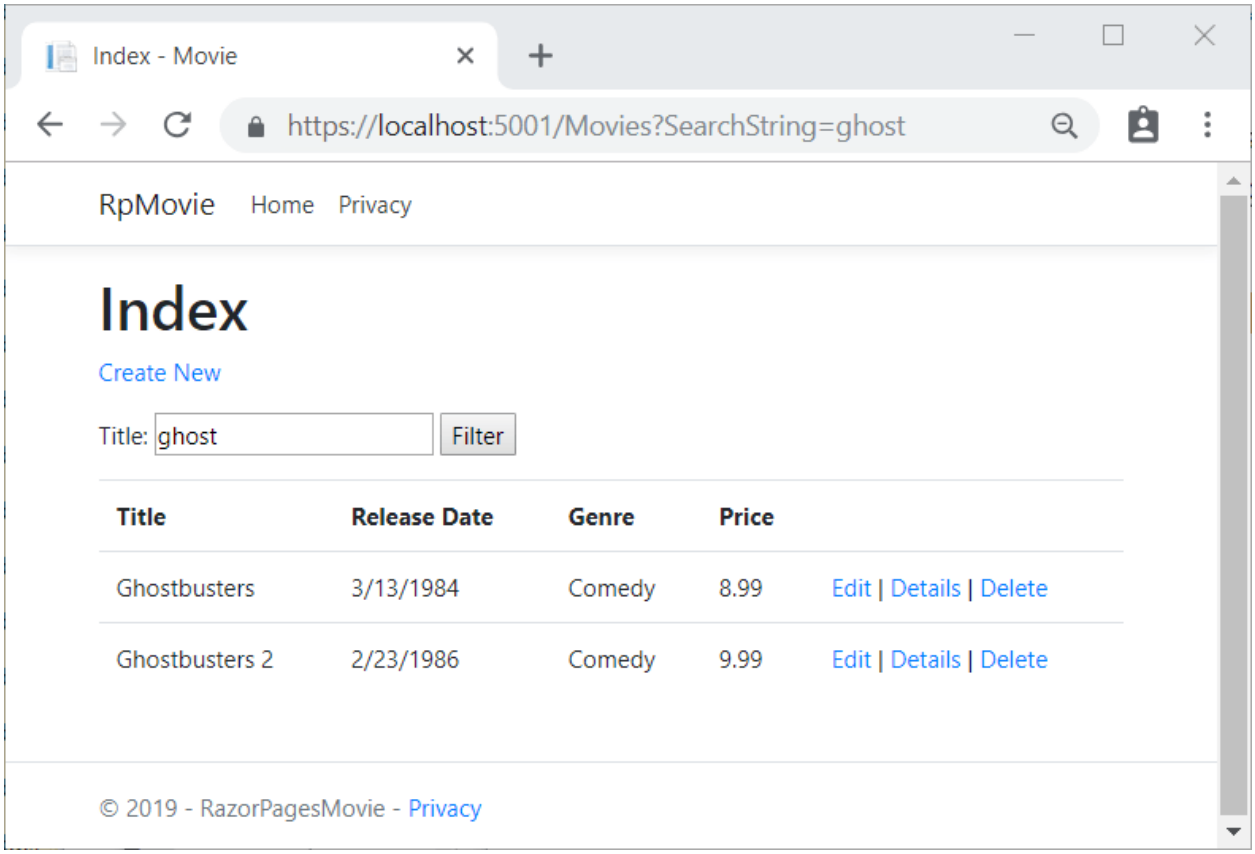
<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

The HTML `<form>` tag uses the following Tag Helpers:

- Form Tag Helper. When the form is submitted, the filter string is sent to the *Pages/Movies/Index* page via query string.
- Input Tag Helper

Save the changes and test the filter.



Search by genre

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

The following code is a LINQ query that retrieves all the genres from the database.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres.

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Add search by genre to the Razor Page

Update *Index.cshtml* as follows:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    @*Markup removed for brevity.*@
```

Test the app by searching by genre, by movie title, and by both. The preceding code uses the Select Tag Helper and Option Tag Helper.

Additional resources

- YouTube version of this tutorial

PREVIOUS: UPDATING THE PAGES

NEXT: ADDING A NEW FIELD

Part 7, add a new field to a Razor Page in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

View or download sample code (how to download).

View or download sample code (how to download).

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field schema change to the database.

When using EF Code First to automatically create a database, Code First:

- Adds an `__EFMigrationsHistory` table to the database to track whether the schema of the database is in sync with the model classes it was generated from.
- If the model classes aren't in sync with the DB, EF throws an exception.

Automatic verification of schema/model in sync makes it easier to find inconsistent database/code issues.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app.

Edit *Pages/Movies/Index.cshtml*, and add a `Rating` field:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
```

```
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the following pages:

- Add the `Rating` field to the Delete and Details pages.

- Update Create.cshtml with a `Rating` field.
- Add the `Rating` field to the Edit Page.

The app won't work until the DB is updated to include the new field. Running the app without updating the database throws a `SqlException`:

```
SqlException: Invalid column name 'Rating'.
```

The `SqlException` exception is caused by the updated Movie model class being different than the schema of the Movie table of the database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle; it allows you to quickly evolve the model and database schema together. The downside is that you lose existing data in the database. Don't use this approach on a production database! Dropping the DB on schema changes and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie` block.

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Price = 7.99M,
        Rating = "R"
    },
```

See the completed SeedData.cs file.

Build the solution.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add a migration for the rating field

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**. In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` DB schema.
- Create code to migrate the DB schema to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

The `Update-Database` command tells the framework to apply the schema changes to the database and to preserve existing data.

If you delete all the records in the DB, the initializer will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from Sql Server Object Explorer (SSOX).

Another option is to delete the database and use migrations to re-create the database. To delete the database in SSOX:

- Select the database in SSOX.
- Right click on the database, and select *Delete*.
- Check **Close existing connections**.
- Select **OK**.
- In the PMC, update the database:

```
Update-Database
```

Run the app and verify you can create/edit/display movies with a `Rating` field. If the database isn't seeded, set a break point in the `SeedData.Initialize` method.

Additional resources

- YouTube version of this tutorial

PREVIOUS: ADDING SEARCH

NEXT: ADDING VALIDATION

View or download sample code (how to download).

View or download sample code (how to download).

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field schema change to the database.

When using EF Code First to automatically create a database, Code First:

- Adds a table to the database to track whether the schema of the database is in sync with the model classes it was generated from.
- If the model classes aren't in sync with the DB, EF throws an exception.

Automatic verification of schema/model in sync makes it easier to find inconsistent database/code issues.

Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app.

Edit *Pages/Movies/Index.cshtml*, and add a `Rating` field:

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
```

```
        {
        <tr><td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Rating)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
        }
    </tbody>
</table>
```

Update the following pages:

- Add the `Rating` field to the Delete and Details pages.
- Update Create.cshtml with a `Rating` field.
- Add the `Rating` field to the Edit Page.

The app won't work until the DB is updated to include the new field. If run now, the app throws a `SqlException`:

```
SqlException: Invalid column name 'Rating'.
```

This error is caused by the updated Movie model class being different than the schema of the Movie table of the database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle; it allows you to quickly evolve the model and database schema together. The downside is that you lose existing data in the database. Don't use this approach on a production database! Dropping the DB on schema changes and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie` block.

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Price = 7.99M,
        Rating = "R"
    },
```

See the completed SeedData.cs file.

Build the solution.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add a migration for the rating field

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` DB schema.
- Create code to migrate the DB schema to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

The `Update-Database` command tells the framework to apply the schema changes to the database.

If you delete all the records in the DB, the initializer will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from Sql Server Object Explorer (SSOX).

Another option is to delete the database and use migrations to re-create the database. To delete the database in SSOX:

- Select the database in SSOX.
- Right click on the database, and select *Delete*.
- Check **Close existing connections**.
- Select **OK**.
- In the PMC, update the database:

  ```
  Update-Database
  ```

Run the app and verify you can create/edit/display movies with a `Rating` field. If the database isn't seeded, set a break point in the `SeedData.Initialize` method.

Additional resources

- YouTube version of this tutorial

PREVIOUS: ADDING SEARCH

NEXT: ADDING VALIDATION

Part 8, add validation to an ASP.NET Core Razor Page

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

In this section, validation logic is added to the `Movie` model. The validation rules are enforced any time a user creates or edits a movie.

Validation

A key tenet of software development is called DRY ("**D**on't **R**epeat **Y**ourself"). Razor Pages encourages development where functionality is specified once, and it's reflected throughout the app. DRY can help:

- Reduce the amount of code in an app.
- Make the code less error prone, and easier to test and maintain.

The validation support provided by Razor Pages and Entity Framework is a good example of the DRY principle. Validation rules are declaratively specified in one place (in the model class), and the rules are enforced everywhere in the app.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

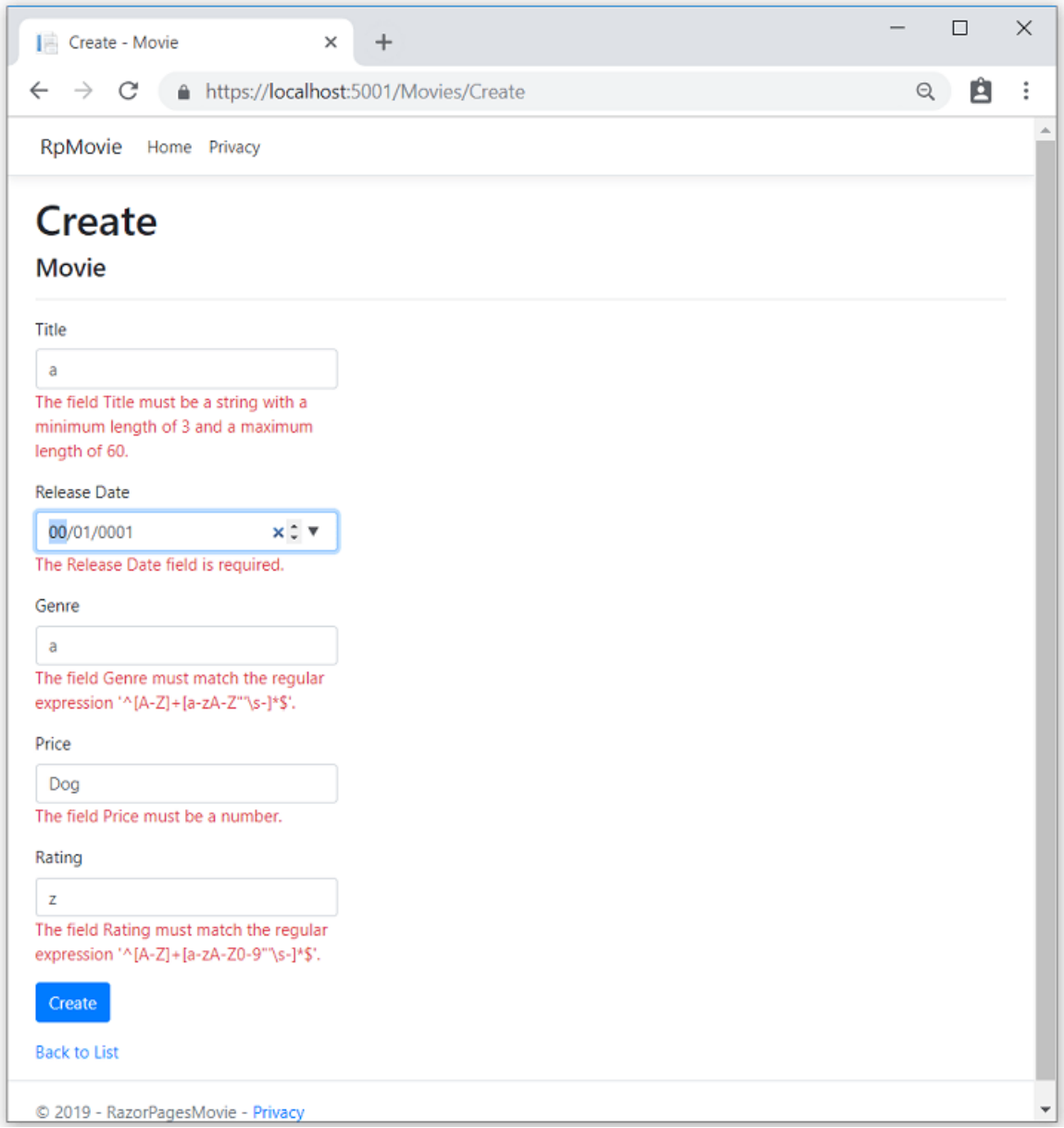
- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White space, numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI in Razor Pages

Run the app and navigate to Pages/Movies.

Select the **Create New** link. Complete the form with some invalid values. When jQuery client-side validation detects the error, it displays an error message.



NOTE

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered a validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (when a user has JavaScript disabled).

A significant benefit is that **no** code changes were necessary in the Create or Edit pages. Once DataAnnotations were applied to the model, the validation UI was enabled. The Razor Pages created in this tutorial automatically picked up the validation rules (using validation attributes on the properties of the `Movie` model class). Test validation using the Edit page, the same validation is applied.

The form data isn't posted to the server until there are no client-side validation errors. Verify form data isn't posted by one or more of the following approaches:

- Put a break point in the `OnPostAsync` method. Submit the form (select **Create** or **Save**). The break point is never hit.
- Use the Fiddler tool.

- Use the browser developer tools to monitor network traffic.

Server-side validation

When JavaScript is disabled in the browser, submitting the form with errors will post to the server.

Optional, test server-side validation:

- Disable JavaScript in the browser. You can disable JavaScript using browser's developer tools. If you can't disable JavaScript in the browser, try another browser.
- Set a break point in the `OnPostAsync` method of the Create or Edit page.
- Submit a form with invalid data.
- Verify the model state is invalid:

```
if (!ModelState.IsValid)
{
    return Page();
}
```

Alternatively, you can Disable client-side validation on the server.

The following code shows a portion of the *Create.cshtml* page scaffolded earlier in the tutorial. It's used by the Create and Edit pages to display the initial form and to redisplay the form in the event of an error.

```
<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Movie.Title" class="control-label"></label>
        <input asp-for="Movie.Title" class="form-control" />
        <span asp-validation-for="Movie.Title" class="text-danger"></span>
    </div>
```

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side. The Validation Tag Helper displays validation errors. See Validation for more information.

The Create and Edit pages have no validation rules in them. The validation rules and the error strings are specified only in the `Movie` class. These validation rules are automatically applied to Razor Pages that edit the `Movie` model.

When validation logic needs to change, it's done only in the model. Validation is applied consistently throughout the application (validation logic is defined in one place). Validation in one place helps keep the code clean, and makes it easier to maintain and update.

Using DataType Attributes

Examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. The `DataType` attribute is applied to the `ReleaseDate` and `Price` properties.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supplies attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email). Use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type. `DataType` attributes are not validation attributes. In the sample application, only the date is displayed, without time.

The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`. A date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers consume. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should be applied when the value is displayed for editing. You might not want that behavior for some fields. For example, in currency values, you probably don't want the currency symbol in the edit UI.

The `DisplayFormat` attribute can be used by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable the ASP.NET Core framework to choose the right field template to render the data. The `DisplayFormat`, if used by itself, uses the string template.

Note: jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client-side validation error, even when the date is in the specified range:

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

Get started with Razor Pages and EF Core shows advanced EF Core operations with Razor Pages.

Apply migrations

The DataAnnotations applied to the class changes the schema. For example, the DataAnnotations applied to the `Title` field:

```
[StringLength(60, MinimumLength = 3)]
[Required]
public string Title { get; set; }
```

- Limits the characters to 60.
- Doesn't allow a `null` value.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

The `Movie` table currently has the following schema:

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (MAX)  NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (MAX)  NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (MAX)  NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

The preceding schema changes don't cause EF to throw an exception. However, create a migration so the schema is consistent with the model.

From the **Tools** menu, select **NuGet Package Manager > Package Manager Console**. In the PMC, enter the following commands:

```
Add-Migration New_DataAnnotations
Update-Database
```

`Update-Database` runs the `Up` methods of the `New_DataAnnotations` class. Examine the `Up` method:

```
public partial class New_DataAnnotations : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "Title",
            table: "Movie",
            maxLength: 60,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);

        migrationBuilder.AlterColumn<string>(
            name: "Rating",
            table: "Movie",
            maxLength: 5,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);

        migrationBuilder.AlterColumn<string>(
            name: "Genre",
            table: "Movie",
            maxLength: 30,
            nullable: false,
            oldClrType: typeof(string),
            oldNullable: true);
    }
```

The updated `Movie` table has the following schema:

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (60)   NOT NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (30)   NOT NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (5)    NOT NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core app in Azure with SQL Database.

Thanks for completing this introduction to Razor Pages. Get started with Razor Pages and EF Core is an excellent follow up to this tutorial.

Additional resources

- Tag Helpers in forms in ASP.NET Core
- Globalization and localization in ASP.NET Core
- Tag Helpers in ASP.NET Core
- Author Tag Helpers in ASP.NET Core
- YouTube version of this tutorial

PREVIOUS: ADDING A NEW FIELD

Filter methods for Razor Pages in ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

By Rick Anderson

Razor Page filters IPageFilter and IAsyncPageFilter allow Razor Pages to run code before and after a Razor Page handler is run. Razor Page filters are similar to ASP.NET Core MVC action filters, except they can't be applied to individual page handler methods.

Razor Page filters:

- Run code after a handler method has been selected, but before model binding occurs.
- Run code before the handler method executes, after model binding is complete.
- Run code after the handler method executes.
- Can be implemented on a page or globally.
- Cannot be applied to specific page handler methods.
- Can have constructor dependencies populated by Dependency Injection (DI). For more information, see ServiceFilterAttribute and TypeFilterAttribute.

While page constructors and middleware enable executing custom code before a handler method executes, only Razor Page filters enable access to HttpContext and the page. Middleware has access to the `HttpContext`, but not to the "page context". Filters have a FilterContext derived parameter, which provides access to `HttpContext`. Here's a sample for a page filter: Implement a filter attribute that adds a header to the response, something that can't be done with constructors or middleware. Access to the page context, which includes access to the instances of the page and it's model, are only available when executing filters, handlers, or the body of a Razor Page.

View or download sample code (how to download)

Razor Page filters provide the following methods, which can be applied globally or at the page level:

- Synchronous methods:
  - OnPageHandlerSelected : Called after a handler method has been selected, but before model binding occurs.
  - OnPageHandlerExecuting : Called before the handler method executes, after model binding is complete.
  - OnPageHandlerExecuted : Called after the handler method executes, before the action result.
- Asynchronous methods:
  - OnPageHandlerSelectionAsync : Called asynchronously after the handler method has been selected, but before model binding occurs.
  - OnPageHandlerExecutionAsync : Called asynchronously before the handler method is invoked, after model binding is complete.

Implement **either** the synchronous or the async version of a filter interface, **not** both. The framework checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both interfaces are implemented, only the async methods are called. The same rule applies to overrides in pages, implement the synchronous or the async version of the override, not both.

Implement Razor Page filters globally

The following code implements `IAsyncPageFilter`:

```
public class SampleAsyncPageFilter : IAsyncPageFilter
{
    private readonly IConfiguration _config;

    public SampleAsyncPageFilter(IConfiguration config)
    {
        _config = config;
    }

    public Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent",
                                                        out StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
                               "SampleAsyncPageFilter.OnPageHandlerSelectionAsync",
                               value, key.ToString());

        return Task.CompletedTask;
    }

    public async Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,
                                                  PageHandlerExecutionDelegate next)
    {
        // Do post work.
        await next.Invoke();
    }
}
```

In the preceding code, `ProcessUserAgent.Write` is user supplied code that works with the user agent string.

The following code enables the `SampleAsyncPageFilter` in the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddMvcOptions(options =>
        {
            options.Filters.Add(new SampleAsyncPageFilter(Configuration));
        });
}
```

The following code calls AddFolderApplicationModelConvention to apply the `SampleAsyncPageFilter` to only pages in */Movies*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
    {
        options.Conventions.AddFolderApplicationModelConvention(
            "/Movies",
            model => model.Filters.Add(new SampleAsyncPageFilter(Configuration)));
    });
}
```

The following code implements the synchronous `IPageFilter`:

```
public class SamplePageFilter : IPageFilter
{
    private readonly IConfiguration _config;

    public SamplePageFilter(IConfiguration config)
    {
        _config = config;
    }

    public void OnPageHandlerSelected(PageHandlerSelectedContext context)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent", out StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
                               "SamplePageFilter.OnPageHandlerSelected",
                               value, key.ToString());
    }

    public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        Debug.WriteLine("Global sync OnPageHandlerExecuting called.");
    }

    public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
    {
        Debug.WriteLine("Global sync OnPageHandlerExecuted called.");
    }
}
```

The following code enables the `SamplePageFilter`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddMvcOptions(options =>
        {
            options.Filters.Add(new SamplePageFilter(Configuration));
        });
}
```

Implement Razor Page filters by overriding filter methods

The following code overrides the asynchronous Razor Page filters:

```
public class IndexModel : PageModel
{
    private readonly IConfiguration _config;

    public IndexModel(IConfiguration config)
    {
        _config = config;
    }

    public override Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
        Debug.WriteLine("/IndexModel OnPageHandlerSelectionAsync");
        return Task.CompletedTask;
    }

    public async override Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,
                                                           PageHandlerExecutionDelegate next)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent", out StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
                               "/IndexModel-OnPageHandlerExecutionAsync",
                                value, key.ToString());

        await next.Invoke();
    }
}
```

Implement a filter attribute

The built-in attribute-based filter OnResultExecutionAsync filter can be subclassed. The following filter adds a header to the response:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace PageFilter.Filters
{
    public class AddHeaderAttribute  : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute (string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(_name, new string[] { _value });
        }
    }
}
```

The following code applies the `AddHeader` attribute:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using PageFilter.Filters;

namespace PageFilter.Movies
{
    [AddHeader("Author", "Rick")]
    public class TestModel : PageModel
    {
        public void OnGet()
        {

        }
    }
}
```

Use a tool such as the browser developer tools to examine the headers. Under **Response Headers**, `author: Rick` is displayed.

See Overriding the default order for instructions on overriding the order.

See Cancellation and short circuiting for instructions to short-circuit the filter pipeline from a filter.

Authorize filter attribute

The Authorize attribute can be applied to a `PageModel`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace PageFilter.Pages
{
    [Authorize]
    public class ModelWithAuthFilterModel : PageModel
    {
        public IActionResult OnGet() => Page();
    }
}
```

By Rick Anderson

Razor Page filters IPageFilter and IAsyncPageFilter allow Razor Pages to run code before and after a Razor Page handler is run. Razor Page filters are similar to ASP.NET Core MVC action filters, except they can't be applied to individual page handler methods.

Razor Page filters:

- Run code after a handler method has been selected, but before model binding occurs.
- Run code before the handler method executes, after model binding is complete.
- Run code after the handler method executes.
- Can be implemented on a page or globally.
- Cannot be applied to specific page handler methods.

Code can be run before a handler method executes using the page constructor or middleware, but only Razor Page filters have access to HttpContext. Filters have a FilterContext derived parameter, which provides access to `HttpContext`. For example, the Implement a filter attribute sample adds a header to the response, something that can't be done with constructors or middleware.

View or download sample code (how to download)

Razor Page filters provide the following methods, which can be applied globally or at the page level:

- Synchronous methods:
  - OnPageHandlerSelected : Called after a handler method has been selected, but before model binding occurs.
  - OnPageHandlerExecuting : Called before the handler method executes, after model binding is complete.
  - OnPageHandlerExecuted : Called after the handler method executes, before the action result.
- Asynchronous methods:
  - OnPageHandlerSelectionAsync : Called asynchronously after the handler method has been selected, but before model binding occurs.
  - OnPageHandlerExecutionAsync : Called asynchronously before the handler method is invoked, after model binding is complete.

> **NOTE**
>
> Implement **either** the synchronous or the async version of a filter interface, not both. The framework checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both interfaces are implemented, only the async methods are called. The same rule applies to overrides in pages, implement the synchronous or the async version of the override, not both.

Implement Razor Page filters globally

The following code implements `IAsyncPageFilter`:

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace PageFilter.Filters
{
    public class SampleAsyncPageFilter : IAsyncPageFilter
    {
        private readonly ILogger _logger;

        public SampleAsyncPageFilter(ILogger logger)
        {
            _logger = logger;
        }

        public async Task OnPageHandlerSelectionAsync(
                                        PageHandlerSelectedContext context)
        {
            _logger.LogDebug("Global OnPageHandlerSelectionAsync called.");
            await Task.CompletedTask;
        }

        public async Task OnPageHandlerExecutionAsync(
                                        PageHandlerExecutingContext context,
                                        PageHandlerExecutionDelegate next)
        {
            _logger.LogDebug("Global OnPageHandlerExecutionAsync called.");
            await next.Invoke();
        }
    }
}
```

In the preceding code, ILogger is not required. It's used in the sample to provide trace information for the

application.

The following code enables the `SampleAsyncPageFilter` in the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new SampleAsyncPageFilter(_logger));
    });
}
```

The following code shows the complete `Startup` class:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using PageFilter.Filters;

namespace PageFilter
{
    public class Startup
    {
        ILogger _logger;
        public Startup(ILoggerFactory loggerFactory, IConfiguration configuration)
        {
            _logger = loggerFactory.CreateLogger<GlobalFiltersLogger>();
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc(options =>
            {
                options.Filters.Add(new SampleAsyncPageFilter(_logger));
            });
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();

            app.UseMvc();
        }
    }
}
```

The following code calls `AddFolderApplicationModelConvention` to apply the `SampleAsyncPageFilter` to only pages in

/subFolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
       .AddRazorPagesOptions(options =>
       {
           options.Conventions.AddFolderApplicationModelConvention(
               "/subFolder",
               model => model.Filters.Add(new SampleAsyncPageFilter(_logger)));
       });
}
```

The following code implements the synchronous `IPageFilter`:

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;

namespace PageFilter.Filters
{
    public class SamplePageFilter : IPageFilter
    {
        private readonly ILogger _logger;

        public SamplePageFilter(ILogger logger)
        {
            _logger = logger;
        }

        public void OnPageHandlerSelected(PageHandlerSelectedContext context)
        {
            _logger.LogDebug("Global sync OnPageHandlerSelected called.");
        }

        public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
        {
            _logger.LogDebug("Global sync PageHandlerExecutingContext called.");
        }

        public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
        {
            _logger.LogDebug("Global sync OnPageHandlerExecuted called.");
        }
    }
}
```

The following code enables the `SamplePageFilter`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new SamplePageFilter(_logger));
    });
}
```

Implement Razor Page filters by overriding filter methods

The following code overrides the synchronous Razor Page filters:

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace PageFilter.Pages
{
    public class IndexModel : PageModel
    {
        private readonly ILogger _logger;

        public IndexModel(ILogger<IndexModel> logger)
        {
            _logger = logger;
        }
        public string Message { get; set; }

        public void OnGet()
        {
            _logger.LogDebug("IndexModel/OnGet");
        }

        public override void OnPageHandlerSelected(
                                    PageHandlerSelectedContext context)
        {
            _logger.LogDebug("IndexModel/OnPageHandlerSelected");
        }

        public override void OnPageHandlerExecuting(
                                    PageHandlerExecutingContext context)
        {
            Message = "Message set in handler executing";
            _logger.LogDebug("IndexModel/OnPageHandlerExecuting");
        }


        public override void OnPageHandlerExecuted(
                                    PageHandlerExecutedContext context)
        {
            _logger.LogDebug("IndexModel/OnPageHandlerExecuted");
        }
    }
}
```

Implement a filter attribute

The built-in attribute-based filter OnResultExecutionAsync filter can be subclassed. The following filter adds a header to the response:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace PageFilter.Filters
{
    public class AddHeaderAttribute  : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute (string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(_name, new string[] { _value });
        }
    }
}
```

The following code applies the `AddHeader` attribute:

```
[AddHeader("Author", "Rick")]
public class ContactModel : PageModel
{
    private readonly ILogger _logger;

    public ContactModel(ILogger<ContactModel> logger)
    {
        _logger = logger;
    }
    public string Message { get; set; }

    public async Task OnGetAsync()
    {
        Message = "Your contact page.";
        _logger.LogDebug("Contact/OnGet");
        await Task.CompletedTask;
    }
}
```

See Overriding the default order for instructions on overriding the order.

See Cancellation and short circuiting for instructions to short-circuit the filter pipeline from a filter.

Authorize filter attribute

The Authorize attribute can be applied to a `PageModel`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace PageFilter.Pages
{
    [Authorize]
    public class ModelWithAuthFilterModel : PageModel
    {
        public IActionResult OnGet() => Page();
    }
}
```

Razor Pages route and app conventions in ASP.NET Core

9/22/2020 • 35 minutes to read • Edit Online

Learn how to use page route and app model provider conventions to control page routing, discovery, and processing in Razor Pages apps.

When you need to configure custom page routes for individual pages, configure routing to pages with the AddPageRoute convention described later in this topic.

To specify a page route, add route segments, or add parameters to a route, use the page's `@page` directive. For more information, see Custom routes.

There are reserved words that can't be used as route segments or parameter names. For more information, see Routing: Reserved routing names.

View or download sample code (how to download)

SCENARIO	THE SAMPLE DEMONSTRATES ...
Model conventions Conventions.Add • IPageRouteModelConvention • IPageApplicationModelConvention • IPageHandlerModelConvention	Add a route template and header to an app's pages.
Page route action conventions • AddFolderRouteModelConvention • AddPageRouteModelConvention • AddPageRoute	Add a route template to pages in a folder and to a single page.
Page model action conventions • AddFolderApplicationModelConvention • AddPageApplicationModelConvention • ConfigureFilter (filter class, lambda expression, or filter factory)	Add a header to pages in a folder, add a header to a single page, and configure a filter factory to add a header to an app's pages.

Razor Pages conventions are configured using an AddRazorPages overload that configures RazorPagesOptions in `Startup.ConfigureServices`. The following convention examples are explained later in this topic:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
    {
        options.Conventions.Add( ... );
        options.Conventions.AddFolderRouteModelConvention(
            "/OtherPages", model => { ... });
        options.Conventions.AddPageRouteModelConvention(
            "/About", model => { ... });
        options.Conventions.AddPageRoute(
            "/Contact", "TheContactPage/{text?}");
        options.Conventions.AddFolderApplicationModelConvention(
            "/OtherPages", model => { ... });
        options.Conventions.AddPageApplicationModelConvention(
            "/About", model => { ... });
        options.Conventions.ConfigureFilter(model => { ... });
        options.Conventions.ConfigureFilter( ... );
    });
}
```

Route order

Routes specify an Order for processing (route matching).

ORDER	BEHAVIOR
-1	The route is processed before other routes are processed.
0	Order isn't specified (default value). Not assigning `Order` (`Order = null`) defaults the route `Order` to 0 (zero) for processing.
1, 2, ... n	Specifies the route processing order.

Route processing is established by convention:

- Routes are processed in sequential order (-1, 0, 1, 2, ... n).
- When routes have the same `Order`, the most specific route is matched first followed by less specific routes.
- When routes with the same `Order` and the same number of parameters match a request URL, routes are processed in the order that they're added to the PageConventionCollection.

If possible, avoid depending on an established route processing order. Generally, routing selects the correct route with URL matching. If you must set route `Order` properties to route requests correctly, the app's routing scheme is probably confusing to clients and fragile to maintain. Seek to simplify the app's routing scheme. The sample app requires an explicit route processing order to demonstrate several routing scenarios using a single app. However, you should attempt to avoid the practice of setting route `Order` in production apps.

Razor Pages routing and MVC controller routing share an implementation. Information on route order in the MVC topics is available at Routing to controller actions: Ordering attribute routes.

Model conventions

Add a delegate for IPageConvention to add model conventions that apply to Razor Pages.

Add a route model convention to all pages

Use Conventions to create and add an IPageRouteModelConvention to the collection of IPageConvention instances that are applied during page route model construction.

The sample app adds a `{globalTemplate?}` route template to all of the pages in the app:

```
public class GlobalTemplatePageRouteModelConvention
    : IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 1,
                    Template = AttributeRouteModel.CombineTemplates(
                        selector.AttributeRouteModel.Template,
                        "{globalTemplate?}"),
                }
            });
        }
    }
}
```

The Order property for the AttributeRouteModel is set to `1`. This ensures the following route matching behavior in the sample app:

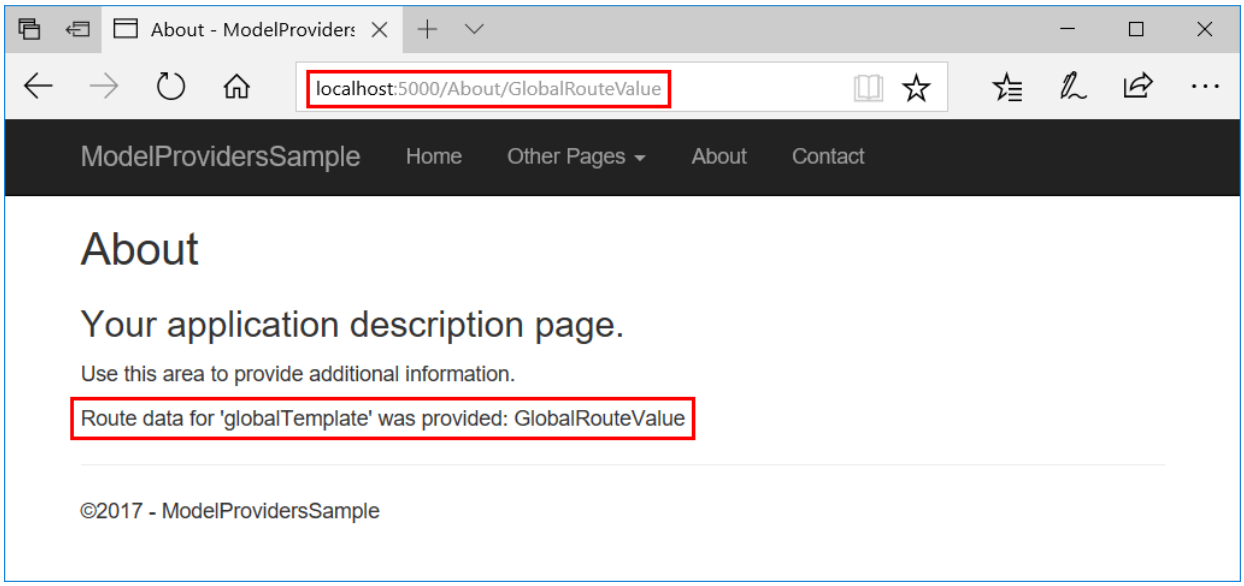
- A route template for `TheContactPage/{text?}` is added later in the topic. The Contact Page route has a default order of `null` (`Order = 0`), so it matches before the `{globalTemplate?}` route template.
- An `{aboutTemplate?}` route template is added later in the topic. The `{aboutTemplate?}` template is given an `Order` of `2`. When the About page is requested at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.
- An `{otherPagesTemplate?}` route template is added later in the topic. The `{otherPagesTemplate?}` template is given an `Order` of `2`. When any page in the *Pages/OtherPages* folder is requested with a route parameter (for example, `/OtherPages/Page1/RouteDataValue`), "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Razor Pages options, such as adding Conventions, are added when Razor Pages is added to the service collection in `Startup.ConfigureServices`. For an example, see the sample app.

```
options.Conventions.Add(new GlobalTemplatePageRouteModelConvention());
```

Request the sample's About page at `localhost:5000/About/GlobalRouteValue` and inspect the result:



Add an app model convention to all pages

Use Conventions to create and add an IPageApplicationModelConvention to the collection of IPageConvention instances that are applied during page app model construction.

To demonstrate this and other conventions later in the topic, the sample app includes an `AddHeaderAttribute` class. The class constructor accepts a `name` string and a `values` string array. These values are used in its `OnResultExecuting` method to set a response header. The full class is shown in the Page model action conventions section later in the topic.

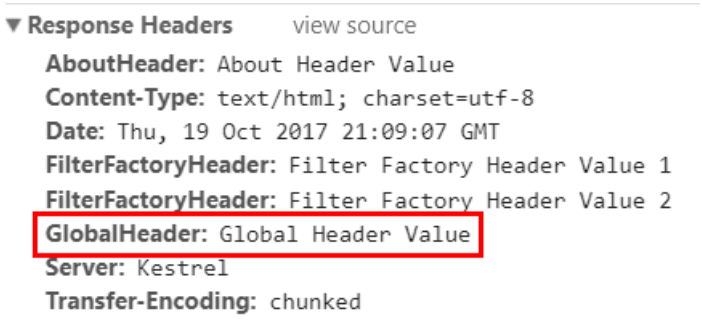
The sample app uses the `AddHeaderAttribute` class to add a header, `GlobalHeader`, to all of the pages in the app:

```
public class GlobalHeaderPageApplicationModelConvention
    : IPageApplicationModelConvention
{
    public void Apply(PageApplicationModel model)
    {
        model.Filters.Add(new AddHeaderAttribute(
            "GlobalHeader", new string[] { "Global Header Value" }));
    }
}
```

Startup.cs:

```
options.Conventions.Add(new GlobalHeaderPageApplicationModelConvention());
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



Add a handler model convention to all pages

Use Conventions to create and add an IPageHandlerModelConvention to the collection of IPageConvention instances that are applied during page handler model construction.

```
public class GlobalPageHandlerModelConvention
    : IPageHandlerModelConvention
{
    public void Apply(PageHandlerModel model)
    {
        // Access the PageHandlerModel
    }
}
```

Startup.cs:

```
options.Conventions.Add(new GlobalPageHandlerModelConvention());
```

Page route action conventions

The default route model provider that derives from IPageRouteModelProvider invokes conventions which are designed to provide extensibility points for configuring page routes.

Folder route model convention

Use AddFolderRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for all of the pages under the specified folder.

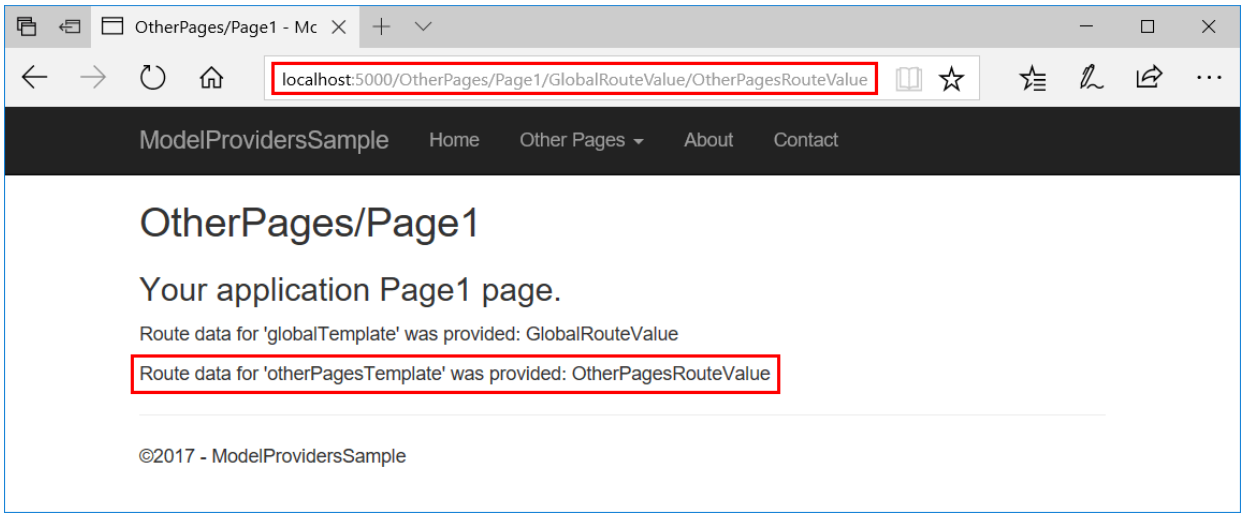
The sample app uses AddFolderRouteModelConvention to add an `{otherPagesTemplate?}` route template to the pages in the *OtherPages* folder:

```
options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{otherPagesTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If a page in the *Pages/OtherPages* folder is requested with a route parameter value (for example, `/OtherPages/Page1/RouteDataValue`), "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1/GlobalRouteValue/OtherPagesRouteValue` and inspect the result:



Page route model convention

Use AddPageRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for the page with the specified name.

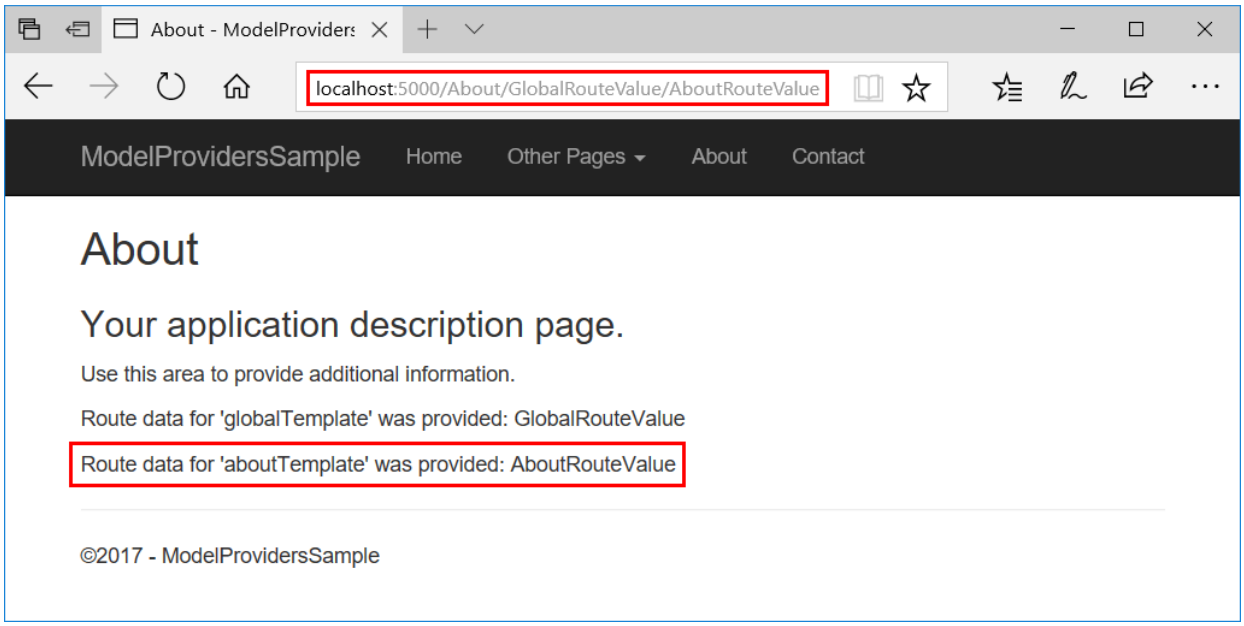
The sample app uses `AddPageRouteModelConvention` to add an `{aboutTemplate?}` route template to the About page:

```
options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{aboutTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If the About page is requested with a route parameter value at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's About page at `localhost:5000/About/GlobalRouteValue/AboutRouteValue` and inspect the result:



Use a parameter transformer to customize page routes

Page routes generated by ASP.NET Core can be customized using a parameter transformer. A parameter transformer implements `IOutboundParameterTransformer` and transforms the value of parameters. For example, a custom `SlugifyParameterTransformer` parameter transformer changes the `SubscriptionManagement` route value to `subscription-management`.

The `PageRouteTransformerConvention` page route model convention applies a parameter transformer to the folder and file name segments of automatically generated page routes in an app. For example, the Razor Pages file at */Pages/SubscriptionManagement/ViewAll.cshtml* would have its route rewritten from `/SubscriptionManagement/ViewAll` to `/subscription-management/view-all`.

`PageRouteTransformerConvention` only transforms the automatically generated segments of a page route that come from the Razor Pages folder and file name. It doesn't transform route segments added with the `@page` directive. The convention also doesn't transform routes added by AddPageRoute.

The `PageRouteTransformerConvention` is registered as an option in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
    {
        options.Conventions.Add(
            new PageRouteTransformerConvention(
                new SlugifyParameterTransformer()));
    });
}
```

```
public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) { return null; }

        return Regex.Replace(value.ToString(),
                             "([a-z])([A-Z])",
                             "$1-$2",
                             RegexOptions.CultureInvariant,
                             TimeSpan.FromMilliseconds(100)).ToLowerInvariant();
    }
}
```

WARNING

When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Configure a page route

Use AddPageRoute to configure a route to a page at the specified page path. Generated links to the page use your specified route. `AddPageRoute` uses `AddPageRouteModelConvention` to establish the route.

The sample app creates a route to `/TheContactPage` for *Contact.cshtml*:

```
options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");
```

The Contact page can also be reached at `/Contact` via its default route.

The sample app's custom route to the Contact page allows for an optional `text` route segment (`{text?}`). The page also includes this optional segment in its `@page` directive in case the visitor accesses the page at its `/Contact` route:

```
@page "{text?}"
@model ContactModel
@{
    ViewData["Title"] = "Contact";
}

<h1>@ViewData["Title"]</h1>
<h2>@Model.Message</h2>

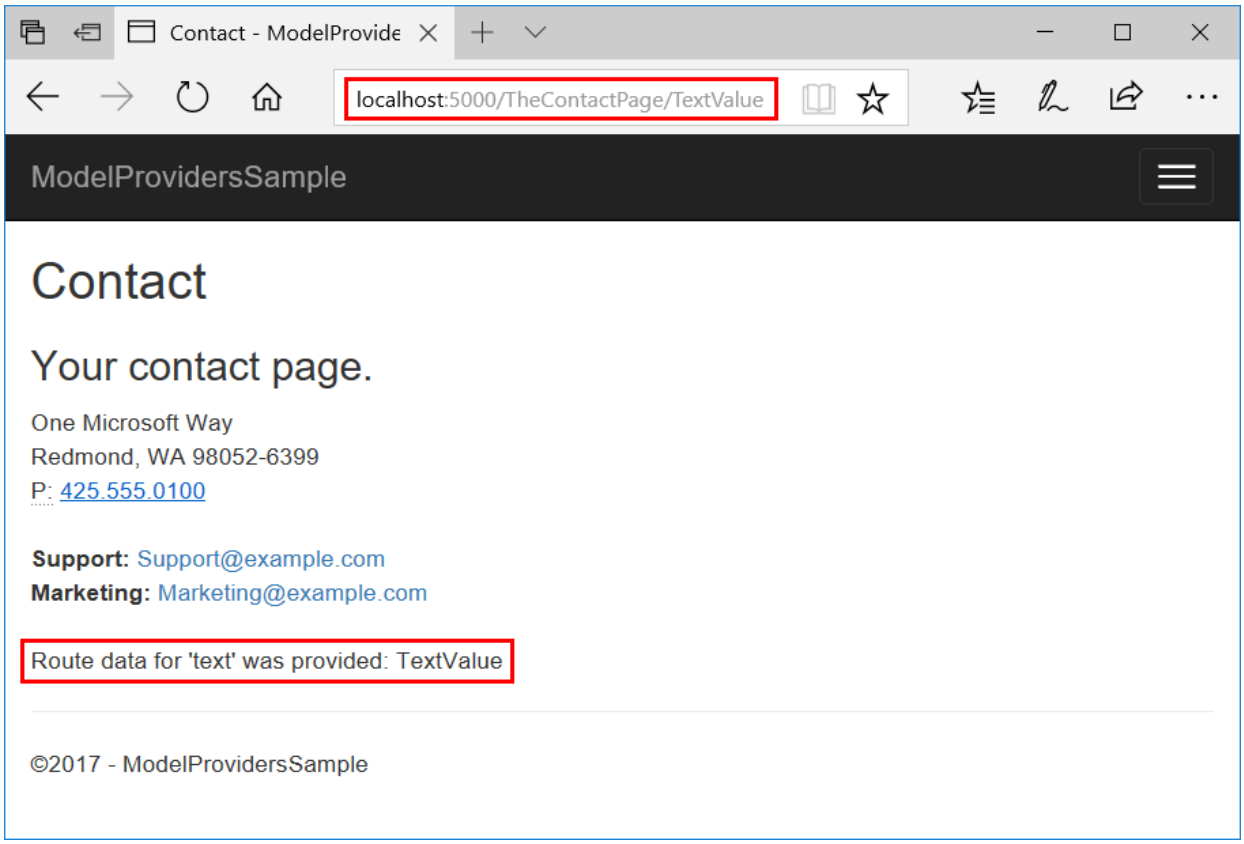
<address>
    One Microsoft Way<br>
    Redmond, WA 98052-6399<br>
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a href="mailto:Support@example.com">Support@example.com</a><br>
    <strong>Marketing:</strong> <a href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>

<p>@Model.RouteDataTextTemplateValue</p>
```

Note that the URL generated for the **Contact** link in the rendered page reflects the updated route:

Visit the Contact page at either its ordinary route, `/Contact`, or the custom route, `/TheContactPage`. If you supply an additional `text` route segment, the page shows the HTML-encoded segment that you provide:



Page model action conventions

The default page model provider that implements IPageApplicationModelProvider invokes conventions which are designed to provide extensibility points for configuring page models. These conventions are useful when building and modifying page discovery and processing scenarios.

For the examples in this section, the sample app uses an `AddHeaderAttribute` class, which is a ResultFilterAttribute, that applies a response header:

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string[] _values;

    public AddHeaderAttribute(string name, string[] values)
    {
        _name = name;
        _values = values;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _values);
        base.OnResultExecuting(context);
    }
}
```

Using conventions, the sample demonstrates how to apply the attribute to all of the pages in a folder and to a single page.

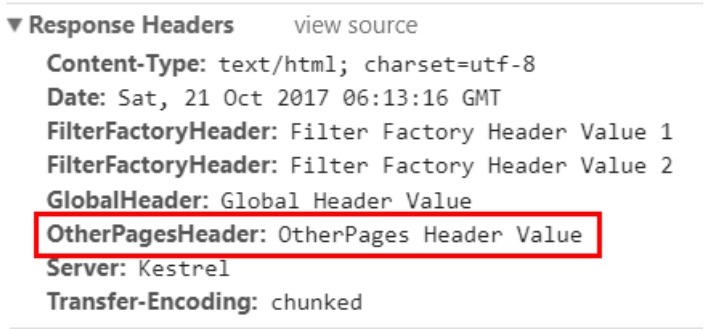
Folder app model convention

Use AddFolderApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on PageApplicationModel instances for all pages under the specified folder.

The sample demonstrates the use of `AddFolderApplicationModelConvention` by adding a header, `OtherPagesHeader`, to the pages inside the *OtherPages* folder of the app:

```
options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "OtherPagesHeader", new string[] { "OtherPages Header Value" }));
});
```

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1` and inspect the headers to view the result:



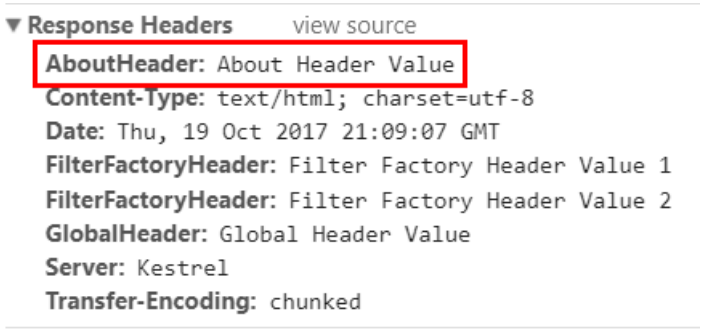
Page app model convention

Use AddPageApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on the PageApplicationModel for the page with the specified name.

The sample demonstrates the use of `AddPageApplicationModelConvention` by adding a header, `AboutHeader`, to the About page:

```
options.Conventions.AddPageApplicationModelConvention("/About", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "AboutHeader", new string[] { "About Header Value" }));
});
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



Configure a filter

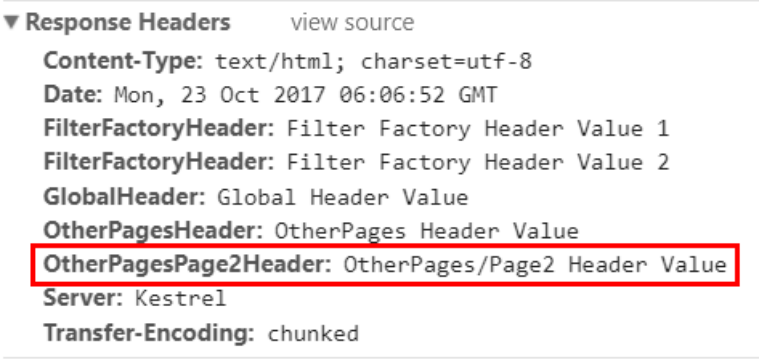
ConfigureFilter configures the specified filter to apply. You can implement a filter class, but the sample app shows how to implement a filter in a lambda expression, which is implemented behind-the-scenes as a factory that returns a filter:

```
options.Conventions.ConfigureFilter(model =>
{
    if (model.RelativePath.Contains("OtherPages/Page2"))
    {
        return new AddHeaderAttribute(
            "OtherPagesPage2Header",
            new string[] { "OtherPages/Page2 Header Value" });
    }
    return new EmptyFilter();
});
```

The page app model is used to check the relative path for segments that lead to the Page2 page in the *OtherPages* folder. If the condition passes, a header is added. If not, the `EmptyFilter` is applied.

`EmptyFilter` is an Action filter. Since Action filters are ignored by Razor Pages, the `EmptyFilter` has no effect as intended if the path doesn't contain `OtherPages/Page2`.

Request the sample's Page2 page at `localhost:5000/OtherPages/Page2` and inspect the headers to view the result:



Configure a filter factory

ConfigureFilter configures the specified factory to apply filters to all Razor Pages.

The sample app provides an example of using a filter factory by adding a header, `FilterFactoryHeader`, with two

values to the app's pages:

```
options.Conventions.ConfigureFilter(new AddHeaderWithFactory());
```

AddHeaderWithFactory.cs:

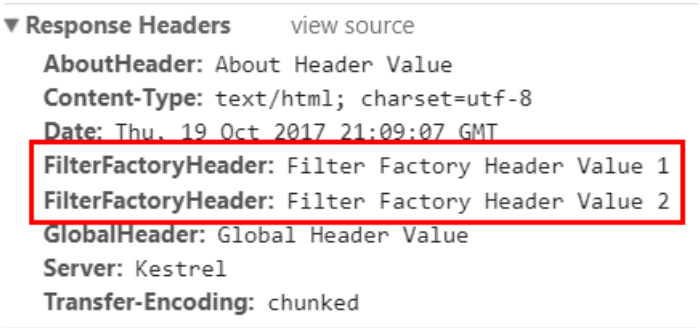
```
public class AddHeaderWithFactory : IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new AddHeaderFilter();
    }

    private class AddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "FilterFactoryHeader",
                new string[]
                {
                    "Filter Factory Header Value 1",
                    "Filter Factory Header Value 2"
                });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



MVC Filters and the Page filter (IPageFilter)

MVC Action filters are ignored by Razor Pages, since Razor Pages use handler methods. Other types of MVC filters are available for you to use: Authorization, Exception, Resource, and Result. For more information, see the Filters topic.

The Page filter (IPageFilter) is a filter that applies to Razor Pages. For more information, see Filter methods for Razor Pages.

Additional resources

- Razor Pages authorization conventions in ASP.NET Core
- Areas in ASP.NET Core

Learn how to use page route and app model provider conventions to control page routing, discovery, and processing in Razor Pages apps.

When you need to configure custom page routes for individual pages, configure routing to pages with the AddPageRoute convention described later in this topic.

To specify a page route, add route segments, or add parameters to a route, use the page's `@page` directive. For more information, see Custom routes.

There are reserved words that can't be used as route segments or parameter names. For more information, see Routing: Reserved routing names.

View or download sample code (how to download)

SCENARIO	THE SAMPLE DEMONSTRATES ...
Model conventions Conventions.Add • IPageRouteModelConvention • IPageApplicationModelConvention • IPageHandlerModelConvention	Add a route template and header to an app's pages.
Page route action conventions • AddFolderRouteModelConvention • AddPageRouteModelConvention • AddPageRoute	Add a route template to pages in a folder and to a single page.
Page model action conventions • AddFolderApplicationModelConvention • AddPageApplicationModelConvention • ConfigureFilter (filter class, lambda expression, or filter factory)	Add a header to pages in a folder, add a header to a single page, and configure a filter factory to add a header to an app's pages.

Razor Pages conventions are added and configured using the AddRazorPagesOptions extension method to AddMvc on the service collection in the `Startup` class. The following convention examples are explained later in this topic:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.Conventions.Add( ... );
            options.Conventions.AddFolderRouteModelConvention(
                "/OtherPages", model => { ... });
            options.Conventions.AddPageRouteModelConvention(
                "/About", model => { ... });
            options.Conventions.AddPageRoute(
                "/Contact", "TheContactPage/{text?}");
            options.Conventions.AddFolderApplicationModelConvention(
                "/OtherPages", model => { ... });
            options.Conventions.AddPageApplicationModelConvention(
                "/About", model => { ... });
            options.Conventions.ConfigureFilter(model => { ... });
            options.Conventions.ConfigureFilter( ... );
        });
}
```

Route order

Routes specify an Order for processing (route matching).

ORDER	BEHAVIOR
-1	The route is processed before other routes are processed.
0	Order isn't specified (default value). Not assigning `Order` (`Order = null`) defaults the route `Order` to 0 (zero) for processing.
1, 2, ... n	Specifies the route processing order.

Route processing is established by convention:

- Routes are processed in sequential order (-1, 0, 1, 2, ... n).
- When routes have the same `Order`, the most specific route is matched first followed by less specific routes.
- When routes with the same `Order` and the same number of parameters match a request URL, routes are processed in the order that they're added to the PageConventionCollection.

If possible, avoid depending on an established route processing order. Generally, routing selects the correct route with URL matching. If you must set route `Order` properties to route requests correctly, the app's routing scheme is probably confusing to clients and fragile to maintain. Seek to simplify the app's routing scheme. The sample app requires an explicit route processing order to demonstrate several routing scenarios using a single app. However, you should attempt to avoid the practice of setting route `Order` in production apps.

Razor Pages routing and MVC controller routing share an implementation. Information on route order in the MVC topics is available at Routing to controller actions: Ordering attribute routes.

Model conventions

Add a delegate for IPageConvention to add model conventions that apply to Razor Pages.

Add a route model convention to all pages

Use Conventions to create and add an IPageRouteModelConvention to the collection of IPageConvention

instances that are applied during page route model construction.

The sample app adds a `{globalTemplate?}` route template to all of the pages in the app:

```
public class GlobalTemplatePageRouteModelConvention
    : IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 1,
                    Template = AttributeRouteModel.CombineTemplates(
                        selector.AttributeRouteModel.Template,
                        "{globalTemplate?}"),
                }
            });
        }
    }
}
```

The Order property for the AttributeRouteModel is set to `1`. This ensures the following route matching behavior in the sample app:

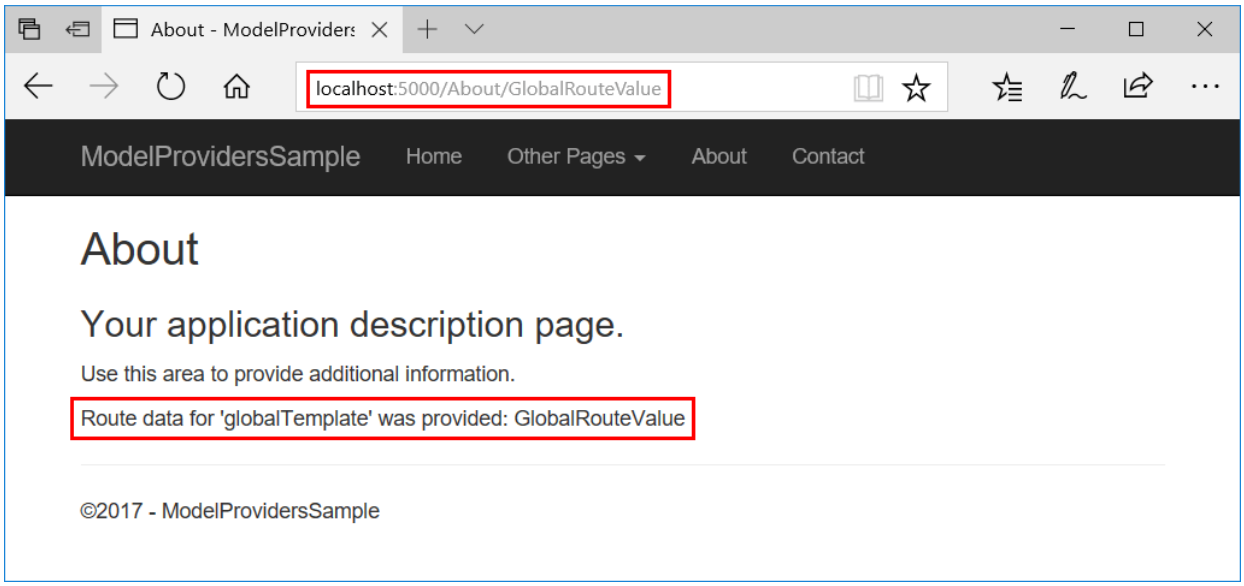
- A route template for `TheContactPage/{text?}` is added later in the topic. The Contact Page route has a default order of `null` (`Order = 0`), so it matches before the `{globalTemplate?}` route template.
- An `{aboutTemplate?}` route template is added later in the topic. The `{aboutTemplate?}` template is given an `Order` of `2`. When the About page is requested at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.
- An `{otherPagesTemplate?}` route template is added later in the topic. The `{otherPagesTemplate?}` template is given an `Order` of `2`. When any page in the *Pages/OtherPages* folder is requested with a route parameter (for example, `/OtherPages/Page1/RouteDataValue`), "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Razor Pages options, such as adding Conventions, are added when MVC is added to the service collection in `Startup.ConfigureServices`. For an example, see the sample app.

```
options.Conventions.Add(new GlobalTemplatePageRouteModelConvention());
```

Request the sample's About page at `localhost:5000/About/GlobalRouteValue` and inspect the result:



Add an app model convention to all pages

Use Conventions to create and add an IPageApplicationModelConvention to the collection of IPageConvention instances that are applied during page app model construction.

To demonstrate this and other conventions later in the topic, the sample app includes an `AddHeaderAttribute` class. The class constructor accepts a `name` string and a `values` string array. These values are used in its `OnResultExecuting` method to set a response header. The full class is shown in the Page model action conventions section later in the topic.

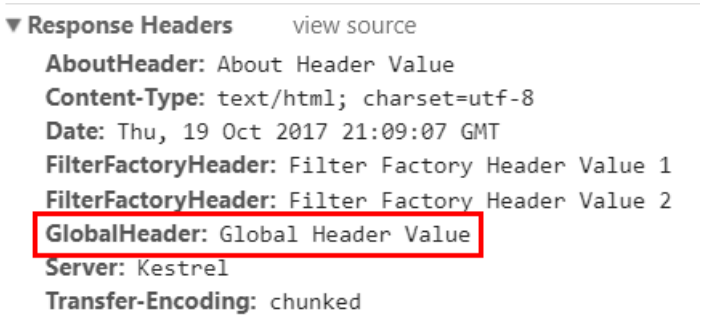
The sample app uses the `AddHeaderAttribute` class to add a header, `GlobalHeader`, to all of the pages in the app:

```
public class GlobalHeaderPageApplicationModelConvention
    : IPageApplicationModelConvention
{
    public void Apply(PageApplicationModel model)
    {
        model.Filters.Add(new AddHeaderAttribute(
            "GlobalHeader", new string[] { "Global Header Value" }));
    }
}
```

Startup.cs:

```
options.Conventions.Add(new GlobalHeaderPageApplicationModelConvention());
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



Add a handler model convention to all pages

Use Conventions to create and add an IPageHandlerModelConvention to the collection of IPageConvention instances that are applied during page handler model construction.

```
public class GlobalPageHandlerModelConvention
    : IPageHandlerModelConvention
{
    public void Apply(PageHandlerModel model)
    {
        // Access the PageHandlerModel
    }
}
```

Startup.cs:

```
options.Conventions.Add(new GlobalPageHandlerModelConvention());
```

Page route action conventions

The default route model provider that derives from IPageRouteModelProvider invokes conventions which are designed to provide extensibility points for configuring page routes.

Folder route model convention

Use AddFolderRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for all of the pages under the specified folder.

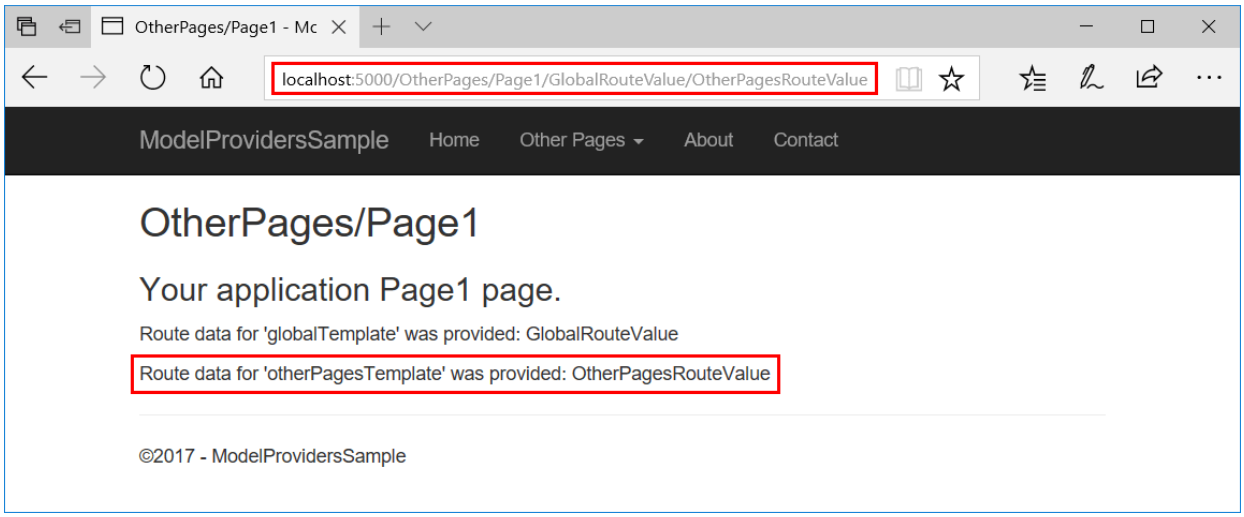
The sample app uses AddFolderRouteModelConvention to add an `{otherPagesTemplate?}` route template to the pages in the *OtherPages* folder:

```
options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{otherPagesTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If a page in the *Pages/OtherPages* folder is requested with a route parameter value (for example, `/OtherPages/Page1/RouteDataValue`), "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1/GlobalRouteValue/OtherPagesRouteValue` and inspect the result:



Page route model convention

Use AddPageRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for the page with the specified name.

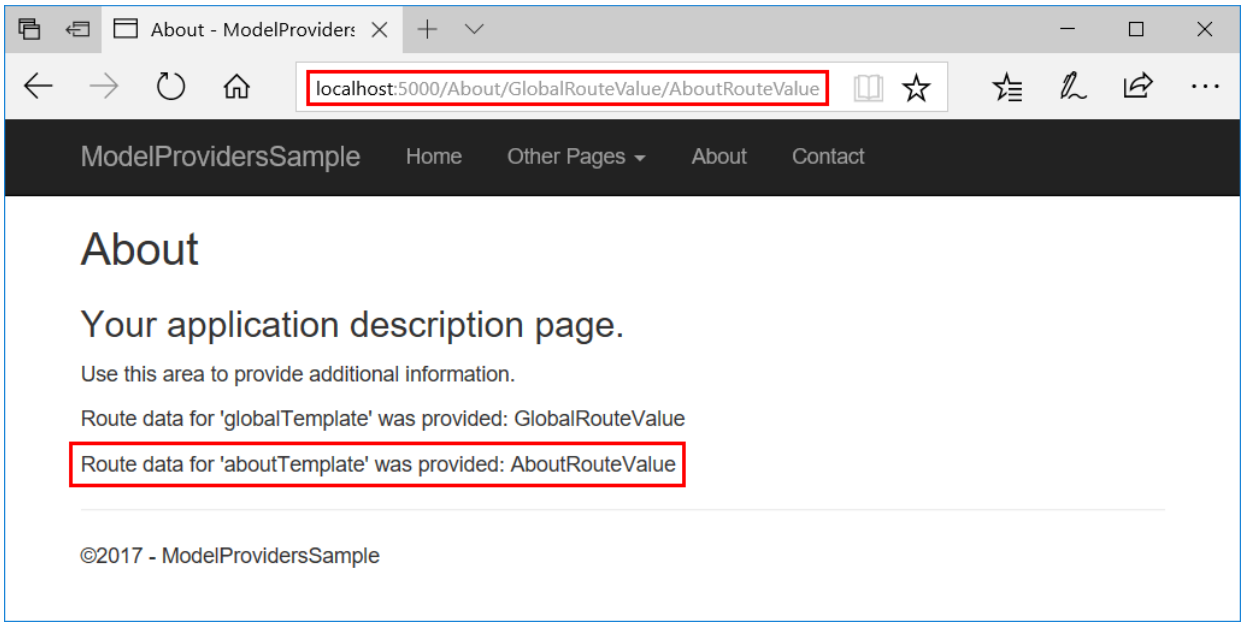
The sample app uses `AddPageRouteModelConvention` to add an `{aboutTemplate?}` route template to the About page:

```
options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{aboutTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If the About page is requested with a route parameter value at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's About page at `localhost:5000/About/GlobalRouteValue/AboutRouteValue` and inspect the result:



Use a parameter transformer to customize page routes

Page routes generated by ASP.NET Core can be customized using a parameter transformer. A parameter transformer implements `IOutboundParameterTransformer` and transforms the value of parameters. For example, a custom `SlugifyParameterTransformer` parameter transformer changes the `SubscriptionManagement` route value to `subscription-management`.

The `PageRouteTransformerConvention` page route model convention applies a parameter transformer to the folder and file name segments of automatically generated page routes in an app. For example, the Razor Pages file at */Pages/SubscriptionManagement/ViewAll.cshtml* would have its route rewritten from `/SubscriptionManagement/ViewAll` to `/subscription-management/view-all`.

`PageRouteTransformerConvention` only transforms the automatically generated segments of a page route that come from the Razor Pages folder and file name. It doesn't transform route segments added with the `@page` directive. The convention also doesn't transform routes added by AddPageRoute.

The `PageRouteTransformerConvention` is registered as an option in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.Conventions.Add(
                new PageRouteTransformerConvention(
                    new SlugifyParameterTransformer()));
        });
}

public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) { return null; }

        // Slugify value
        return Regex.Replace(value.ToString(), "([a-z])([A-Z])", "$1-$2").ToLower();
    }
}
```

Configure a page route

Use AddPageRoute to configure a route to a page at the specified page path. Generated links to the page use your specified route. `AddPageRoute` uses `AddPageRouteModelConvention` to establish the route.

The sample app creates a route to `/TheContactPage` for *Contact.cshtml*:

```
options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");
```

The Contact page can also be reached at `/Contact` via its default route.

The sample app's custom route to the Contact page allows for an optional `text` route segment (`{text?}`). The page also includes this optional segment in its `@page` directive in case the visitor accesses the page at its `/Contact` route:

```
@page "{text?}"
@model ContactModel
@{
    ViewData["Title"] = "Contact";
}

<h1>@ViewData["Title"]</h1>
<h2>@Model.Message</h2>

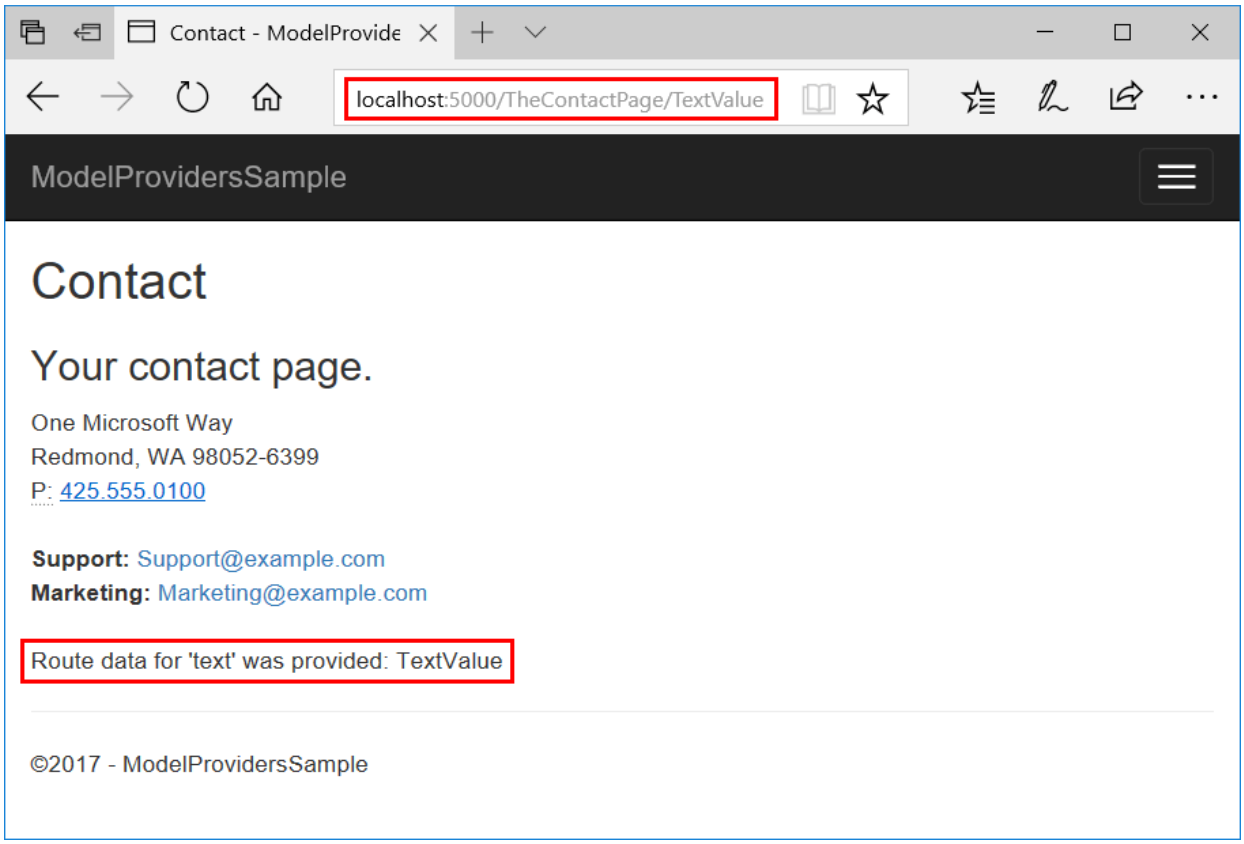
<address>
    One Microsoft Way<br>
    Redmond, WA 98052-6399<br>
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a href="mailto:Support@example.com">Support@example.com</a><br>
    <strong>Marketing:</strong> <a href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>

<p>@Model.RouteDataTextTemplateValue</p>
```

Note that the URL generated for the **Contact** link in the rendered page reflects the updated route:

Visit the Contact page at either its ordinary route, `/Contact` , or the custom route, `/TheContactPage` . If you supply an additional `text` route segment, the page shows the HTML-encoded segment that you provide:



Page model action conventions

The default page model provider that implements IPageApplicationModelProvider invokes conventions which are designed to provide extensibility points for configuring page models. These conventions are useful when building and modifying page discovery and processing scenarios.

For the examples in this section, the sample app uses an `AddHeaderAttribute` class, which is a ResultFilterAttribute, that applies a response header:

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string[] _values;

    public AddHeaderAttribute(string name, string[] values)
    {
        _name = name;
        _values = values;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _values);
        base.OnResultExecuting(context);
    }
}
```

Using conventions, the sample demonstrates how to apply the attribute to all of the pages in a folder and to a single page.

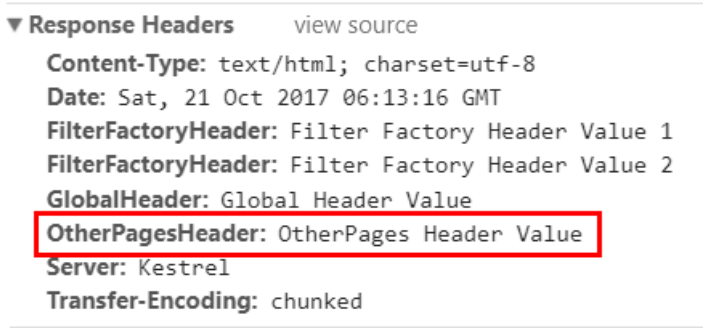
Folder app model convention

Use AddFolderApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on PageApplicationModel instances for all pages under the specified folder.

The sample demonstrates the use of `AddFolderApplicationModelConvention` by adding a header, `OtherPagesHeader`, to the pages inside the *OtherPages* folder of the app:

```
options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "OtherPagesHeader", new string[] { "OtherPages Header Value" }));
});
```

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1` and inspect the headers to view the result:



Page app model convention

Use AddPageApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on the PageApplicationModel for the page with the specified name.

The sample demonstrates the use of `AddPageApplicationModelConvention` by adding a header, `AboutHeader`, to the About page:

```
options.Conventions.AddPageApplicationModelConvention("/About", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "AboutHeader", new string[] { "About Header Value" }));
});
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:

Configure a filter

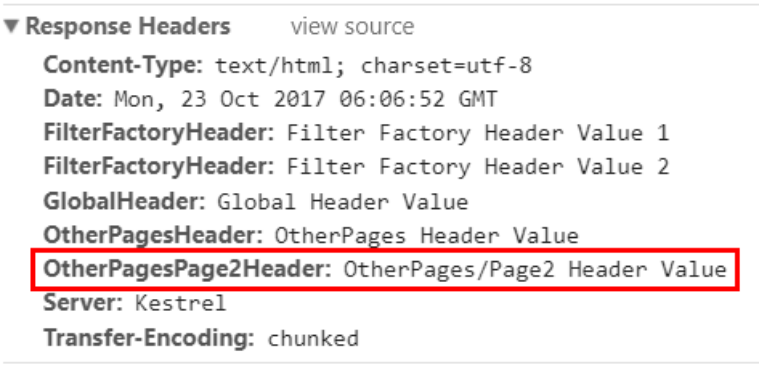
ConfigureFilter configures the specified filter to apply. You can implement a filter class, but the sample app shows how to implement a filter in a lambda expression, which is implemented behind-the-scenes as a factory that returns a filter:

```
options.Conventions.ConfigureFilter(model =>
{
    if (model.RelativePath.Contains("OtherPages/Page2"))
    {
        return new AddHeaderAttribute(
            "OtherPagesPage2Header",
            new string[] { "OtherPages/Page2 Header Value" });
    }
    return new EmptyFilter();
});
```

The page app model is used to check the relative path for segments that lead to the Page2 page in the *OtherPages* folder. If the condition passes, a header is added. If not, the `EmptyFilter` is applied.

`EmptyFilter` is an Action filter. Since Action filters are ignored by Razor Pages, the `EmptyFilter` has no effect as intended if the path doesn't contain `OtherPages/Page2`.

Request the sample's Page2 page at `localhost:5000/OtherPages/Page2` and inspect the headers to view the result:



Configure a filter factory

ConfigureFilter configures the specified factory to apply filters to all Razor Pages.

The sample app provides an example of using a filter factory by adding a header, `FilterFactoryHeader`, with two values to the app's pages:

```
options.Conventions.ConfigureFilter(new AddHeaderWithFactory());
```

AddHeaderWithFactory.cs:

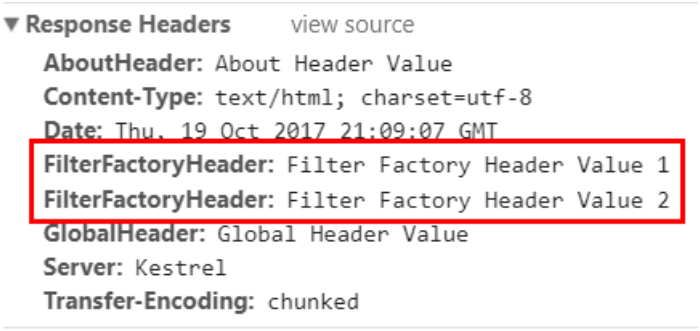
```
public class AddHeaderWithFactory : IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new AddHeaderFilter();
    }

    private class AddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "FilterFactoryHeader",
                new string[]
                {
                    "Filter Factory Header Value 1",
                    "Filter Factory Header Value 2"
                });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



MVC Filters and the Page filter (IPageFilter)

MVC Action filters are ignored by Razor Pages, since Razor Pages use handler methods. Other types of MVC filters are available for you to use: Authorization, Exception, Resource, and Result. For more information, see the Filters topic.

The Page filter (IPageFilter) is a filter that applies to Razor Pages. For more information, see Filter methods for Razor Pages.

Additional resources

- Razor Pages authorization conventions in ASP.NET Core
- Areas in ASP.NET Core

Learn how to use page route and app model provider conventions to control page routing, discovery, and processing in Razor Pages apps.

When you need to configure custom page routes for individual pages, configure routing to pages with the AddPageRoute convention described later in this topic.

To specify a page route, add route segments, or add parameters to a route, use the page's `@page` directive. For more information, see Custom routes.

There are reserved words that can't be used as route segments or parameter names. For more information, see Routing: Reserved routing names.

View or download sample code (how to download)

SCENARIO	THE SAMPLE DEMONSTRATES ...
Model conventions Conventions.Add • IPageRouteModelConvention • IPageApplicationModelConvention • IPageHandlerModelConvention	Add a route template and header to an app's pages.
Page route action conventions • AddFolderRouteModelConvention • AddPageRouteModelConvention • AddPageRoute	Add a route template to pages in a folder and to a single page.
Page model action conventions • AddFolderApplicationModelConvention • AddPageApplicationModelConvention • ConfigureFilter (filter class, lambda expression, or filter factory)	Add a header to pages in a folder, add a header to a single page, and configure a filter factory to add a header to an app's pages.

Razor Pages conventions are added and configured using the AddRazorPagesOptions extension method to AddMvc on the service collection in the `Startup` class. The following convention examples are explained later in this topic:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.Conventions.Add( ... );
            options.Conventions.AddFolderRouteModelConvention(
                "/OtherPages", model => { ... });
            options.Conventions.AddPageRouteModelConvention(
                "/About", model => { ... });
            options.Conventions.AddPageRoute(
                "/Contact", "TheContactPage/{text?}");
            options.Conventions.AddFolderApplicationModelConvention(
                "/OtherPages", model => { ... });
            options.Conventions.AddPageApplicationModelConvention(
                "/About", model => { ... });
            options.Conventions.ConfigureFilter(model => { ... });
            options.Conventions.ConfigureFilter( ... );
        });
}
```

Route order

Routes specify an Order for processing (route matching).

ORDER	BEHAVIOR
-1	The route is processed before other routes are processed.
0	Order isn't specified (default value). Not assigning `Order` (`Order = null`) defaults the route `Order` to 0 (zero) for processing.
1, 2, ... n	Specifies the route processing order.

Route processing is established by convention:

- Routes are processed in sequential order (-1, 0, 1, 2, ... n).
- When routes have the same `Order`, the most specific route is matched first followed by less specific routes.
- When routes with the same `Order` and the same number of parameters match a request URL, routes are processed in the order that they're added to the PageConventionCollection.

If possible, avoid depending on an established route processing order. Generally, routing selects the correct route with URL matching. If you must set route `Order` properties to route requests correctly, the app's routing scheme is probably confusing to clients and fragile to maintain. Seek to simplify the app's routing scheme. The sample app requires an explicit route processing order to demonstrate several routing scenarios using a single app. However, you should attempt to avoid the practice of setting route `Order` in production apps.

Razor Pages routing and MVC controller routing share an implementation. Information on route order in the MVC topics is available at Routing to controller actions: Ordering attribute routes.

Model conventions

Add a delegate for IPageConvention to add model conventions that apply to Razor Pages.

Add a route model convention to all pages

Use Conventions to create and add an IPageRouteModelConvention to the collection of IPageConvention

instances that are applied during page route model construction.

The sample app adds a `{globalTemplate?}` route template to all of the pages in the app:

```
public class GlobalTemplatePageRouteModelConvention
    : IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 1,
                    Template = AttributeRouteModel.CombineTemplates(
                        selector.AttributeRouteModel.Template,
                        "{globalTemplate?}"),
                }
            });
        }
    }
}
```

The Order property for the AttributeRouteModel is set to `1`. This ensures the following route matching behavior in the sample app:

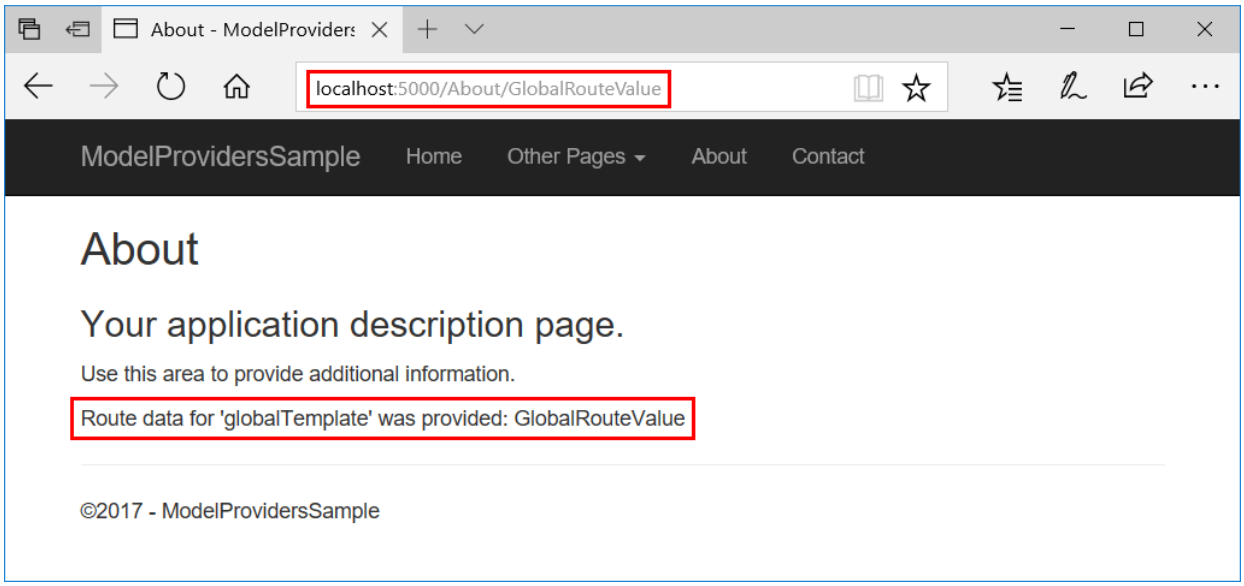
- A route template for `TheContactPage/{text?}` is added later in the topic. The Contact Page route has a default order of `null` (`Order = 0`), so it matches before the `{globalTemplate?}` route template.
- An `{aboutTemplate?}` route template is added later in the topic. The `{aboutTemplate?}` template is given an `Order` of `2`. When the About page is requested at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.
- An `{otherPagesTemplate?}` route template is added later in the topic. The `{otherPagesTemplate?}` template is given an `Order` of `2`. When any page in the *Pages/OtherPages* folder is requested with a route parameter (for example, `/OtherPages/Page1/RouteDataValue`), "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Razor Pages options, such as adding Conventions, are added when MVC is added to the service collection in `Startup.ConfigureServices`. For an example, see the sample app.

```
options.Conventions.Add(new GlobalTemplatePageRouteModelConvention());
```

Request the sample's About page at `localhost:5000/About/GlobalRouteValue` and inspect the result:



Add an app model convention to all pages

Use Conventions to create and add an IPageApplicationModelConvention to the collection of IPageConvention instances that are applied during page app model construction.

To demonstrate this and other conventions later in the topic, the sample app includes an `AddHeaderAttribute` class. The class constructor accepts a `name` string and a `values` string array. These values are used in its `OnResultExecuting` method to set a response header. The full class is shown in the Page model action conventions section later in the topic.

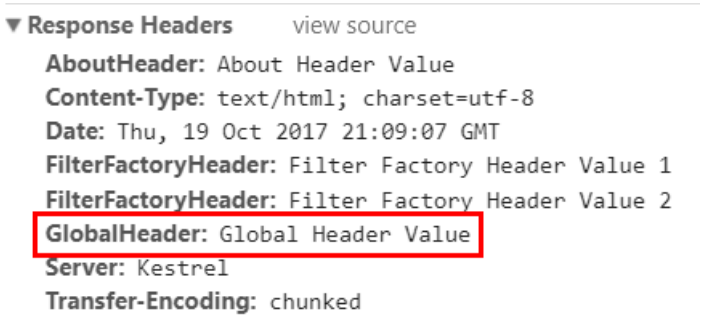
The sample app uses the `AddHeaderAttribute` class to add a header, `GlobalHeader`, to all of the pages in the app:

```
public class GlobalHeaderPageApplicationModelConvention
    : IPageApplicationModelConvention
{
    public void Apply(PageApplicationModel model)
    {
        model.Filters.Add(new AddHeaderAttribute(
            "GlobalHeader", new string[] { "Global Header Value" }));
    }
}
```

Startup.cs:

```
options.Conventions.Add(new GlobalHeaderPageApplicationModelConvention());
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



Add a handler model convention to all pages

Use Conventions to create and add an IPageHandlerModelConvention to the collection of IPageConvention instances that are applied during page handler model construction.

```
public class GlobalPageHandlerModelConvention
    : IPageHandlerModelConvention
{
    public void Apply(PageHandlerModel model)
    {
        // Access the PageHandlerModel
    }
}
```

Startup.cs:

```
options.Conventions.Add(new GlobalPageHandlerModelConvention());
```

Page route action conventions

The default route model provider that derives from IPageRouteModelProvider invokes conventions which are designed to provide extensibility points for configuring page routes.

Folder route model convention

Use AddFolderRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for all of the pages under the specified folder.

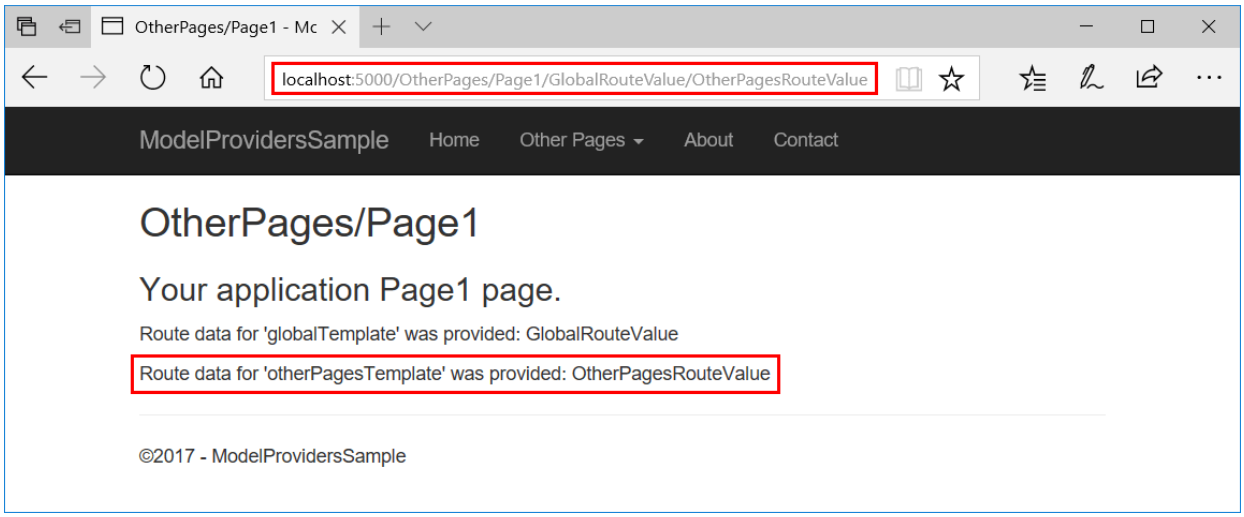
The sample app uses AddFolderRouteModelConvention to add an `{otherPagesTemplate?}` route template to the pages in the *OtherPages* folder:

```
options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{otherPagesTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If a page in the *Pages/OtherPages* folder is requested with a route parameter value (for example, `/OtherPages/Page1/RouteDataValue`), "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1/GlobalRouteValue/OtherPagesRouteValue` and inspect the result:



Page route model convention

Use AddPageRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for the page with the specified name.

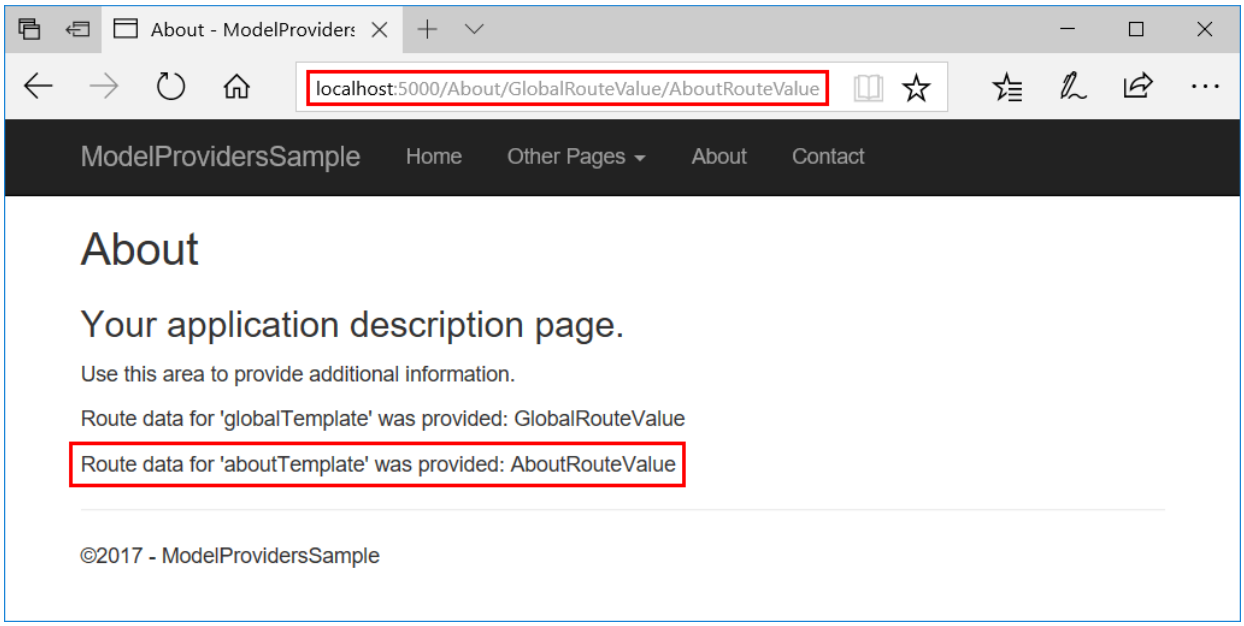
The sample app uses `AddPageRouteModelConvention` to add an `{aboutTemplate?}` route template to the About page:

```
options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel.Template,
                    "{aboutTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If the About page is requested with a route parameter value at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's About page at `localhost:5000/About/GlobalRouteValue/AboutRouteValue` and inspect the result:



Configure a page route

Use AddPageRoute to configure a route to a page at the specified page path. Generated links to the page use your specified route. `AddPageRoute` uses `AddPageRouteModelConvention` to establish the route.

The sample app creates a route to `/TheContactPage` for *Contact.cshtml*:

```
options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");
```

The Contact page can also be reached at `/Contact` via its default route.

The sample app's custom route to the Contact page allows for an optional `text` route segment (`{text?}`). The page also includes this optional segment in its `@page` directive in case the visitor accesses the page at its `/Contact` route:

```
@page "{text?}"
@model ContactModel
@{
    ViewData["Title"] = "Contact";
}

<h1>@ViewData["Title"]</h1>
<h2>@Model.Message</h2>

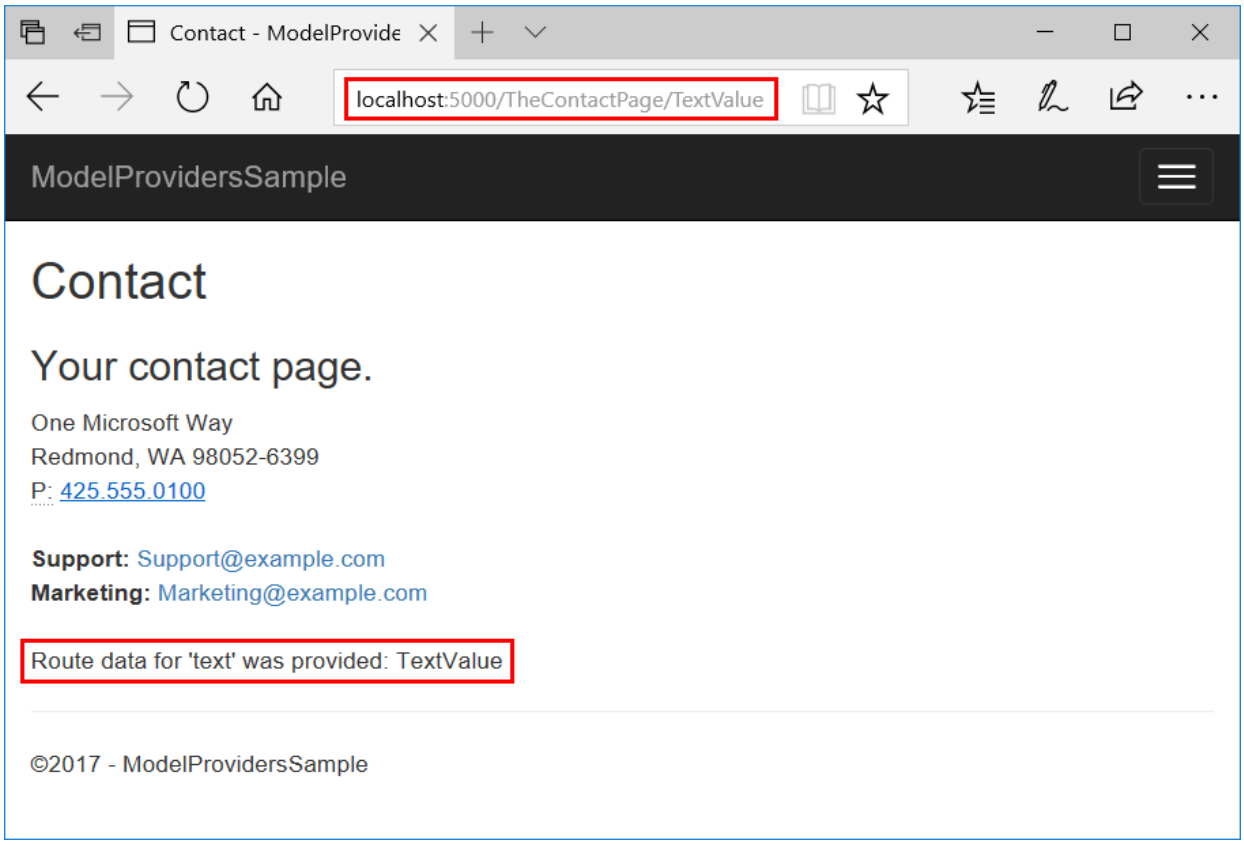
<address>
    One Microsoft Way<br>
    Redmond, WA 98052-6399<br>
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a href="mailto:Support@example.com">Support@example.com</a><br>
    <strong>Marketing:</strong> <a href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>

<p>@Model.RouteDataTextTemplateValue</p>
```

Note that the URL generated for the **Contact** link in the rendered page reflects the updated route:

Visit the Contact page at either its ordinary route, `/Contact`, or the custom route, `/TheContactPage`. If you supply an additional `text` route segment, the page shows the HTML-encoded segment that you provide:



Page model action conventions

The default page model provider that implements IPageApplicationModelProvider invokes conventions which are designed to provide extensibility points for configuring page models. These conventions are useful when building and modifying page discovery and processing scenarios.

For the examples in this section, the sample app uses an `AddHeaderAttribute` class, which is a ResultFilterAttribute, that applies a response header:

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string[] _values;

    public AddHeaderAttribute(string name, string[] values)
    {
        _name = name;
        _values = values;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _values);
        base.OnResultExecuting(context);
    }
}
```

Using conventions, the sample demonstrates how to apply the attribute to all of the pages in a folder and to a single page.

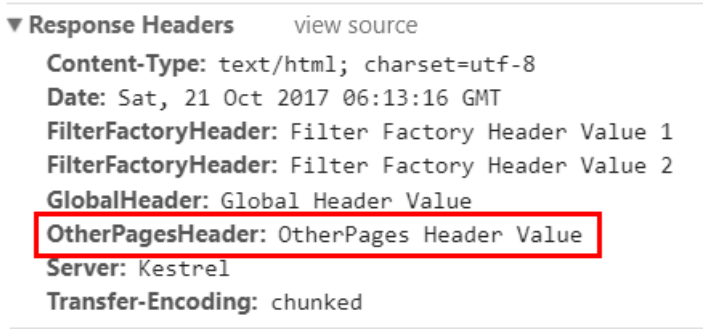
Folder app model convention

Use AddFolderApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on PageApplicationModel instances for all pages under the specified folder.

The sample demonstrates the use of `AddFolderApplicationModelConvention` by adding a header, `OtherPagesHeader`, to the pages inside the *OtherPages* folder of the app:

```
options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "OtherPagesHeader", new string[] { "OtherPages Header Value" }));
});
```

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1` and inspect the headers to view the result:



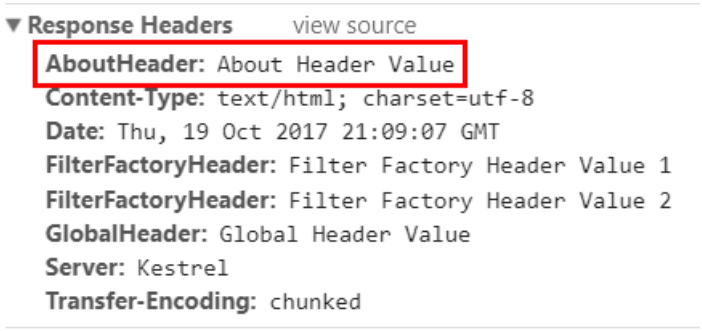
Page app model convention

Use AddPageApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on the PageApplicationModel for the page with the specified name.

The sample demonstrates the use of `AddPageApplicationModelConvention` by adding a header, `AboutHeader`, to the About page:

```
options.Conventions.AddPageApplicationModelConvention("/About", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "AboutHeader", new string[] { "About Header Value" }));
});
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



Configure a filter

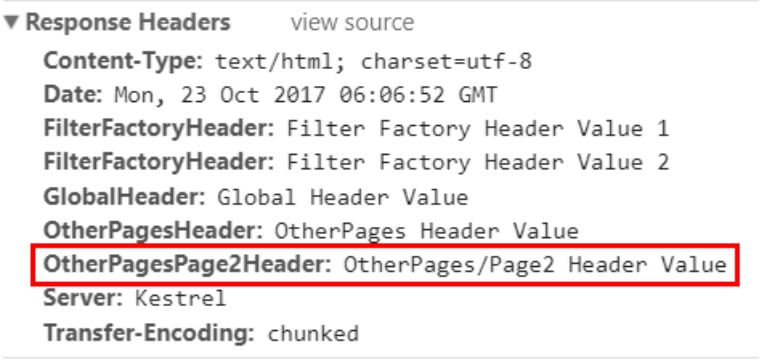
ConfigureFilter configures the specified filter to apply. You can implement a filter class, but the sample app shows how to implement a filter in a lambda expression, which is implemented behind-the-scenes as a factory that returns a filter:

```
options.Conventions.ConfigureFilter(model =>
{
    if (model.RelativePath.Contains("OtherPages/Page2"))
    {
        return new AddHeaderAttribute(
            "OtherPagesPage2Header",
            new string[] { "OtherPages/Page2 Header Value" });
    }
    return new EmptyFilter();
});
```

The page app model is used to check the relative path for segments that lead to the Page2 page in the *OtherPages* folder. If the condition passes, a header is added. If not, the `EmptyFilter` is applied.

`EmptyFilter` is an Action filter. Since Action filters are ignored by Razor Pages, the `EmptyFilter` has no effect as intended if the path doesn't contain `OtherPages/Page2`.

Request the sample's Page2 page at `localhost:5000/OtherPages/Page2` and inspect the headers to view the result:



Configure a filter factory

ConfigureFilter configures the specified factory to apply filters to all Razor Pages.

The sample app provides an example of using a filter factory by adding a header, `FilterFactoryHeader`, with two

values to the app's pages:

```
options.Conventions.ConfigureFilter(new AddHeaderWithFactory());
```

AddHeaderWithFactory.cs:

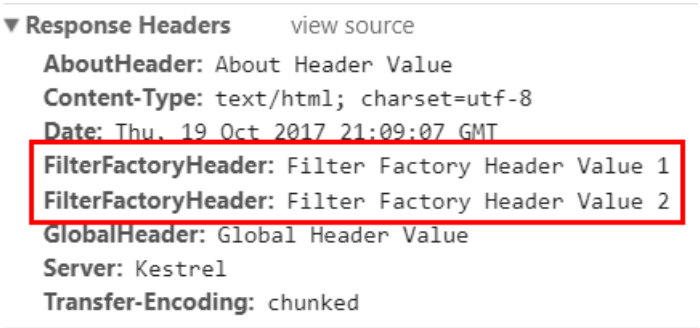
```
public class AddHeaderWithFactory : IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new AddHeaderFilter();
    }

    private class AddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "FilterFactoryHeader",
                new string[]
                {
                    "Filter Factory Header Value 1",
                    "Filter Factory Header Value 2"
                });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



MVC Filters and the Page filter (IPageFilter)

MVC Action filters are ignored by Razor Pages, since Razor Pages use handler methods. Other types of MVC filters are available for you to use: Authorization, Exception, Resource, and Result. For more information, see the Filters topic.

The Page filter (IPageFilter) is a filter that applies to Razor Pages. For more information, see Filter methods for Razor Pages.

Additional resources

- Razor Pages authorization conventions in ASP.NET Core
- Areas in ASP.NET Core

Overview of ASP.NET Core MVC

9/22/2020 • 9 minutes to read • Edit Online

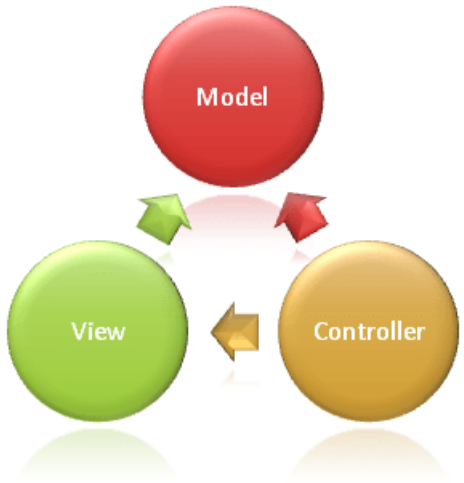
By Steve Smith

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

What is the MVC pattern?

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job. It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, an object containing business logic must be modified every time the user interface is changed. This often introduces errors and requires the retesting of business logic after every minimal user interface change.

> **NOTE**
> Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views typically use ViewModel

types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

View Responsibilities

Views are responsible for presenting content through the user interface. They use the Razor view engine to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a View Component, ViewModel, or view template to simplify the view.

Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

> **NOTE**
> Controllers shouldn't be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, push business logic out of the controller and into the domain model.

> **TIP**
> If you find that your controller actions frequently perform the same kinds of actions, move these common actions into filters.

What is ASP.NET Core MVC

The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.

ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns. It gives you full control over markup, supports TDD-friendly development and uses the latest web standards.

Features

ASP.NET Core MVC includes the following:

- Routing
- Model binding
- Model validation
- Dependency injection
- Filters
- Areas
- Web APIs
- Testability
- Razor view engine
- Strongly typed views
- Tag Helpers

- View Components

Routing

ASP.NET Core MVC is built on top of ASP.NET Core's routing, a powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs. This enables you to define your application's URL naming patterns that work well for search engine optimization (SEO) and for link generation, without regard for how the files on your web server are organized. You can define your routes using a convenient route template syntax that supports route value constraints, defaults and optional values.

Convention-based routing enables you to globally define the URL formats that your application accepts and how each of those formats maps to a specific action method on given controller. When an incoming request is received, the routing engine parses the URL and matches it to one of the defined URL formats, and then calls the associated controller's action method.

```
routes.MapRoute(name: "Default", template: "{controller=Home}/{action=Index}/{id?}");
```

Attribute routing enables you to specify routing information by decorating your controllers and actions with attributes that define your application's routes. This means that your route definitions are placed next to the controller and action with which they're associated.

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
      ...
    }
}
```

Model binding

ASP.NET Core MVC model binding converts client request data (form values, route data, query string parameters, HTTP headers) into objects that the controller can handle. As a result, your controller logic doesn't have to do the work of figuring out the incoming request data; it simply has the data as parameters to its action methods.

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null) { ... }
```

Model validation

ASP.NET Core MVC supports validation by decorating your model object with data annotation validation attributes. The validation attributes are checked on the client side before values are posted to the server, as well as on the server before the controller action is called.

```
using System.ComponentModel.DataAnnotations;
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}
```

A controller action:

```
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    if (ModelState.IsValid)
    {
      // work with the model
    }
    // At this point, something failed, redisplay form
    return View(model);
}
```

The framework handles validating request data both on the client and on the server. Validation logic specified on model types is added to the rendered views as unobtrusive annotations and is enforced in the browser with jQuery Validation.

Dependency injection

ASP.NET Core has built-in support for dependency injection (DI). In ASP.NET Core MVC, controllers can request needed services through their constructors, allowing them to follow the Explicit Dependencies Principle.

Your app can also use dependency injection in view files, using the `@inject` directive:

```
@inject SomeService ServiceName

<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ServiceName.GetTitle</title>
</head>
<body>
    <h1>@ServiceName.GetTitle</h1>
</body>
</html>
```

Filters

Filters help developers encapsulate cross-cutting concerns, like exception handling or authorization. Filters enable running custom pre- and post-processing logic for action methods, and can be configured to run at certain points within the execution pipeline for a given request. Filters can be applied to controllers or actions as attributes (or can be run globally). Several filters (such as `Authorize`) are included in the framework. `[Authorize]` is the attribute that is used to create MVC authorization filters.

```
[Authorize]
public class AccountController : Controller
```

Areas

Areas provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is an MVC structure inside an application. In an MVC project, logical components like Model, Controller, and View are kept in different folders, and MVC uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search etc. Each of these units have their own logical component views, controllers, and models.

Web APIs

In addition to being a great platform for building web sites, ASP.NET Core MVC has great support for building Web APIs. You can build services that reach a broad range of clients including browsers and mobile devices.

The framework includes support for HTTP content-negotiation with built-in support to format data as JSON or XML. Write custom formatters to add support for your own formats.

Use link generation to enable support for hypermedia. Easily enable support for cross-origin resource sharing (CORS) so that your Web APIs can be shared across multiple Web applications.

Testability

The framework's use of interfaces and dependency injection make it well-suited to unit testing, and the framework includes features (like a TestHost and InMemory provider for Entity Framework) that make integration tests quick and easy as well. Learn more about how to test controller logic.

Razor view engine

ASP.NET Core MVC views use the Razor view engine to render views. Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code. Razor is used to dynamically generate web content on the server. You can cleanly mix server code with client side content and code.

```
<ul>
    @for (int i = 0; i < 5; i++) {
        <li>List item @i</li>
    }
</ul>
```

Using the Razor view engine you can define layouts, partial views and replaceable sections.

Strongly typed views

Razor views in MVC can be strongly typed based on your model. Controllers can pass a strongly typed model to views enabling your views to have type checking and IntelliSense support.

For example, the following view renders a model of type `IEnumerable<Product>`:

```
@model IEnumerable<Product>
<ul>
    @foreach (Product p in Model)
    {
        <li>@p.Name</li>
    }
</ul>
```

Tag Helpers

Tag Helpers enable server side code to participate in creating and rendering HTML elements in Razor files. You

can use tag helpers to define custom tags (for example, `<environment>`) or to modify the behavior of existing tags (for example, `<label>`). Tag Helpers bind to specific elements based on the element name and its attributes. They provide the benefits of server-side rendering while still preserving an HTML editing experience.

There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in LinkTagHelper can be used to create a link to the `Login` action of the `AccountsController`:

```
<p>
    Thank you for confirming your email.
    Please <a asp-controller="Account" asp-action="Login">Click here to Log in</a>.
</p>
```

The `EnvironmentTagHelper` can be used to include different scripts in your views (for example, raw or minified) based on the runtime environment, such as Development, Staging, or Production:

```
<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.min.js"
            asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
            asp-fallback-test="window.jQuery">
    </script>
</environment>
```

Tag Helpers provide an HTML-friendly development experience and a rich IntelliSense environment for creating HTML and Razor markup. Most of the built-in Tag Helpers target existing HTML elements and provide server-side attributes for the element.

View Components

View Components allow you to package rendering logic and reuse it throughout the application. They're similar to partial views, but with associated logic.

Compatibility version

The SetCompatibilityVersion method allows an app to opt-in or opt-out of potentially breaking behavior changes introduced in ASP.NET Core MVC 2.1 or later.

For more information, see Compatibility version for ASP.NET Core MVC.

Additional resources

- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)
- Integrate ASP.NET Core Razor components into Razor Pages and MVC apps

Create a web app with ASP.NET Core MVC

9/22/2020 • 2 minutes to read • Edit Online

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point.

The tutorial series includes the following:

1. Get started
2. Add a controller
3. Add a view
4. Add a model
5. Work with SQL Server LocalDB
6. Controller methods and views
7. Add search
8. Add a new field
9. Add validation
10. Examine the Details and Delete methods

Get started with ASP.NET Core MVC

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point.

This tutorial teaches the basics of building an ASP.NET Core MVC web app.

The app manages a database of movie titles. You learn how to:

- Create a web app.
- Add and scaffold a model.
- Work with a database.
- Add search and validation.

At the end, you have an app that can manage and display movie data.

View or download sample code (how to download).

Prerequisites

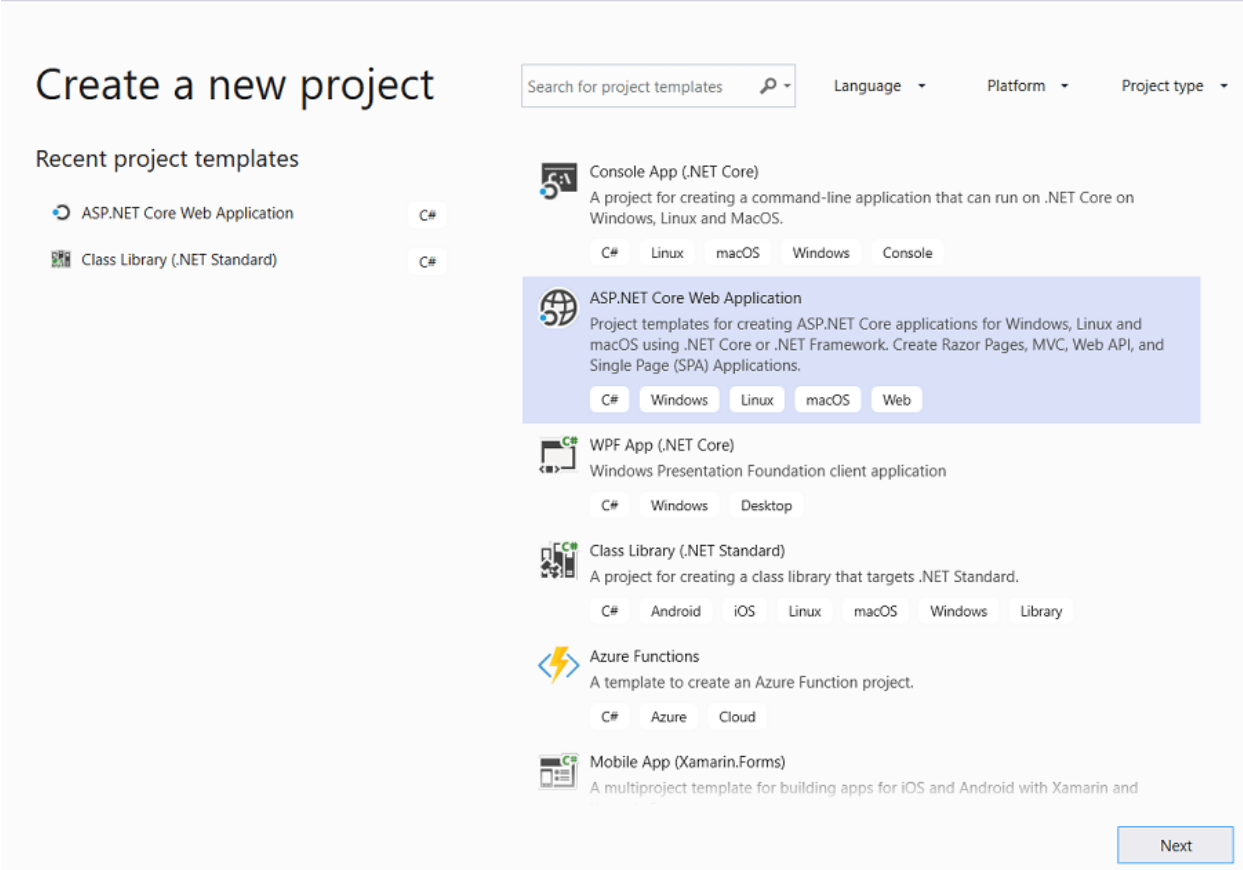
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

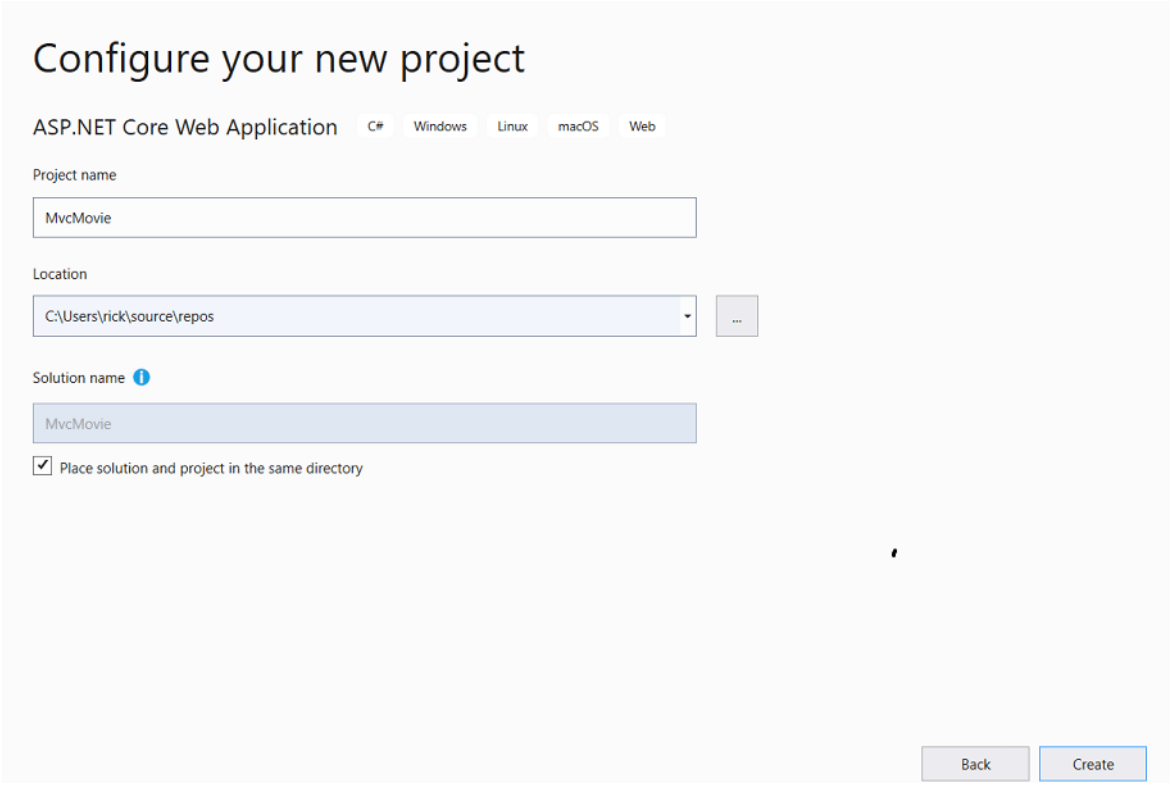
Create a web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

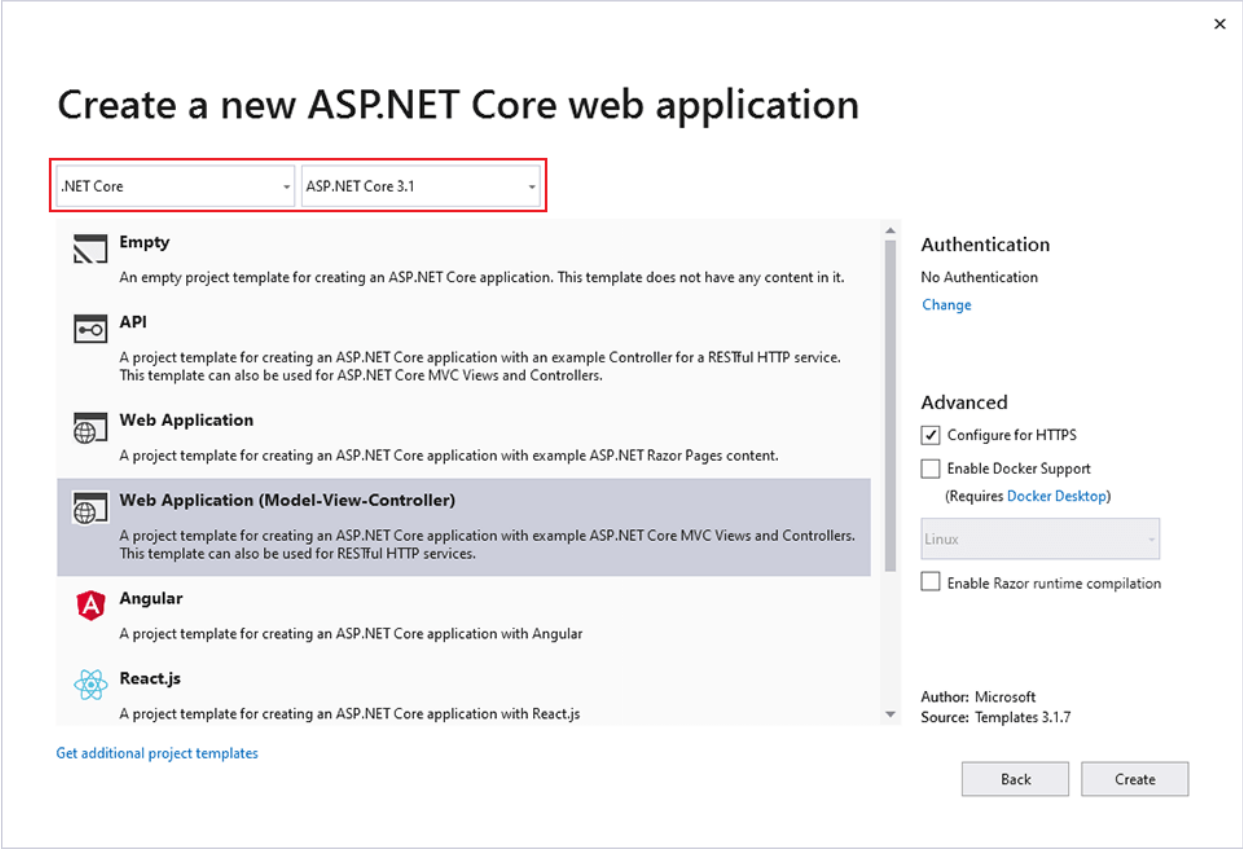
- From the Visual Studio select **Create a new project**.
- Select **ASP.NET Core Web Application** and then select **Next**.



- Name the project **MvcMovie** and select **Create**. It's important to name the project **MvcMovie** so when you copy code, the namespace will match.



- Select **Web Application(Model-View-Controller)**, and then select **Create**.



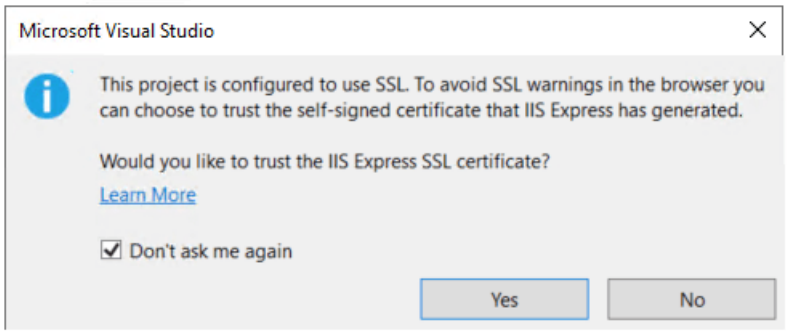
Visual Studio used the default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a basic starter project.

Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

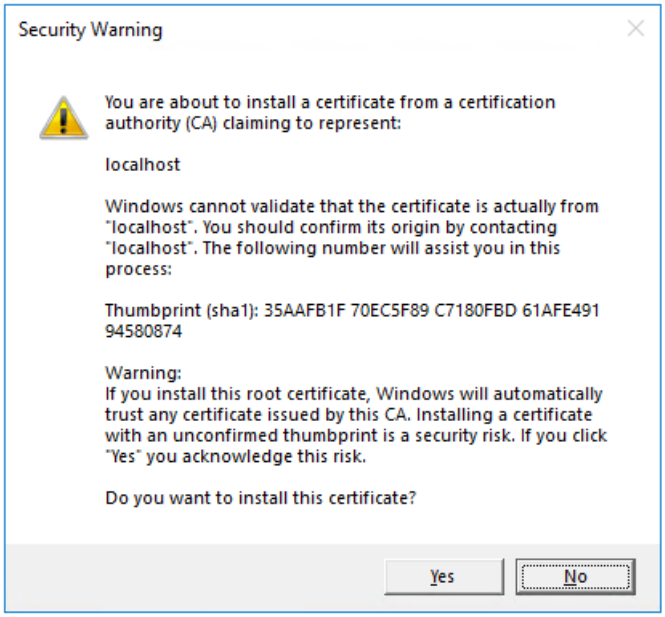
Select **Ctrl-F5** to run the app in non-debug mode.

Visual Studio displays the following dialog:



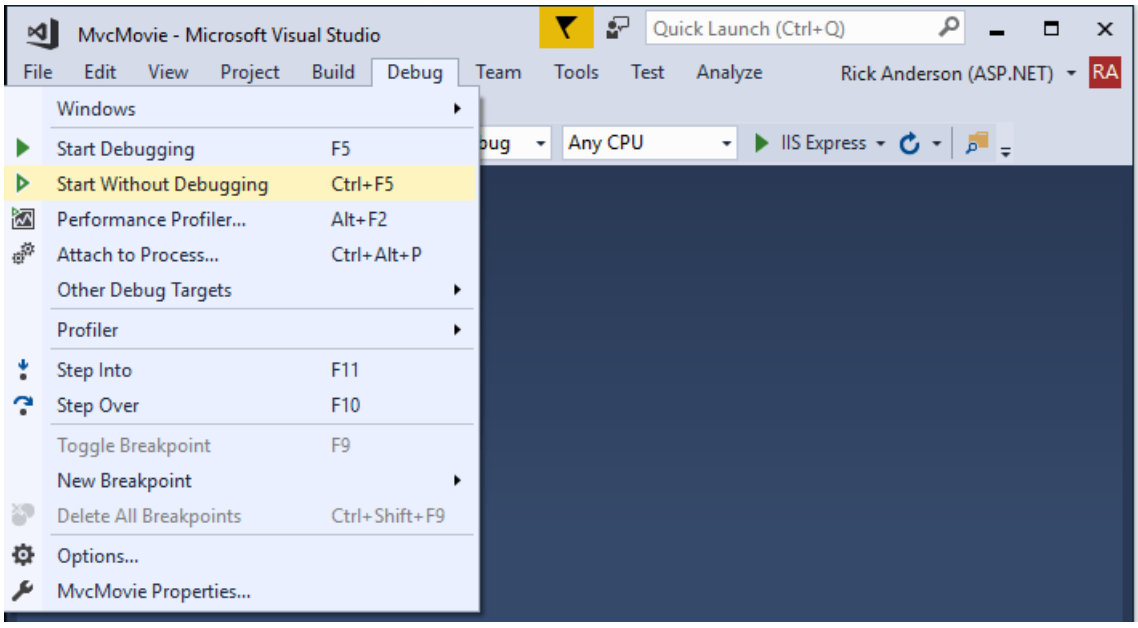
Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:

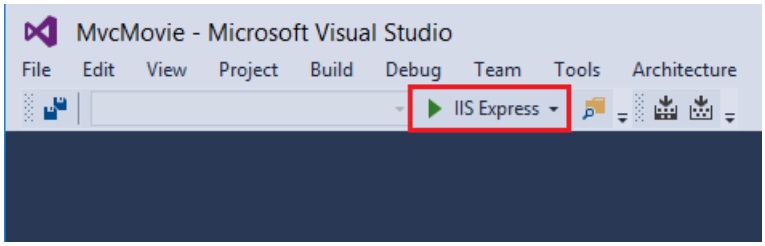


Select **Yes** if you agree to trust the development certificate.

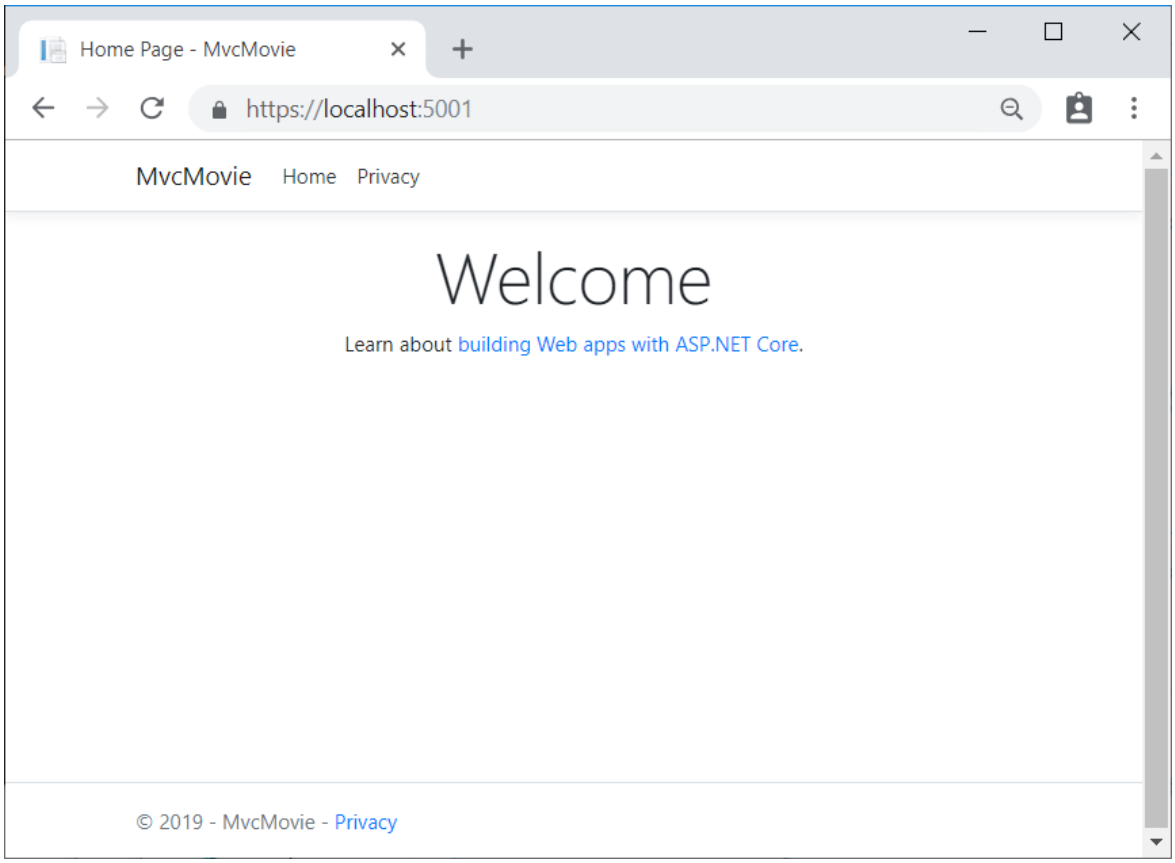
- Visual Studio starts IIS Express and runs the app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for your local computer. When Visual Studio creates a web project, a random port is used for the web server.
- Launching the app with Ctrl+F5 (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
- You can launch the app in debug or non-debug mode from the **Debug** menu item:



- You can debug the app by selecting the **IIS Express** button



The following image shows the app:



- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next part of this tutorial, you learn about MVC and start writing some code.

NEXT

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point.

This tutorial teaches the basics of building an ASP.NET Core MVC web app.

The app manages a database of movie titles. You learn how to:

- Create a web app.
- Add and scaffold a model.
- Work with a database.
- Add search and validation.

At the end, you have an app that can manage and display movie data.

View or download sample code (how to download).

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

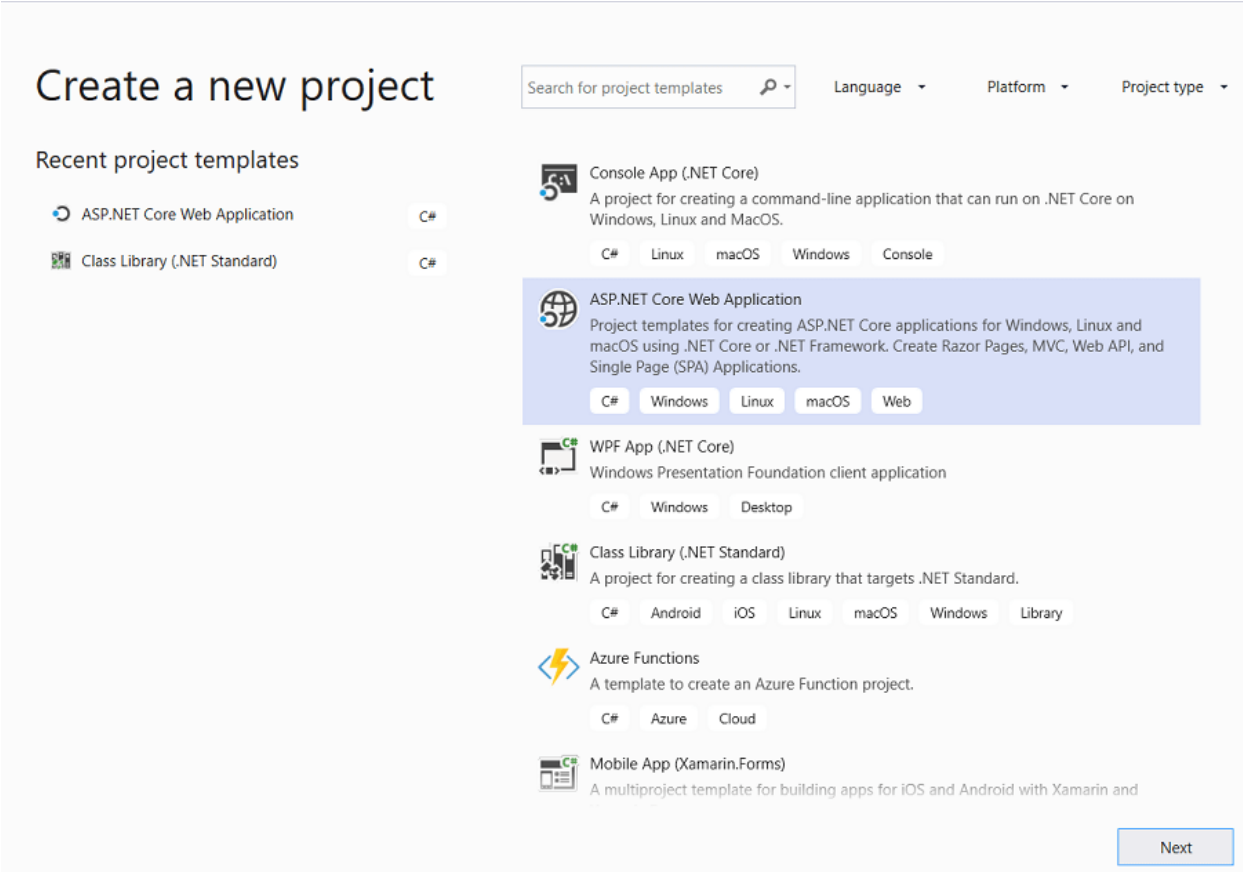
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

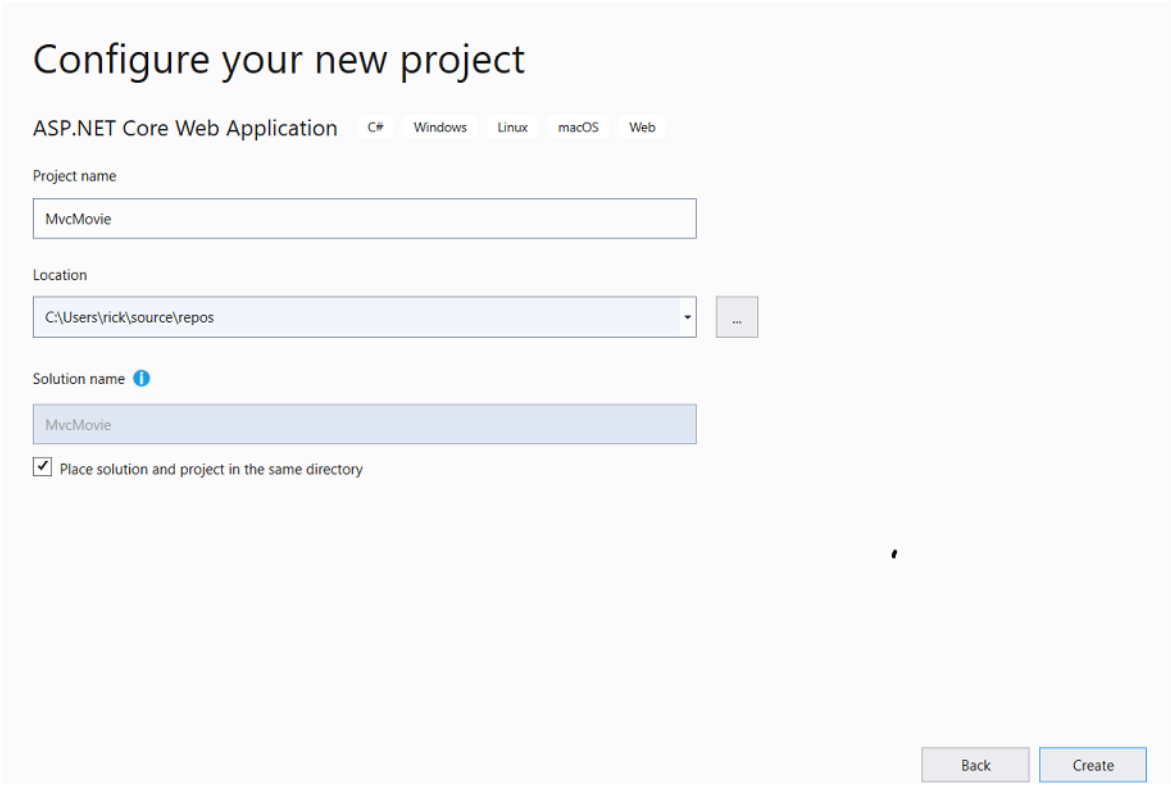
Create a web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

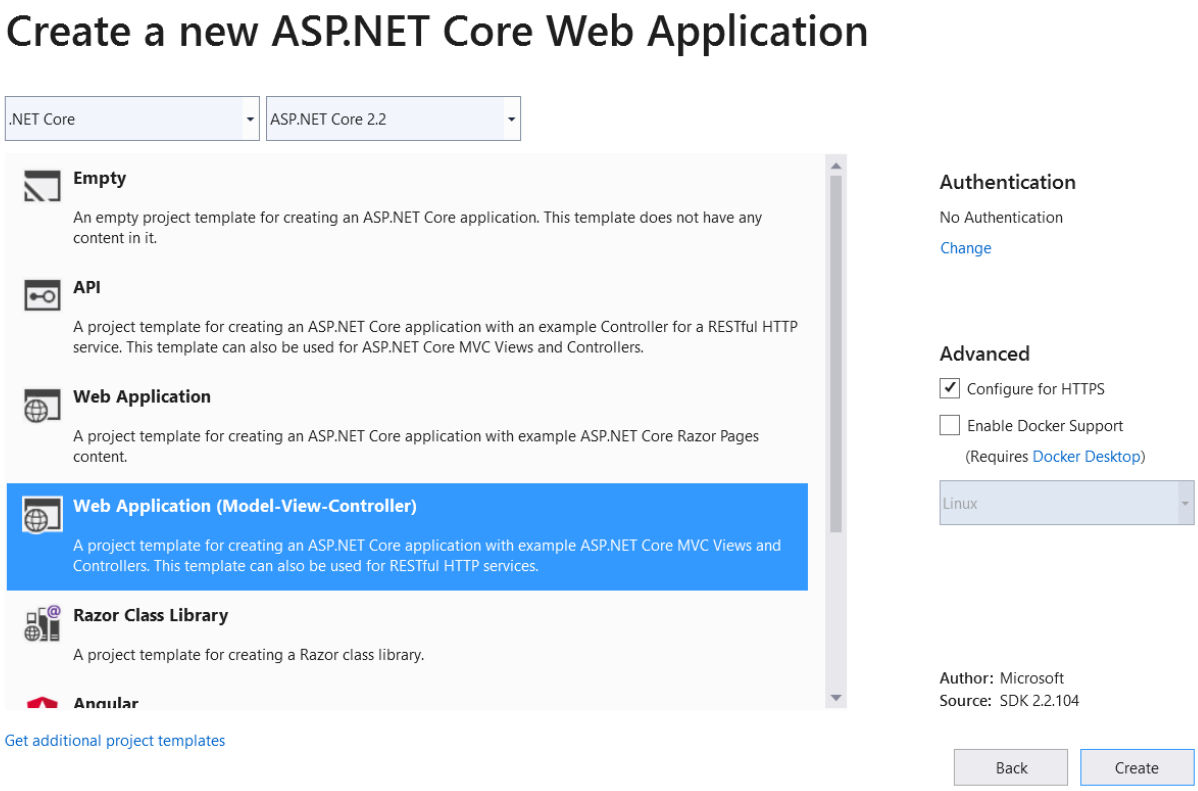
- From the Visual Studio select **Create a new project**.
- Select **ASP.NET Core Web Application** and then select **Next**.



- Name the project **MvcMovie** and select **Create**. It's important to name the project **MvcMovie** so when you copy code, the namespace will match.



- Select **Web Application(Model-View-Controller)**, and then select **Create**.



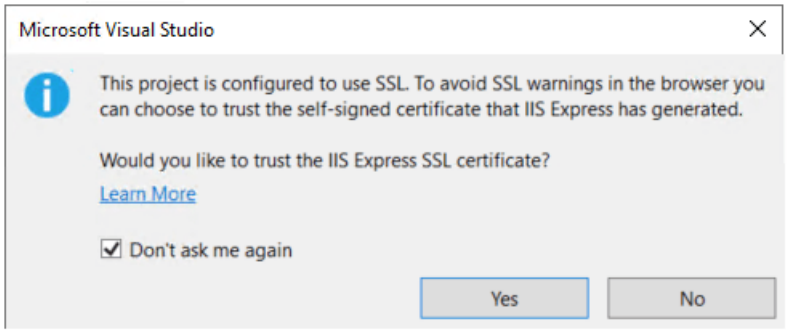
Visual Studio used the default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a basic starter project, and it's a good place to start.

Run the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

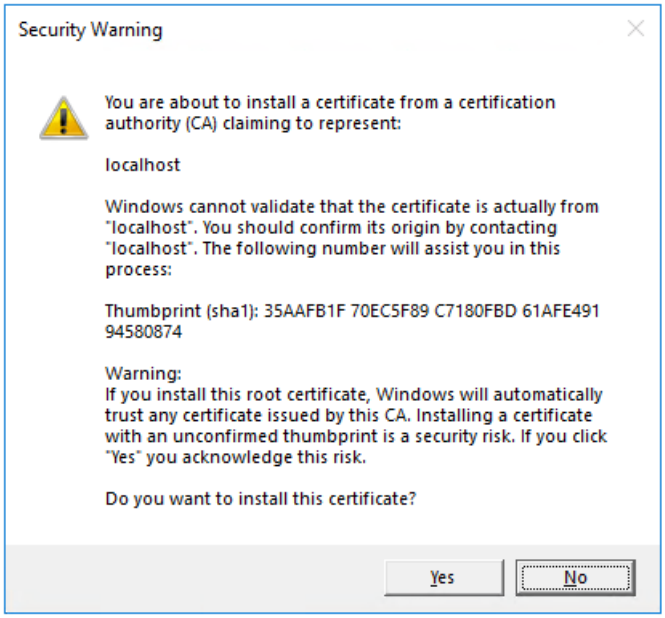
Select **Ctrl-F5** to run the app in non-debug mode.

Visual Studio displays the following dialog:



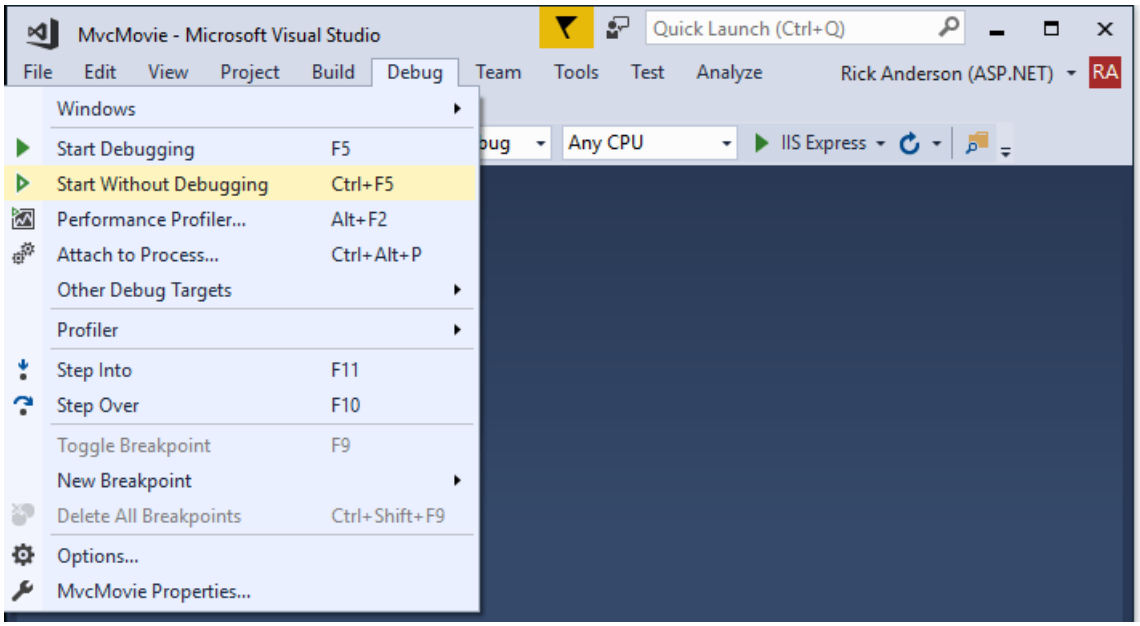
Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:

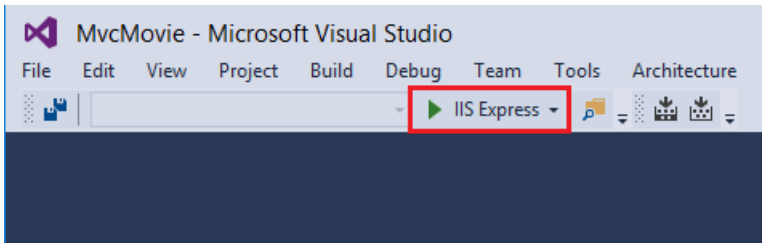


Select **Yes** if you agree to trust the development certificate.

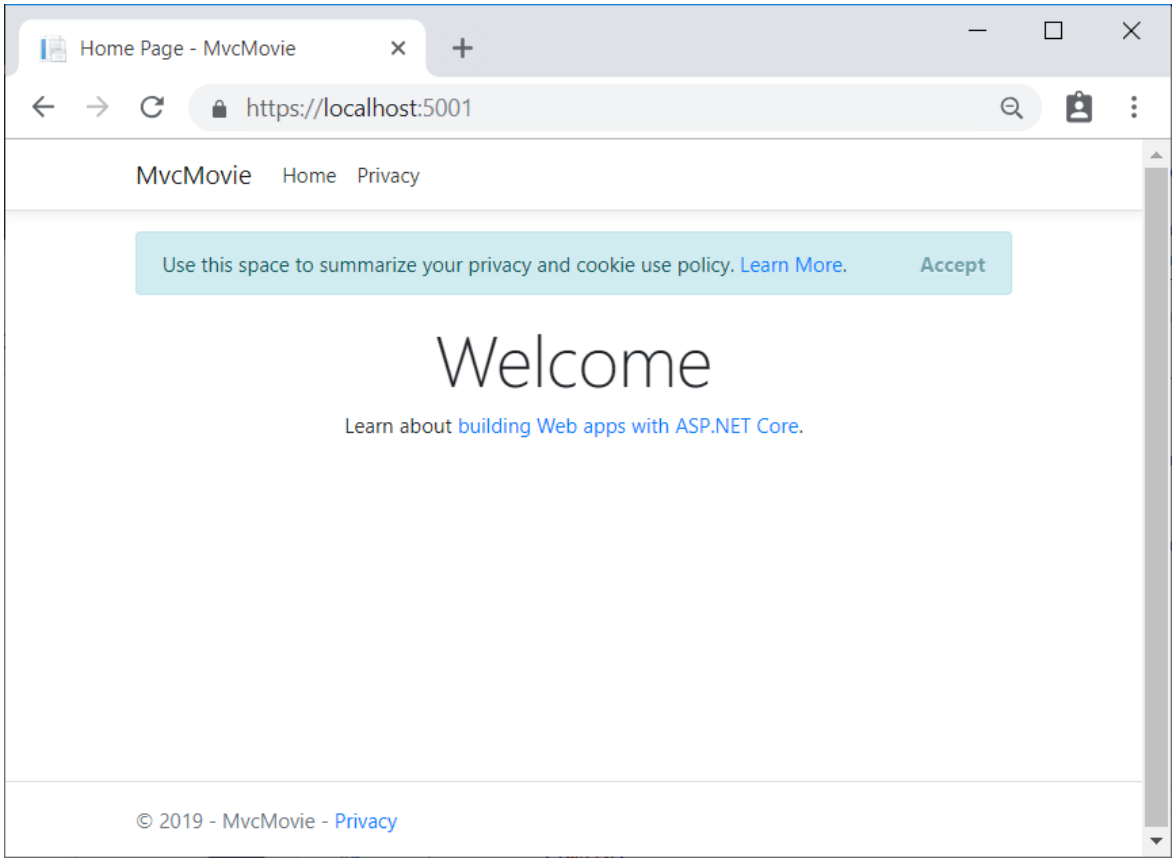
- Visual Studio starts IIS Express and runs the app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for your local computer. When Visual Studio creates a web project, a random port is used for the web server.
- Launching the app with Ctrl+F5 (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
- You can launch the app in debug or non-debug mode from the **Debug** menu item:



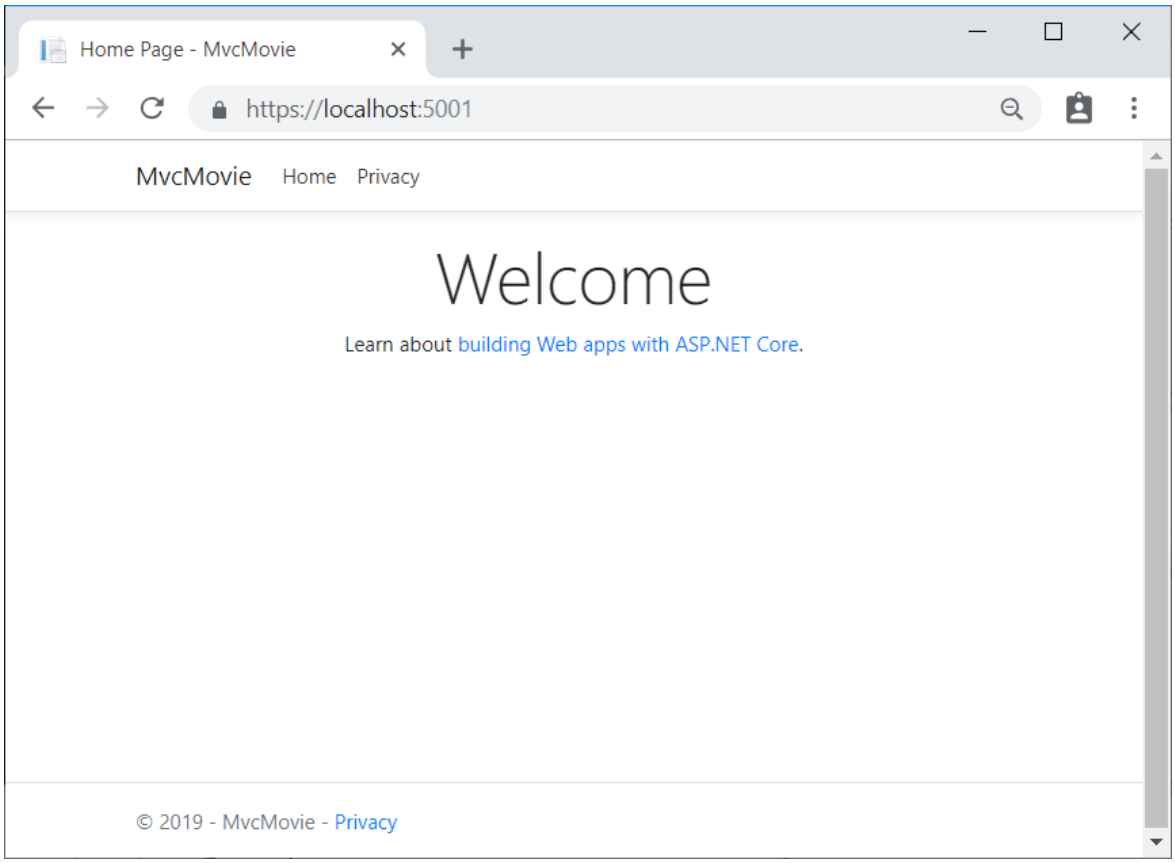
- You can debug the app by selecting the **IIS Express** button



- Select **Accept** to consent to tracking. This app doesn't track personal information. The template generated code includes assets to help meet General Data Protection Regulation (GDPR).



The following image shows the app after accepting tracking:



- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next part of this tutorial, you learn about MVC and start writing some code.

NEXT

Part 2, add a controller to an ASP.NET Core MVC app

9/22/2020 • 12 minutes to read • Edit Online

By Rick Anderson

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps. MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that handle browser requests. They retrieve model data and call view templates that return a response. In an MVC app, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database. For example, `https://localhost:5001/Home/Privacy` has route data of `Home` (the controller) and `Privacy` (the action method to call on the home controller). `https://localhost:5001/Movies/Edit/5` is a request to edit the movie with ID=5 using the movie controller. Route data is explained later in the tutorial.

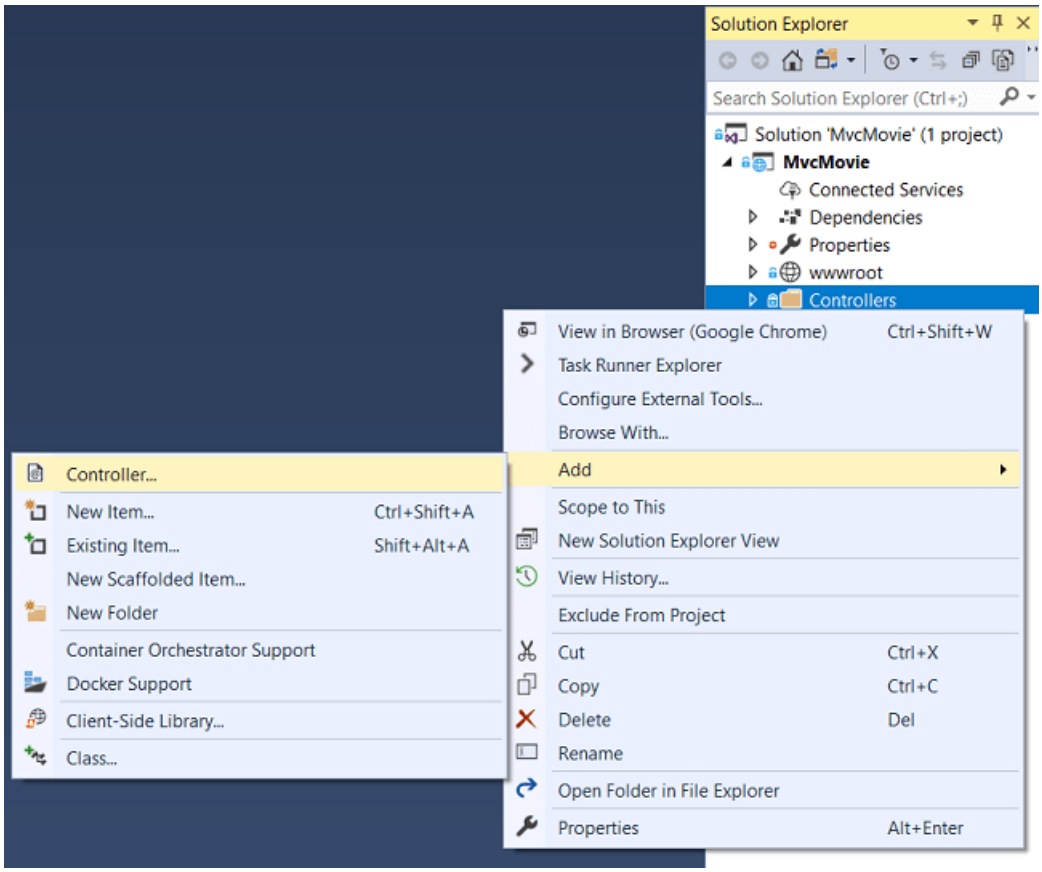
The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

We cover these concepts in this tutorial series and show you how to use them to build a movie app. The MVC project contains folders for the *Controllers* and *Views*.

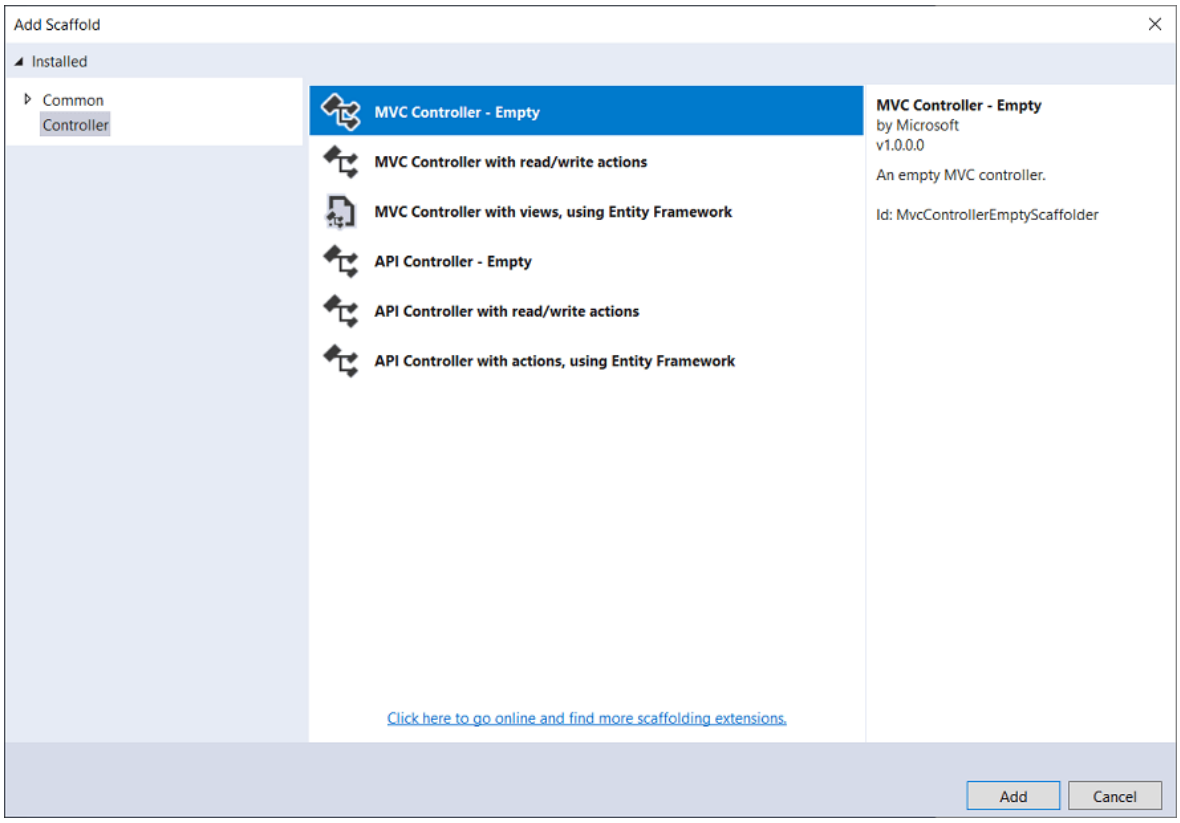
Add a controller

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click **Controllers** > **Add** > **Controller**



- In the **Add Scaffold** dialog box, select **Controller Class - Empty**



- In the **Add Empty MVC Controller dialog**, enter **HelloWorldController** and select **ADD**.

Replace the contents of *Controllers/HelloWorldController.cs* with the following:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

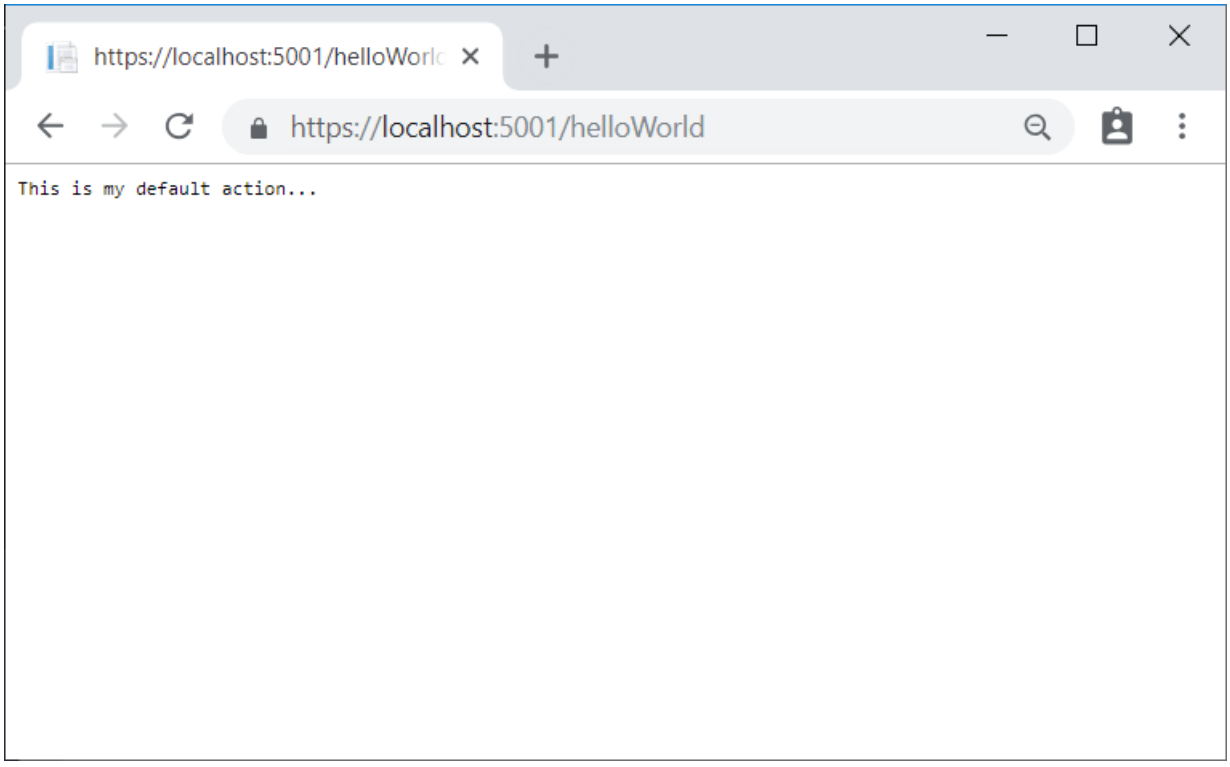
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`, and combines the protocol used: `HTTPS`, the network location of the web server (including the TCP port): `localhost:5001` and the target URI `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL. The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial the scaffolding engine is used to generate `HTTP POST` methods which update data.

Run the app in non-debug mode and append "HelloWorld" to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default URL routing logic used by MVC uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

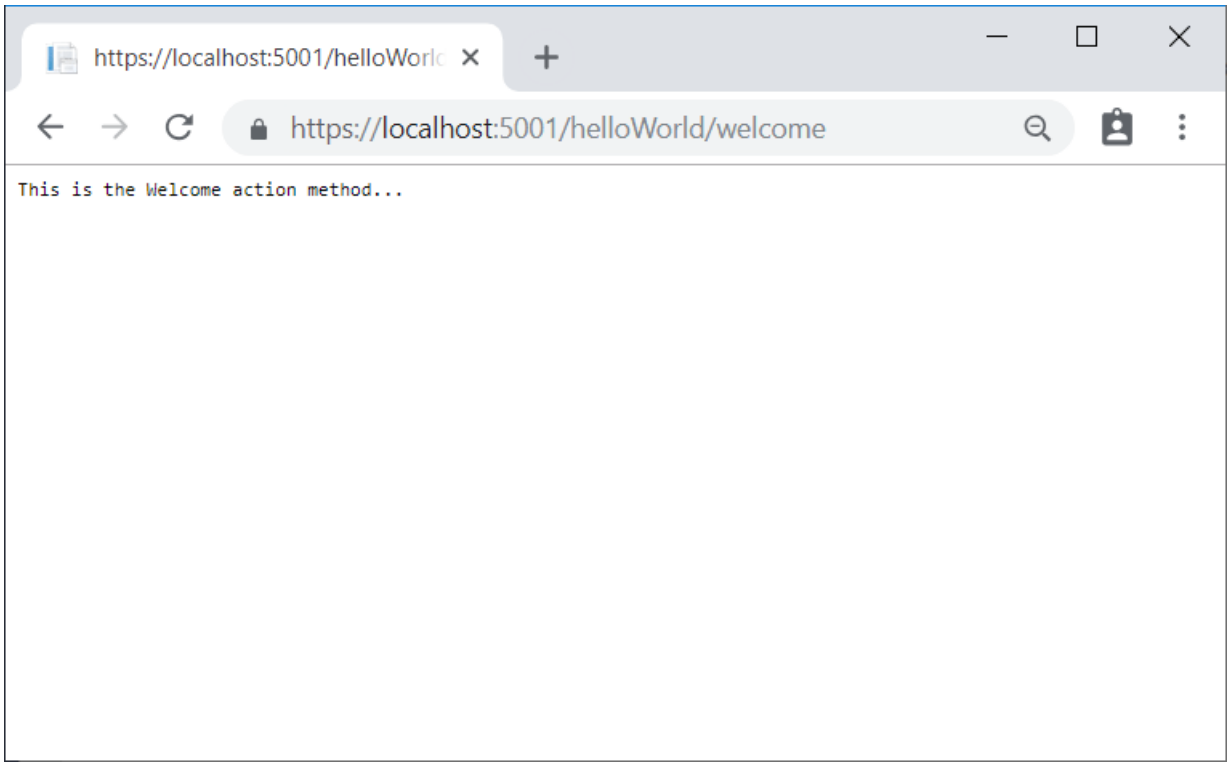
The routing format is set in the `Configure` method in *Startup.cs* file.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above.

The first URL segment determines the controller class to run. So `localhost:{PORT}/HelloWorld` maps to the **HelloWorld**Controller class. The second part of the URL segment determines the action method on the class. So `localhost:{PORT}/HelloWorld/Index` would cause the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:{PORT}/HelloWorld` and the `Index` method was called by default. That's because `Index` is the default method that will be called on a controller if a method name isn't explicitly specified. The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to `https://localhost:{PORT}/HelloWorld/Welcome`. The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`. Change the `Welcome` method to include two parameters as shown in the following code.

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

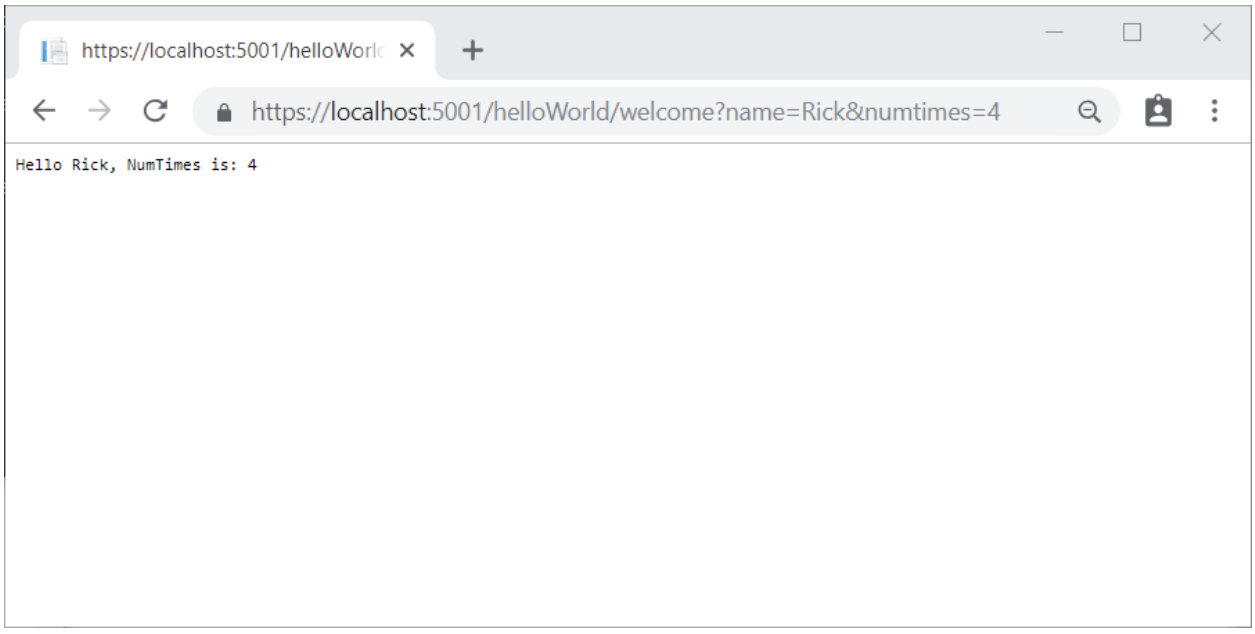
The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript).
- Uses Interpolated Strings in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

(Replace `{PORT}` with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method. See Model Binding for more information.



In the image above, the URL segment (`Parameters`) isn't used, the `name` and `numTimes` parameters are passed in the query string. The `?` (question mark) in the above URL is a separator, and the query string follows. The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick`

This time the third URL segment matched the route parameter `id`. The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method. The trailing `?` (in `id?`) indicates the `id` parameter is optional.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

In these examples the controller has been doing the "VC" portion of MVC - that is, the **V**iew and the **C**ontroller work. The controller is returning HTML directly. Generally you don't want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead you typically use a separate Razor view template file to generate the HTML response. You do that in the next tutorial.

PREVIOUS NEXT

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps. MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that handle browser requests. They retrieve model data and call view templates that return a response. In an MVC app, the view only displays information; the controller handles and responds to user input and interaction. For example, the controller handles route data and query-string values, and passes these values to the model. The model might use these values to query the database. For example, `https://localhost:5001/Home/About` has route data of `Home` (the controller) and `About` (the action method to call on the home controller). `https://localhost:5001/Movies/Edit/5` is a request to edit the movie with ID=5 using the movie controller. Route data is explained later in the tutorial.

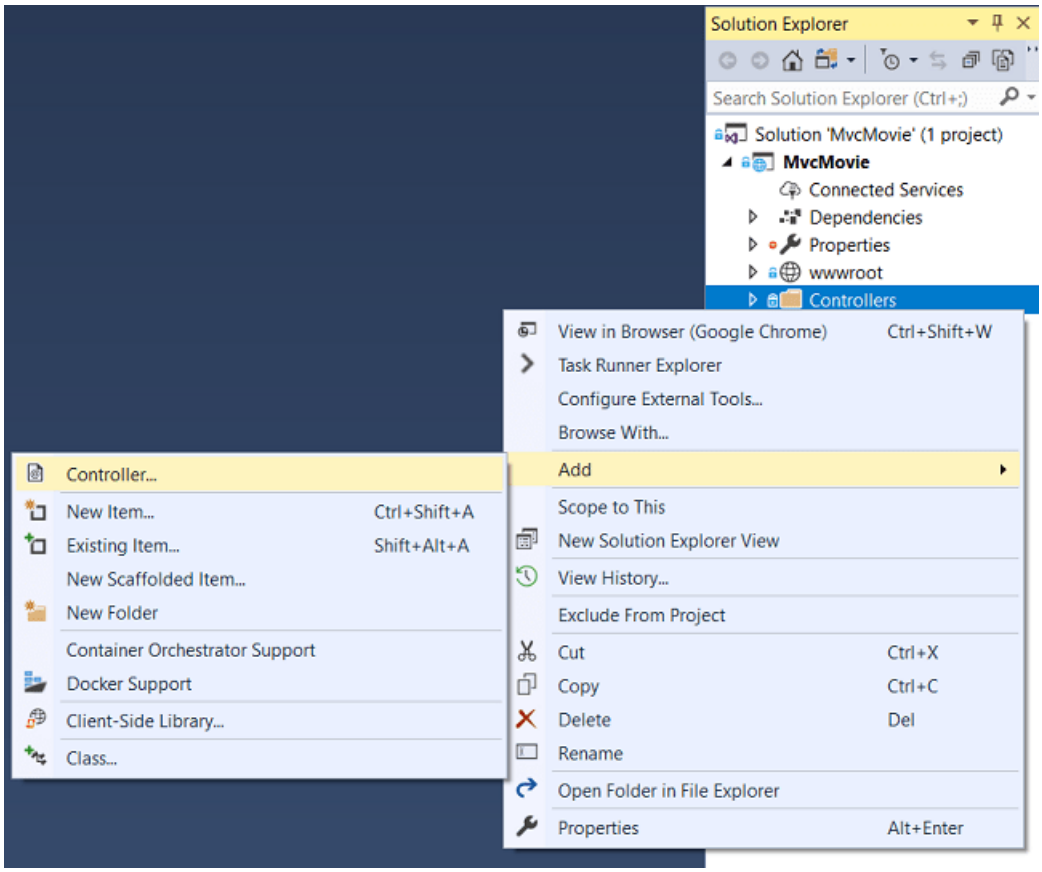
The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

We cover these concepts in this tutorial series and show you how to use them to build a movie app. The MVC project contains folders for the *Controllers* and *Views*.

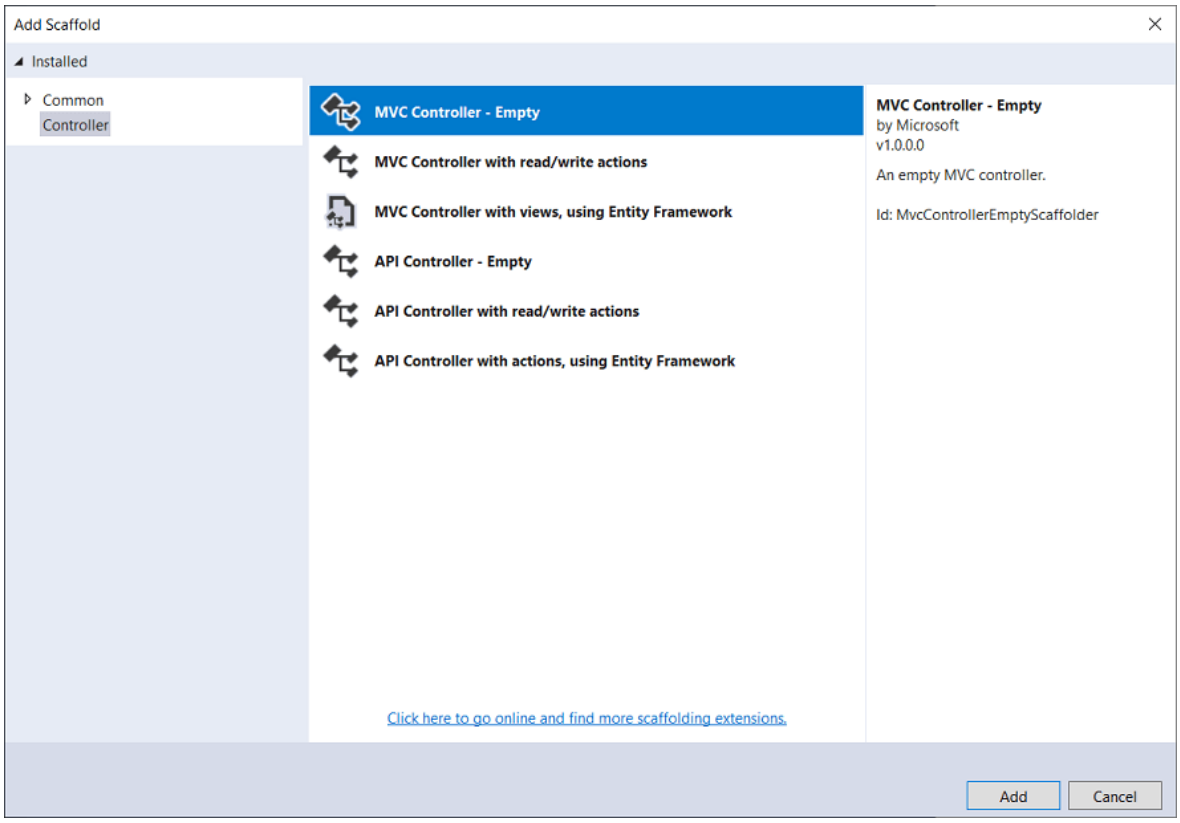
Add a controller

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click **Controllers** > **Add** > **Controller**



- In the **Add Scaffold** dialog box, select **MVC Controller - Empty**



- In the **Add Empty MVC Controller dialog**, enter **HelloWorldController** and select **ADD**.

Replace the contents of *Controllers/HelloWorldController.cs* with the following:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my default action...";
        }

        //
        // GET: /HelloWorld/Welcome/

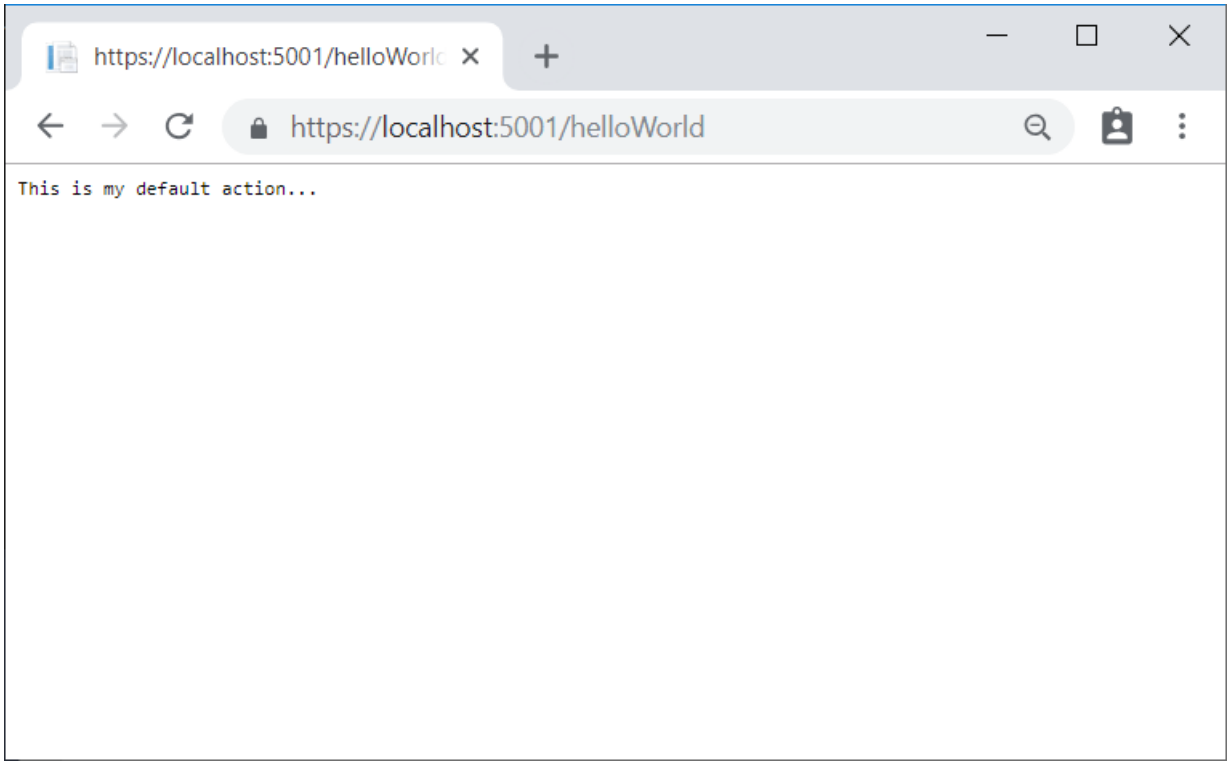
        public string Welcome()
        {
            return "This is the Welcome action method...";
        }
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`, and combines the protocol used: `HTTPS`, the network location of the web server (including the TCP port): `localhost:5001` and the target URI `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL. The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial the scaffolding engine is used to generate `HTTP POST` methods which update data.

Run the app in non-debug mode and append "HelloWorld" to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default URL routing logic used by MVC uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

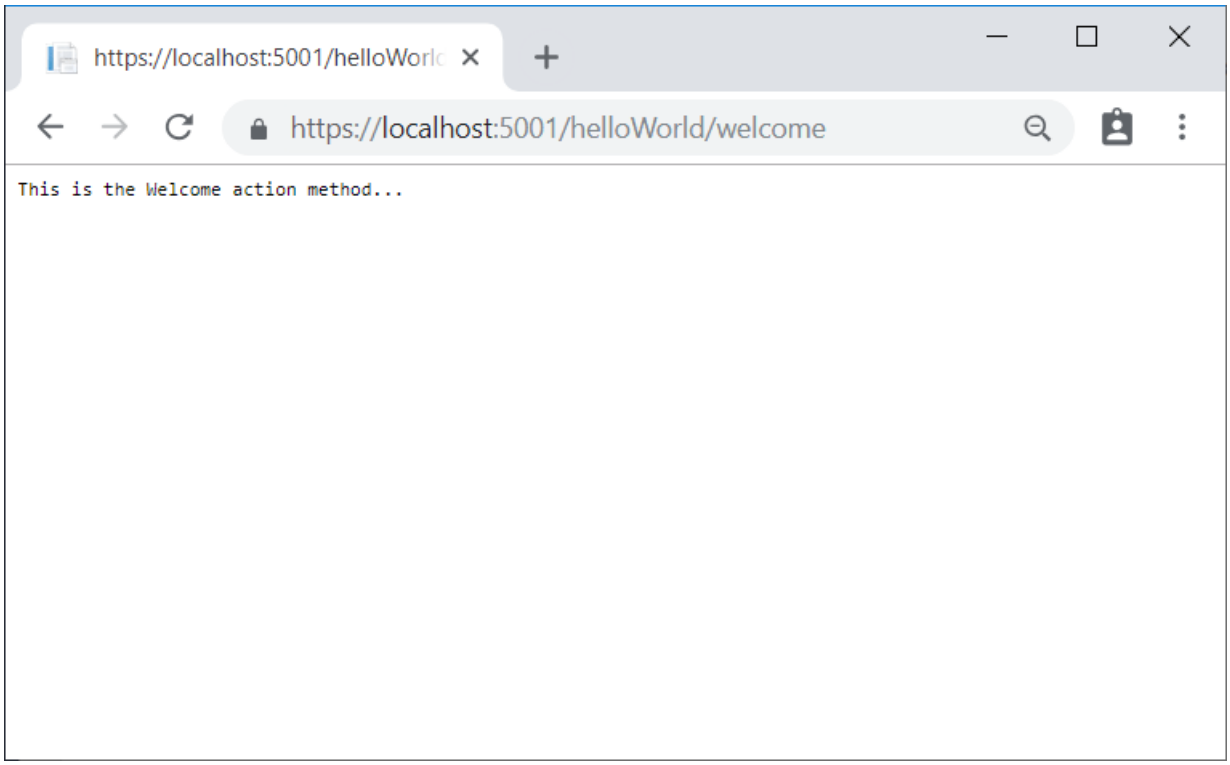
The routing format is set in the `Configure` method in *Startup.cs* file.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above.

The first URL segment determines the controller class to run. So `localhost:{PORT}/HelloWorld` maps to the `HelloWorldController` class. The second part of the URL segment determines the action method on the class. So `localhost:{PORT}/HelloWorld/Index` would cause the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:{PORT}/HelloWorld` and the `Index` method was called by default. This is because `Index` is the default method that will be called on a controller if a method name isn't explicitly specified. The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to `https://localhost:{PORT}/HelloWorld/Welcome`. The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`. Change the `Welcome` method to include two parameters as shown in the following code.

```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

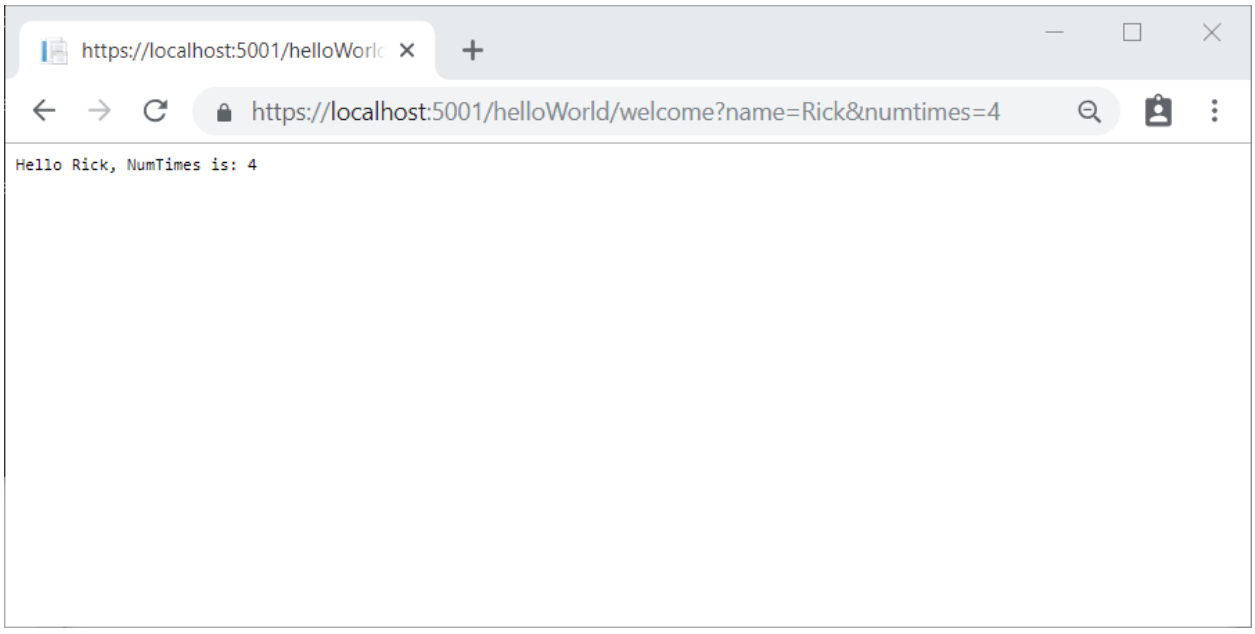
The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript).
- Uses Interpolated Strings in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

(Replace `{PORT}` with your port number.) You can try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string in the address bar to parameters in your method. See Model Binding for more information.



In the image above, the URL segment (`Parameters`) isn't used, the `name` and `numTimes` parameters are passed in the query string. The `?` (question mark) in the above URL is a separator, and the query string follows. The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick`

This time the third URL segment matched the route parameter `id`. The `Welcome` method contains a parameter `id` that matched the URL template in the `MapRoute` method. The trailing `?` (in `id?`) indicates the `id` parameter is optional.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

In these examples the controller has been doing the "VC" portion of MVC - that is, the view and controller work. The controller is returning HTML directly. Generally you don't want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead you typically use a separate Razor view template file to help generate the HTML response. You do that in the next tutorial.

PREVIOUS NEXT

Part 3, add a view to an ASP.NET Core MVC app

9/22/2020 • 16 minutes to read • Edit Online

By Rick Anderson

In this section you modify the `HelloWorldController` class to use Razor view files to cleanly encapsulate the process of generating HTML responses to a client.

You create a view template file using Razor. Razor-based view templates have a *.cshtml* file extension. They provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message that's hard-coded in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

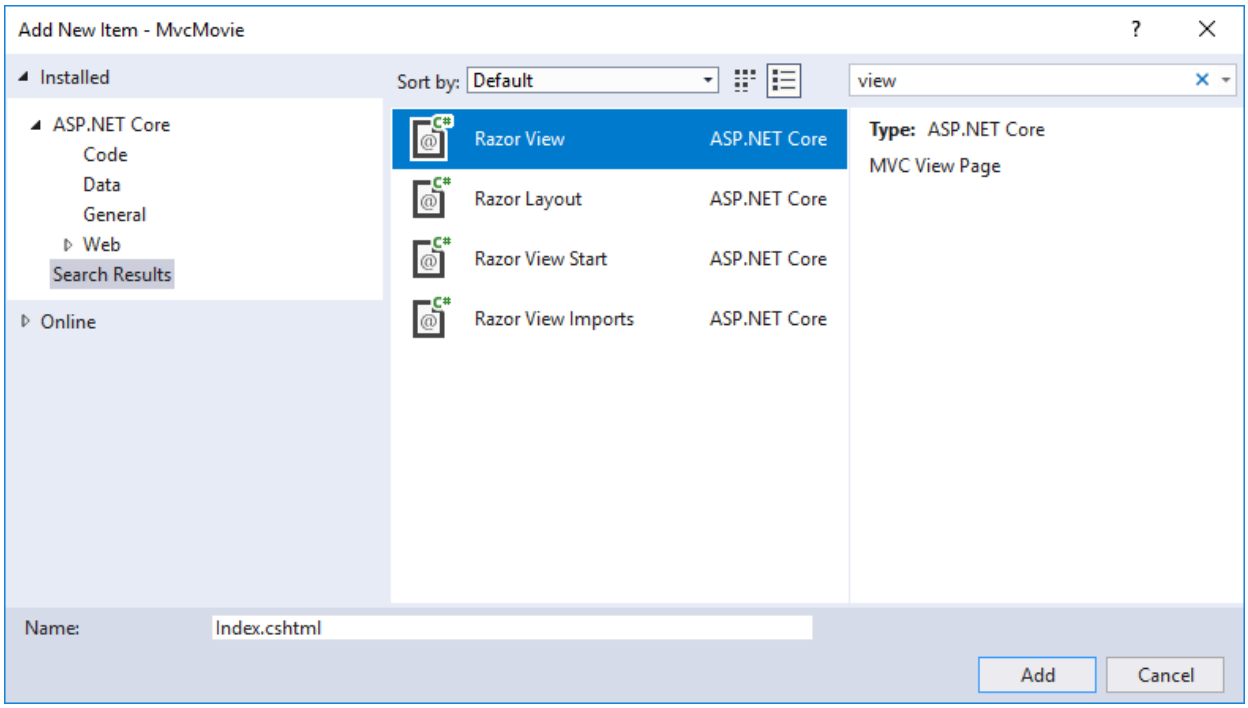
```
public IActionResult Index()
{
    return View();
}
```

The preceding code calls the controller's View method. It uses a view template to generate an HTML response. Controller methods (also known as *action methods*), such as the `Index` method above, generally return an IActionResult (or a class derived from ActionResult), not a type like `string`.

Add a view

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Right click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.
- Right click on the *Views/HelloWorld* folder, and then **Add > New Item**.
- In the **Add New Item - MvcMovie** dialog
  - In the search box in the upper-right, enter *view*
  - Select **Razor View**
  - Keep the **Name** box value, *Index.cshtml*.
  - Select **Add**



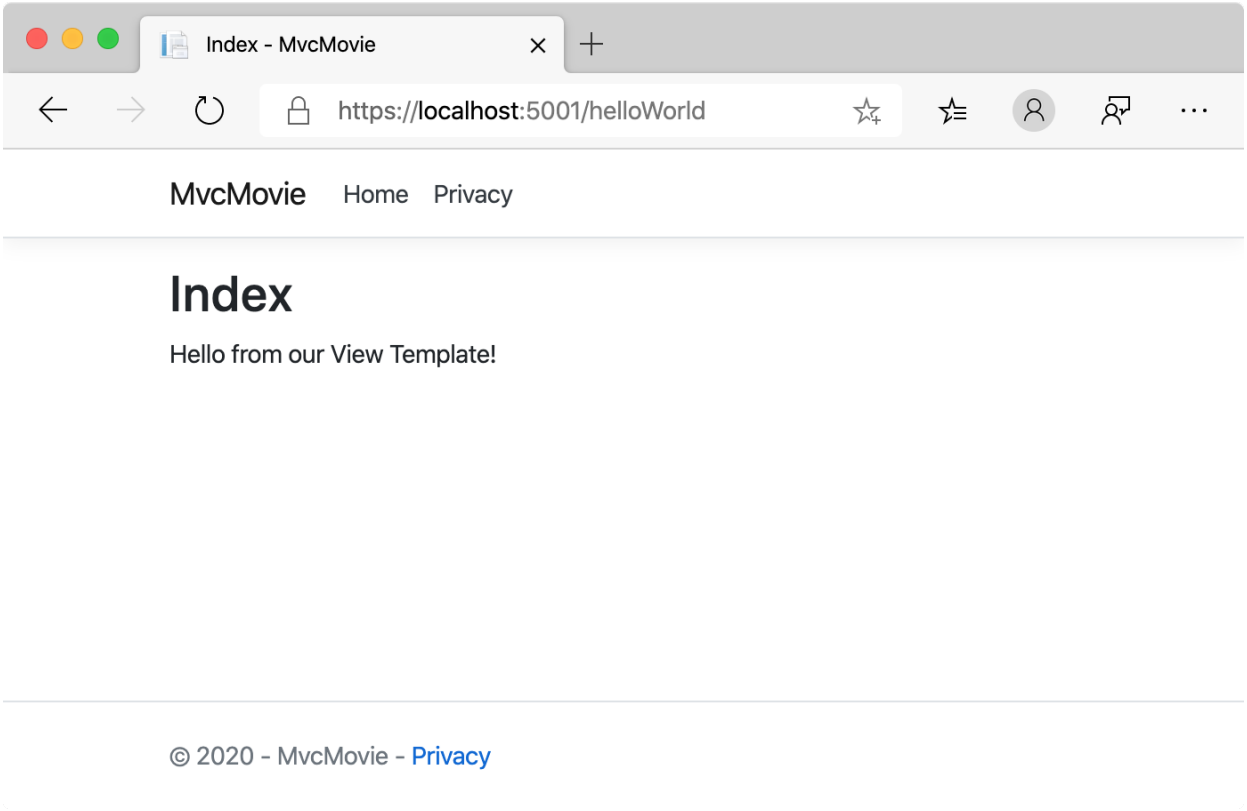
Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://localhost:{PORT}/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much; it ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because a view template file name wasn't specified, MVC defaulted to using the default view file. The default view file has the same name as the method (`Index`), so the view template in */Views/HelloWorld/Index.cshtml* is used. The image below shows the string "Hello from our View Template!" hard-coded in the view.



Change views and layout pages

Select the menu links (**MvcMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Views/Shared/_Layout.cshtml* file. Open the *Views/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the **Views/Home/Privacy.cshtml** view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

Replace the content of the *Views/Shared/_Layout.cshtml* file with the following markup. The changes are highlighted:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2020 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @RenderSection("Scripts", required: false)
</body>
</html>
```

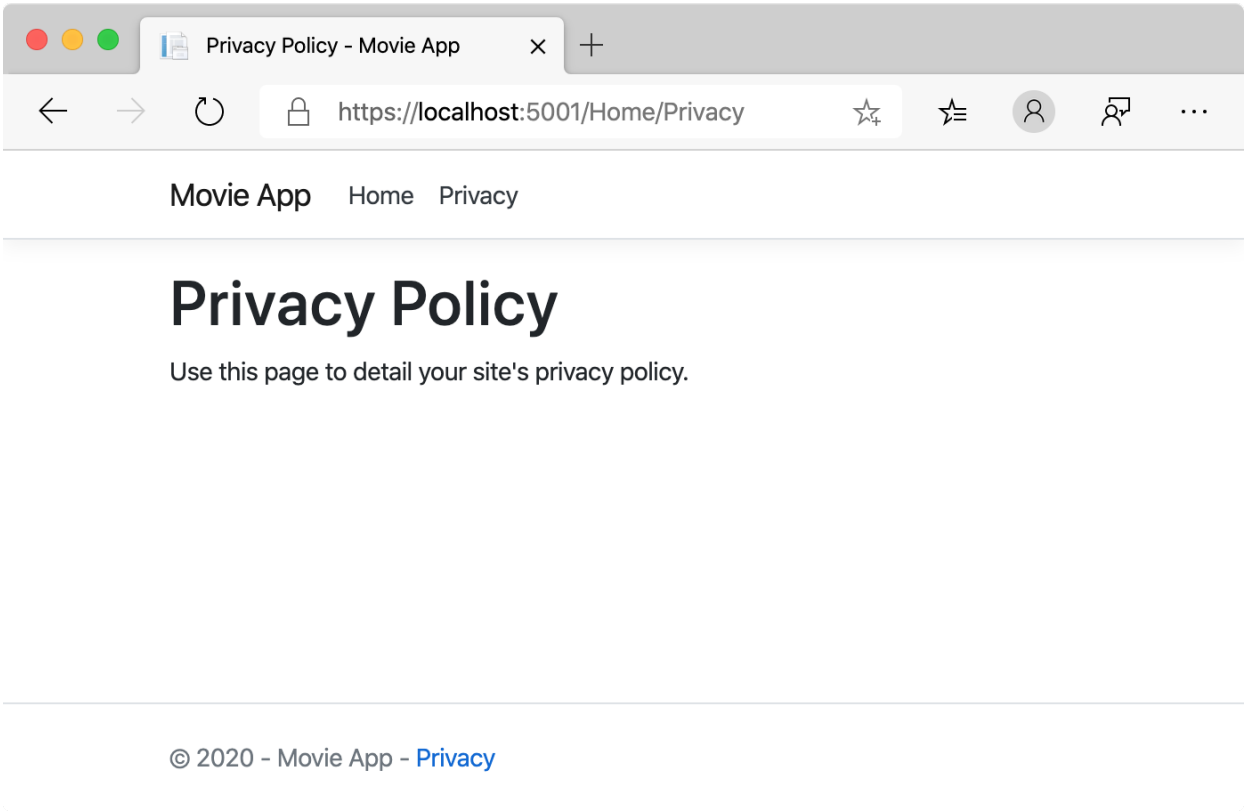
The preceding markup made the following changes:

- 3 occurrences of `MvcMovie` to `Movie App`.
- The anchor element
  `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to
  `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

In the preceding markup, the `asp-area=""` anchor Tag Helper attribute and attribute value was omitted because this app is not using Areas.

Note: The `Movies` controller has not been implemented. At this point, the `Movie App` link is not functional.

Save your changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - Mvc Movie**:



Select the **Home** link and notice that the title and anchor text also display **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the *Views/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Change the title and `<h2>` element of the *Views/HelloWorld/Index.cshtml* view file:

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

The title and `<h2>` element are slightly different so you can see which bit of code changes the display.

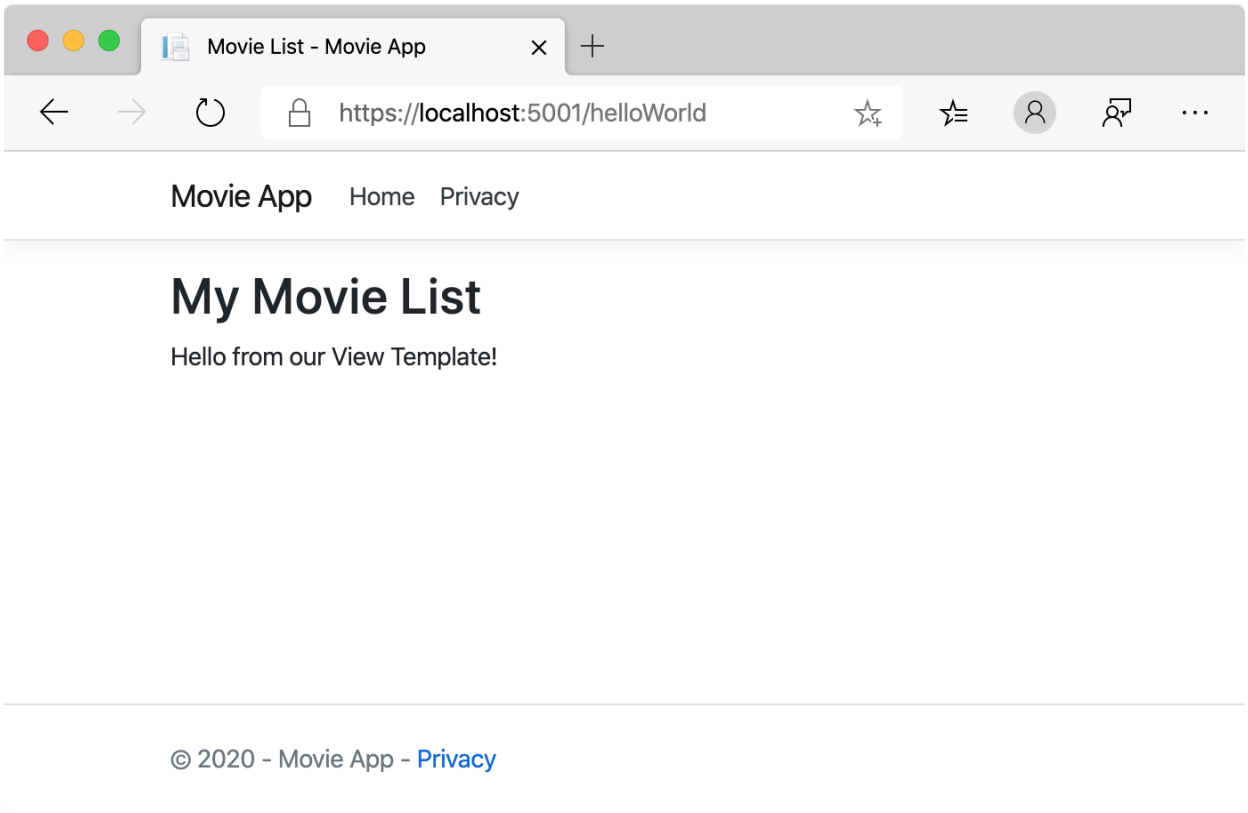
`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

Save the change and navigate to `https://localhost:{PORT}/HelloWorld`. Notice that the browser title, the primary

heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with `ViewData["Title"]` we set in the *Index.cshtml* view template and the additional "- Movie App" added in the layout file.

The content in the *Index.cshtml* view template is merged with the *Views/Shared/_Layout.cshtml* view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app. To learn more, see Layout.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code clean, testable, and maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and a `ID` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate bits of data must be passed from the controller to the view in order to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary that the view template can then access.

In *HelloWorldController.cs*, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means any type can be used; the `ViewData`

object has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete *HelloWorldController.cs* file looks like this:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named *Views/HelloWorld/Welcome.cshtml*.

You'll create a loop in the *Welcome.cshtml* view template that displays "Hello" `NumTimes`. Replace the contents of *Views/HelloWorld/Welcome.cshtml* with the following:

```
@{
    ViewData["Title"] = "Welcome";
}

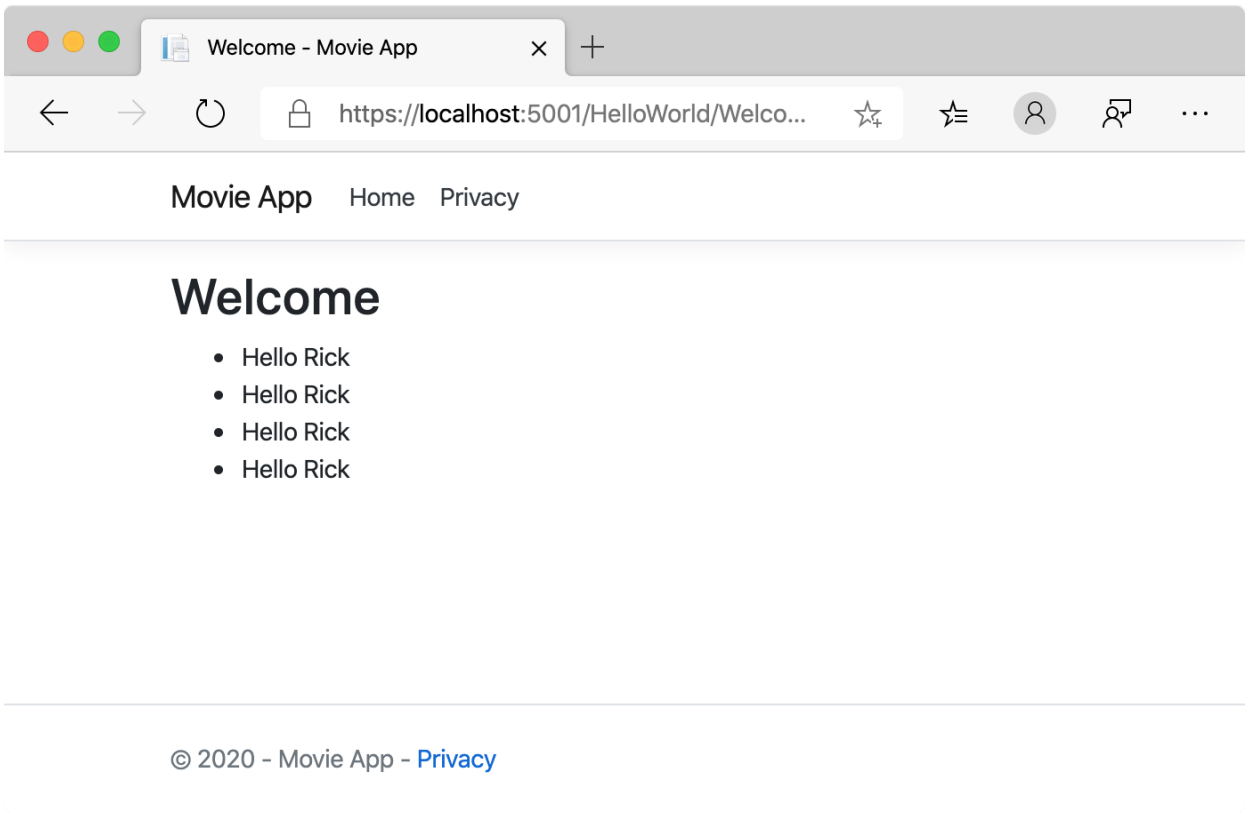
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder . The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the sample above, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See When to use ViewBag, ViewData, or TempData for more information.

In the next tutorial, a database of movies is created.

PREVIOUS NEXT

In this section you modify the `HelloWorldController` class to use Razor view files to cleanly encapsulate the process of generating HTML responses to a client.

You create a view template file using Razor. Razor-based view templates have a *.cshtml* file extension. They provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message that's hard-coded in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

```
public IActionResult Index()
{
    return View();
}
```

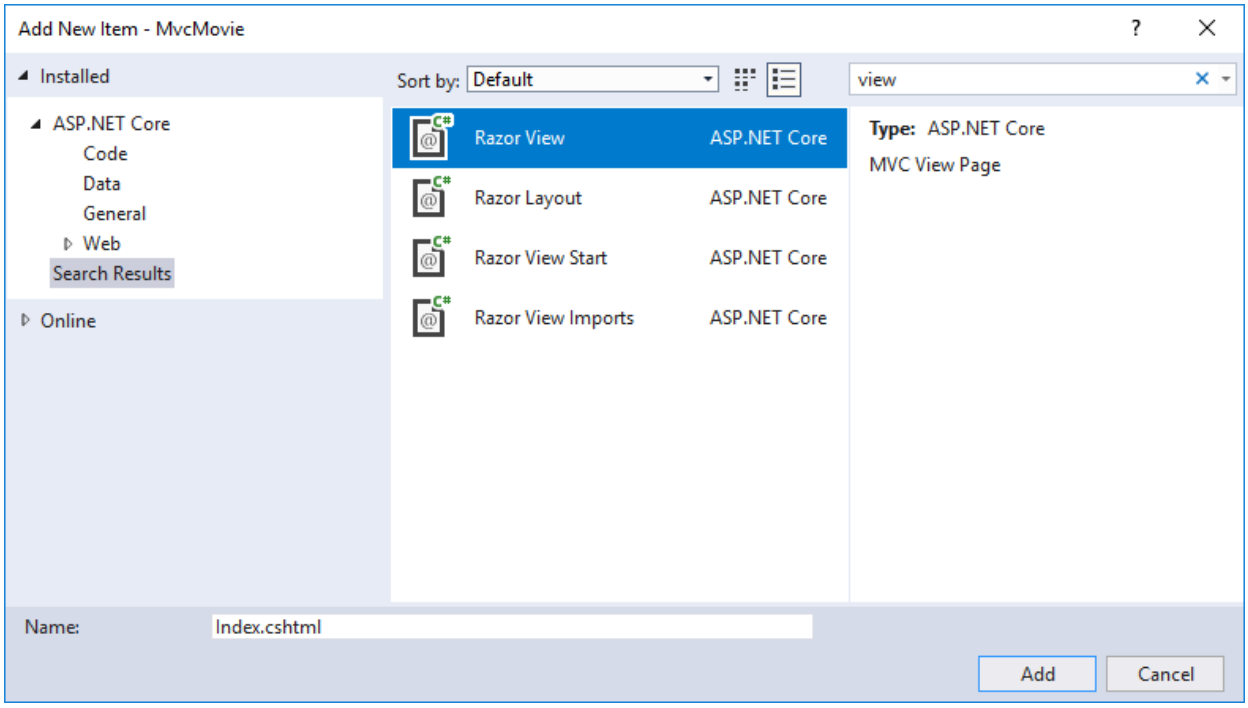
The preceding code calls the controller's View method. It uses a view template to generate an HTML response. Controller methods (also known as *action methods*), such as the `Index` method above, generally return an IActionResult (or a class derived from ActionResult), not a type like `string`.

Add a view

- Visual Studio
- Visual Studio Code

- Visual Studio for Mac

- Right click on the *Views* folder, and then **Add** > **New Folder** and name the folder *HelloWorld*.
- Right click on the *Views/HelloWorld* folder, and then **Add** > **New Item**.
- In the **Add New Item - MvcMovie** dialog
  - In the search box in the upper-right, enter *view*
  - Select **Razor View**
  - Keep the **Name** box value, *Index.cshtml*.
  - Select **Add**



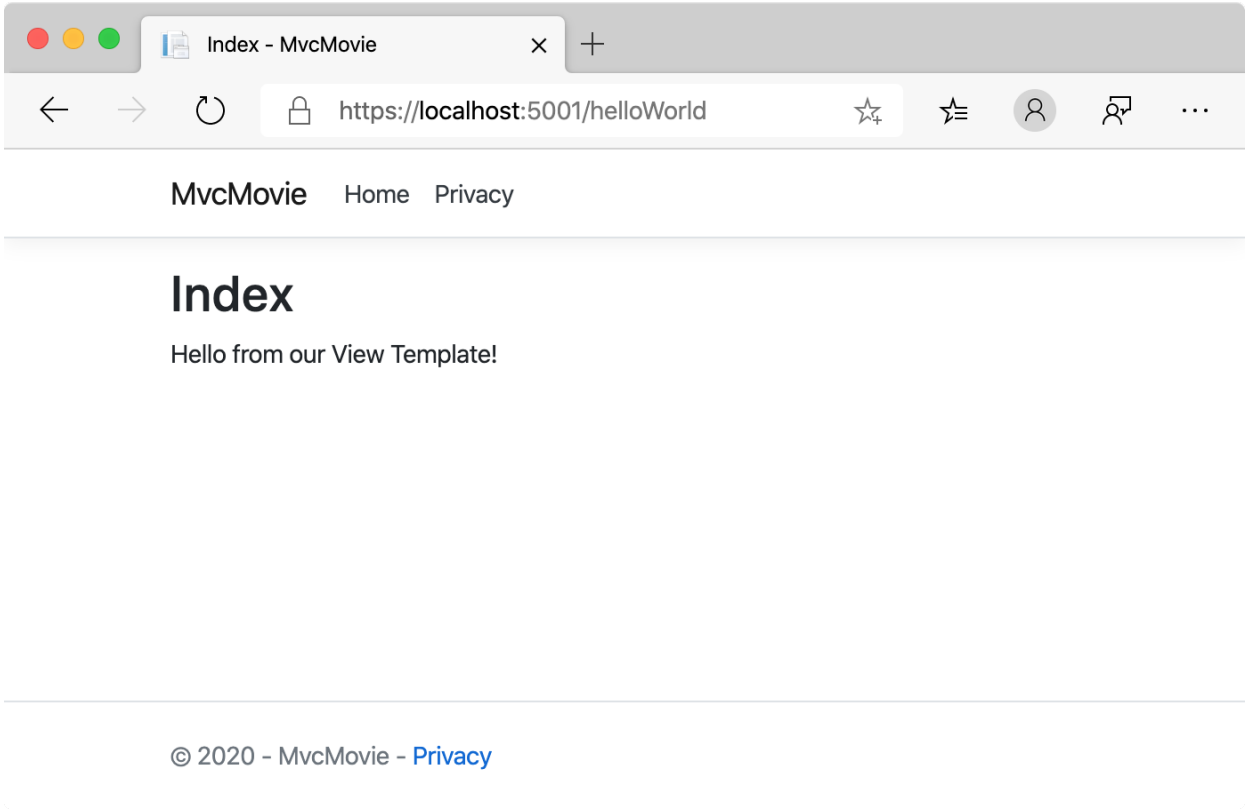
Replace the contents of the *Views/HelloWorld/Index.cshtml* Razor view file with the following:

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://localhost:{PORT}/HelloWorld`. The `Index` method in the `HelloWorldController` didn't do much; it ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser. Because a view template file name wasn't specified, MVC defaulted to using the default view file. The default view file has the same name as the method (`Index`), so in the */Views/HelloWorld/Index.cshtml* is used. The image below shows the string "Hello from our View Template!" hard-coded in the view.



Change views and layout pages

Select the menu links (**MvcMovie**, **Home**, and **Privacy**). Each page shows the same menu layout. The menu layout is implemented in the *Views/Shared/_Layout.cshtml* file. Open the *Views/Shared/_Layout.cshtml* file.

Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the **Views/Home/Privacy.cshtml** view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

- In the title and footer elements, change `MvcMovie` to `Movie App`.
- Change the anchor element
  `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to
  `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

The following markup shows the highlighted changes:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute"
```

```
            crossorigin="anonymous"
            integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE="/>
    </environment>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>

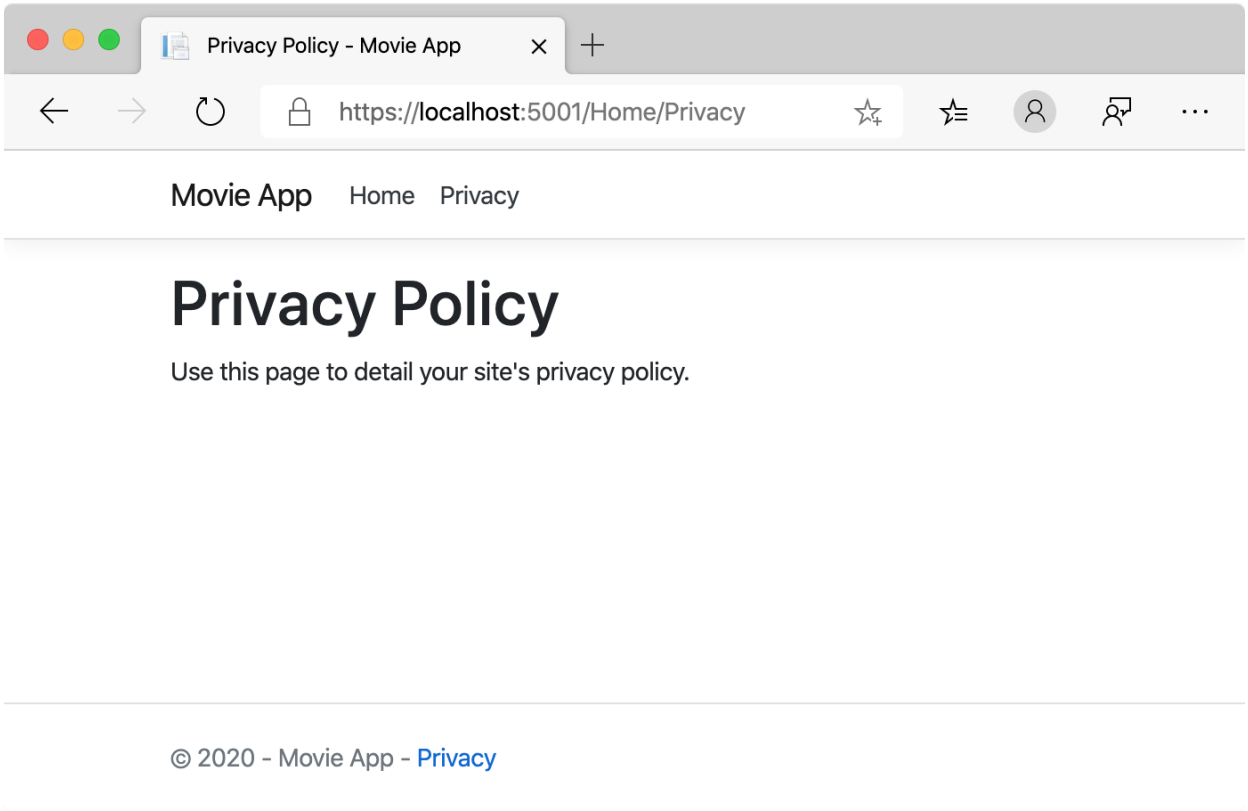
    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">
        </script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/js/bootstrap.bundle.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha256-E/V4cWE4qvAeO5MOhjtGtqDzPndRO1LBk8lJ/PR7CA4=">
        </script>
    </environment>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

In the preceding markup, the `asp-area` anchor Tag Helper attribute was omitted because this app is not using Areas.

Note: The `Movies` controller has not been implemented. At this point, the `Movie App` link is not functional.

Save your changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - Mvc Movie**:



Select the **Home** link and notice that the title and anchor text also display **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the *Views/_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The *Views/_ViewStart.cshtml* file brings in the *Views/Shared/_Layout.cshtml* file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Change the title and `<h2>` element of the *Views/HelloWorld/Index.cshtml* view file:

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

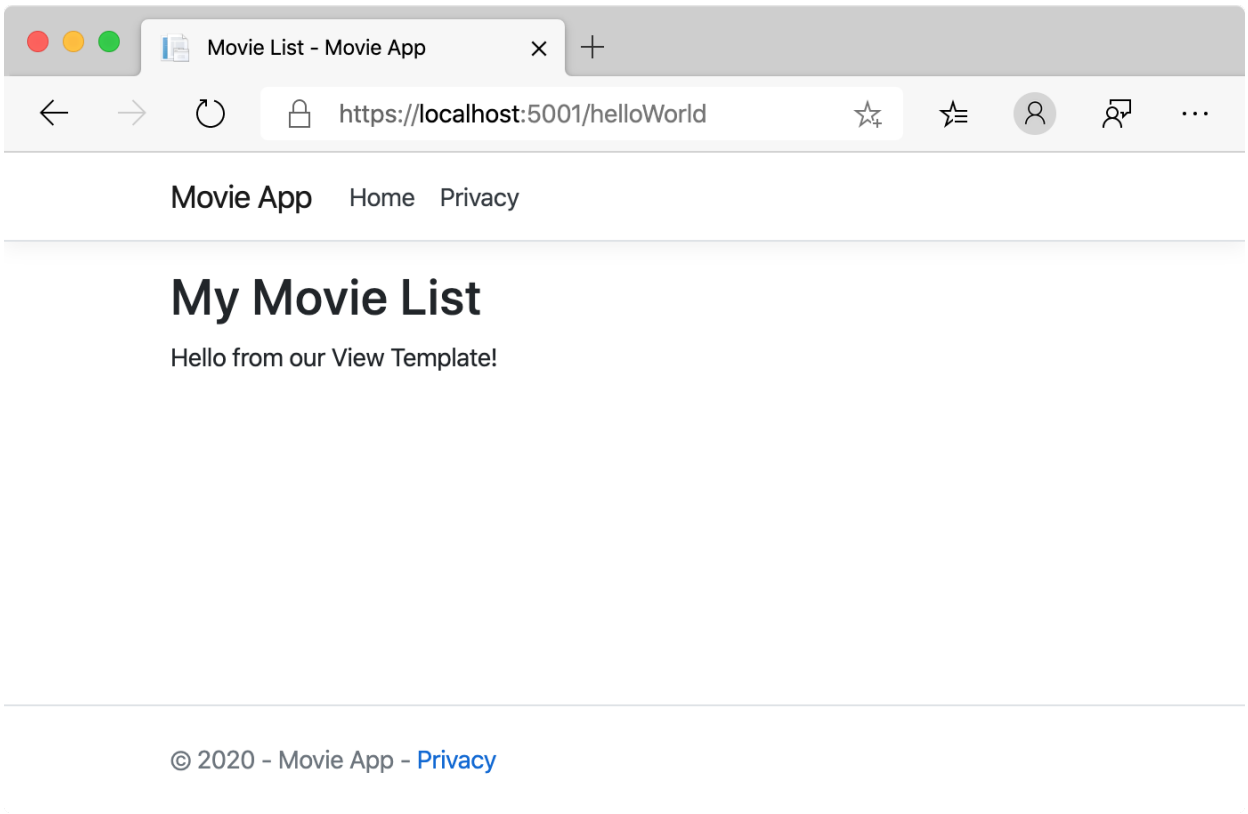
The title and `<h2>` element are slightly different so you can see which bit of code changes the display.

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

```
<title>@ViewData["Title"] - Movie App</title>
```

Save the change and navigate to `https://localhost:{PORT}/HelloWorld`. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with `ViewData["Title"]` we set in the *Index.cshtml* view template and the additional "- Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *Views/Shared/_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application. To learn more see Layout.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code clean, testable, and maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and a `ID` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate bits of data must be passed from the controller to the view in order to generate the

response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary that the view template can then access.

In *HelloWorldController.cs*, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means any type can be used; the `ViewData` object has no defined properties until you put something inside it. The MVC model binding system automatically maps the named parameters (`name` and `numTimes`) from the query string in the address bar to parameters in your method. The complete *HelloWorldController.cs* file looks like this:

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Welcome(string name, int numTimes = 1)
        {
            ViewData["Message"] = "Hello " + name;
            ViewData["NumTimes"] = numTimes;

            return View();
        }
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named *Views/HelloWorld/Welcome.cshtml*.

You'll create a loop in the *Welcome.cshtml* view template that displays "Hello" `NumTimes`. Replace the contents of *Views/HelloWorld/Welcome.cshtml* with the following:

```
@{
    ViewData["Title"] = "Welcome";
}

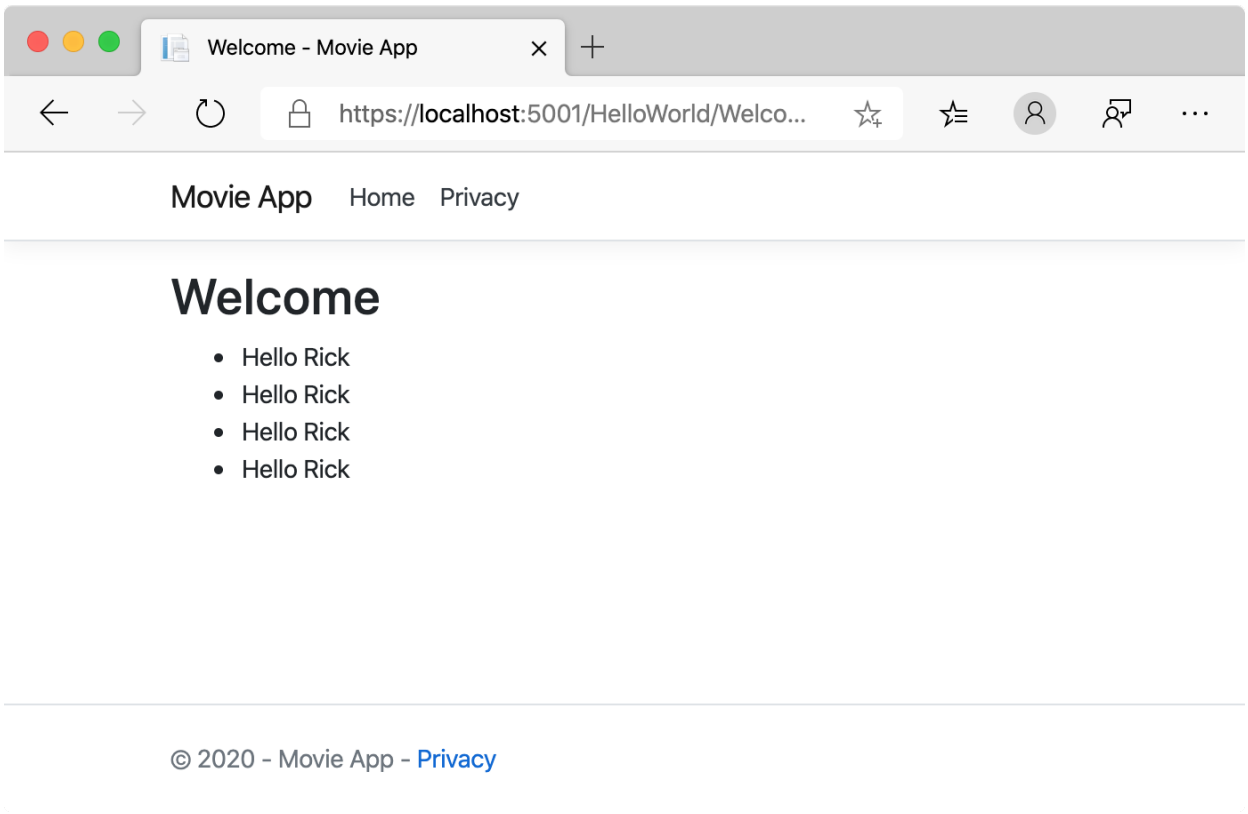
<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder . The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the sample above, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the `ViewData` dictionary approach. See When to use ViewBag, ViewData, or TempData for more information.

In the next tutorial, a database of movies is created.

PREVIOUS NEXT

Part 4, add a model to an ASP.NET Core MVC app

9/22/2020 • 28 minutes to read • Edit Online

By Rick Anderson and Tom Dykstra

In this section, you add classes for managing movies in a database. These classes will be the "**M**odel" part of the **M**VC app.

You use these classes with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes you create are known as POCO classes (from **P**lain **O**ld **C**LR **O**bjects) because they don't have any dependency on EF Core. They just define the properties of the data that will be stored in the database.

In this tutorial, you write the model classes first, and EF Core creates the database.

Add a data model class

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Right-click the *Models* folder > **Add** > **Class**. Name the file *Movie.cs*.

Update the *Movie.cs* file with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains an `Id` field, which is required by the database for the primary key.

The DataType attribute on `ReleaseDate` specifies the type of the data (`Date`). With this attribute:

- The user is not required to enter time information in the date field.
- Only the date is displayed, not time information.

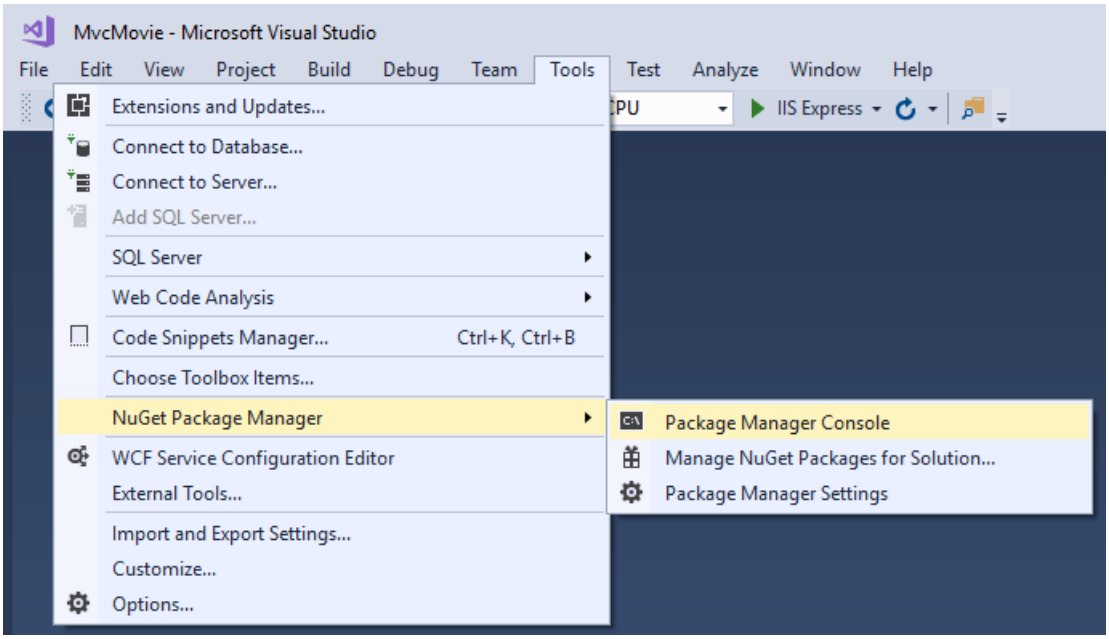
DataAnnotations are covered in a later tutorial.

Add NuGet packages

- Visual Studio

- Visual Studio Code
- Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).



In the PMC, run the following command:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

The preceding command adds the EF Core SQL Server provider. The provider package installs the EF Core package as a dependency. Additional packages are installed automatically in the scaffolding step later in the tutorial.

Create a database context class

A database context class is needed to coordinate EF Core functionality (Create, Read, Update, Delete) for the `Movie` model. The database context is derived from Microsoft.EntityFrameworkCore.DbContext and specifies the entities to include in the data model.

Create a *Data* folder.

Add a *Data/MvcMovieContext.cs* file with the following code:

```
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an

entity set typically corresponds to a database table. An entity corresponds to a row in the table.

Register the database context

ASP.NET Core is built with dependency injection (DI). Services (such as the EF Core DB context) must be registered with DI during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial. In this section, you register the database context with the DI container.

Add the following `using` statements at the top of *Startup.cs*:

```
using MvcMovie.Data;
using Microsoft.EntityFrameworkCore;
```

Add the following highlighted code in `Startup.ConfigureServices`:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddDbContext<MvcMovieContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Add a database connection string

Add a connection string to the *appsettings.json* file:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

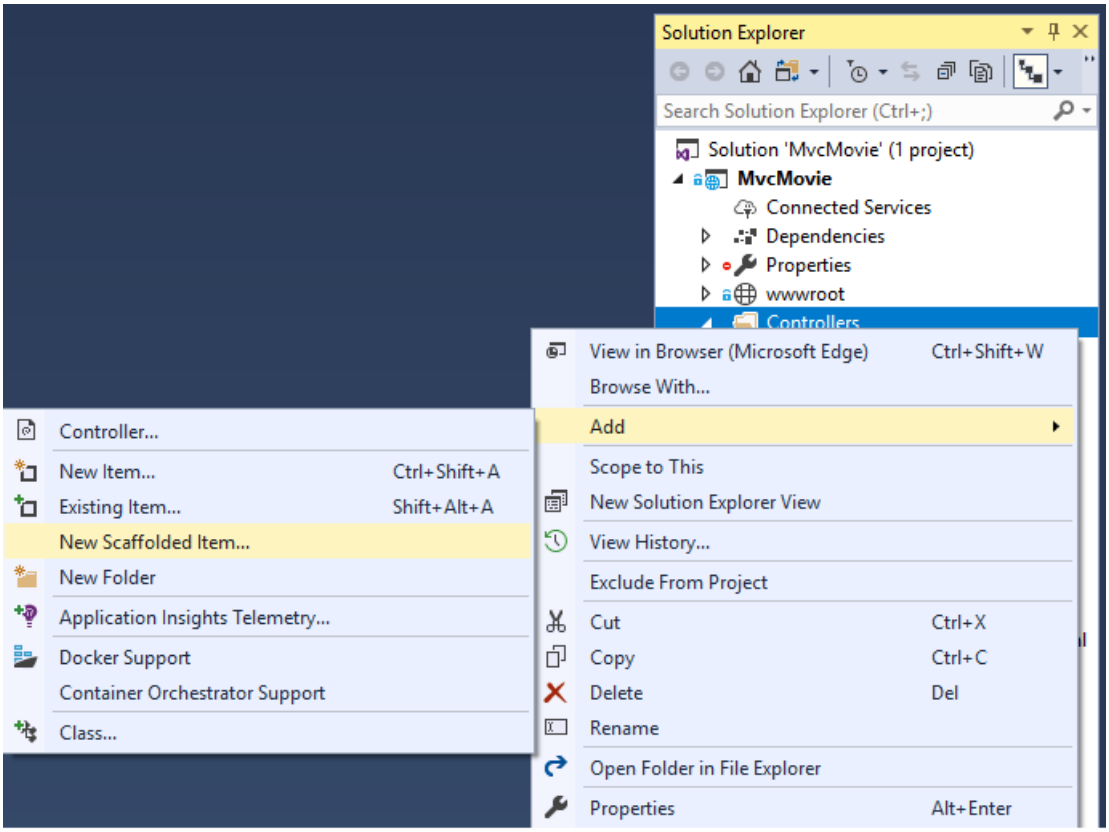
Build the project as a check for compiler errors.

Scaffold movie pages

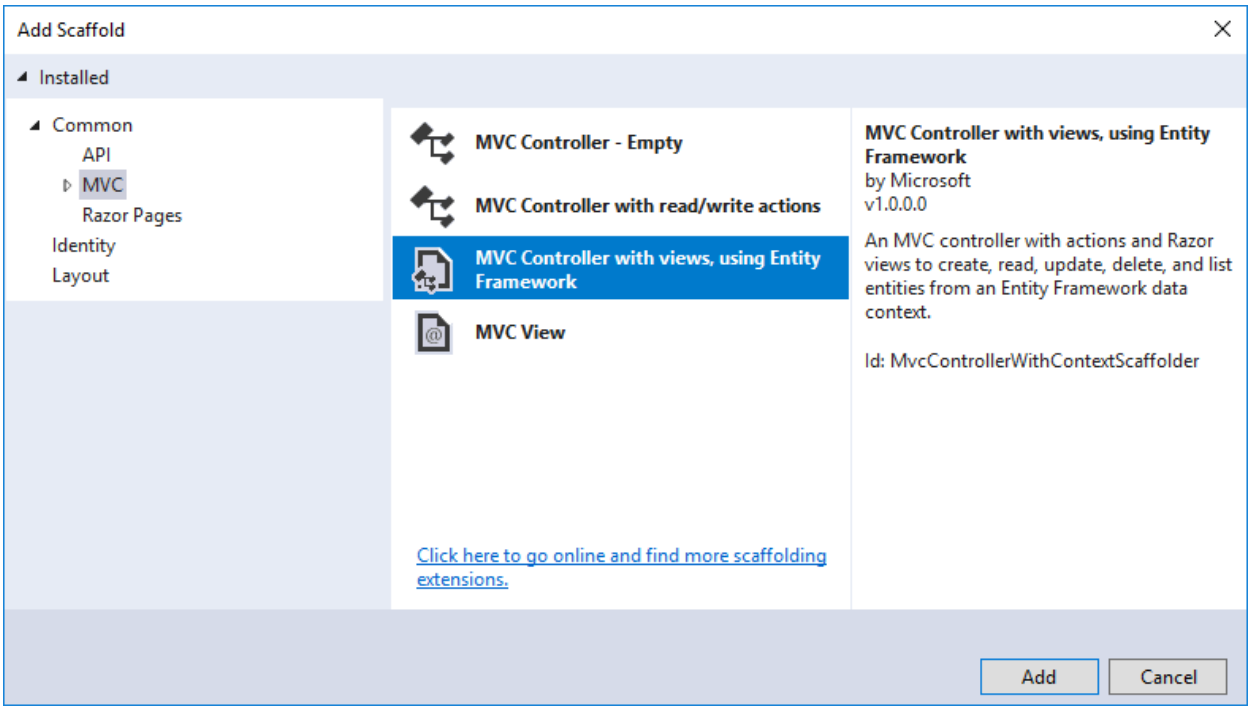
Use the scaffolding tool to produce Create, Read, Update, and Delete (CRUD) pages for the movie model.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In **Solution Explorer**, right-click the *Controllers* folder > **Add** > **New Scaffolded Item**.

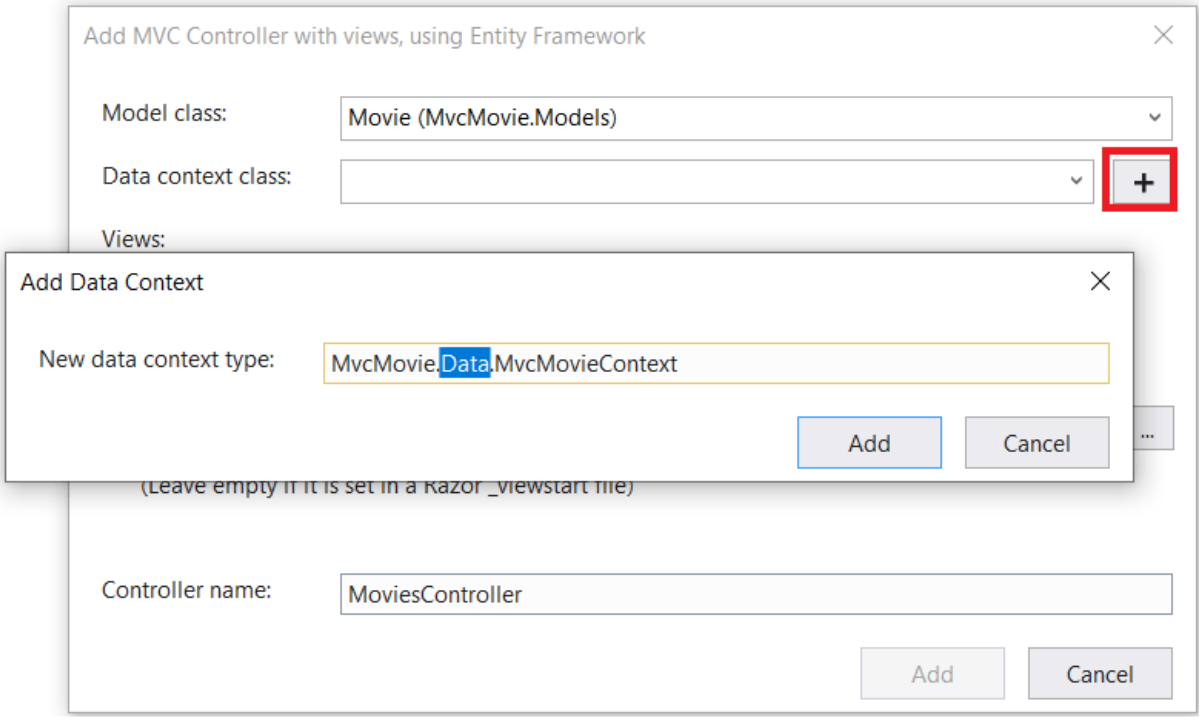


In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework** > **Add**.



Complete the **Add Controller** dialog:

- **Model class:** *Movie (MvcMovie.Models)*
- **Data context class:** *MvcMovieContext (MvcMovie.Data)*



- **Views:** Keep the default of each option checked
- **Controller name:** Keep the default *MoviesController*
- Select **Add**

Visual Studio creates:

- A movies controller (*Controllers/MoviesController.cs*)
- Razor view files for Create, Delete, Details, Edit, and Index pages (*Views/Movies/*.cshtml*)

The automatic creation of these files is known as *scaffolding*.

You can't use the scaffolded pages yet because the database doesn't exist. If you run the app and click on the **Movie App** link, you get a *Cannot open database* or *no such table: Movie* error message.

Initial migration

Use the EF Core Migrations feature to create the database. Migrations is a set of tools that let you create and update a database to match your data model.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).

In the PMC, enter the following commands:

```
Add-Migration InitialCreate
Update-Database
```

- `Add-Migration InitialCreate`: Generates a *Migrations/{timestamp}_InitialCreate.cs* migration file. The `InitialCreate` argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the `MvcMovieContext` class.

- `Update-Database`: Updates the database to the latest migration, which the previous command created. This command runs the `Up` method in the *Migrations/{time-stamp}_InitialCreate.cs* file, which creates the database.

  The database update command generates the following warning:

  > No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.

  You can ignore that warning, it will be fixed in a later tutorial.

For more information on the PMC tools for EF Core, see EF Core tools reference - PMC in Visual Studio.

The InitialCreate class

Examine the *Migrations/{timestamp}_InitialCreate.cs* migration file:

```
public partial class Initial : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Movie",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
                                SqlServerValueGenerationStrategy.IdentityColumn),
                Title = table.Column<string>(nullable: true),
                ReleaseDate = table.Column<DateTime>(nullable: false),
                Genre = table.Column<string>(nullable: true),
                Price = table.Column<decimal>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Movie", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Movie");
    }
}
```

The `Up` method creates the Movie table and configures `Id` as the primary key. The `Down` method reverts the schema changes made by the `Up` migration.

Test the app

- Run the app and click the **Movie App** link.

  If you get an exception similar to one of the following:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
SqlException: Cannot open database "MvcMovieContext-1" requested by the login. The login failed.
```

You probably missed the migrations step.

- Test the **Create** page. Enter and submit data.

> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** pages.

Dependency injection in the controller

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Open the *Controllers/MoviesController.cs* file and examine the constructor:

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the `MoviesController` class and views.

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller (the first URL segment).
- The action to `details` (the second URL segment).
- The id to 1 (the last URL segment).

You can also pass in the `id` with a query string as follows:

`https://localhost:5001/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in case an ID value isn't provided.

A lambda expression is passed in to `FirstOrDefaultAsync` to select movie entities that match the route data or query string value.

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the *Views/Movies/Details.cshtml* file:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

The `@model` statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following `@model` statement was included:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows access to the movie that the controller passed to the view. The `Model` object is strongly typed. For example, in the *Details.cshtml* view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the *Index.cshtml* view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

When the movies controller was created, scaffolding included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each item in the loop is typed as `Movie`. Among other benefits, this means that you get compile time checking of the code.

Additional resources

- Tag Helpers
- Globalization and localization

Add a data model class

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Right-click the *Models* folder > **Add** > **Class**. Name the class **Movie**.

Add the following properties to the `Movie` class:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

The `Movie` class contains:

- The `Id` field which is required by the database for the primary key.
- `[DataType(DataType.Date)]`: The DataType attribute specifies the type of the data (`Date`). With this attribute:
  - The user is not required to enter time information in the date field.
  - Only the date is displayed, not time information.

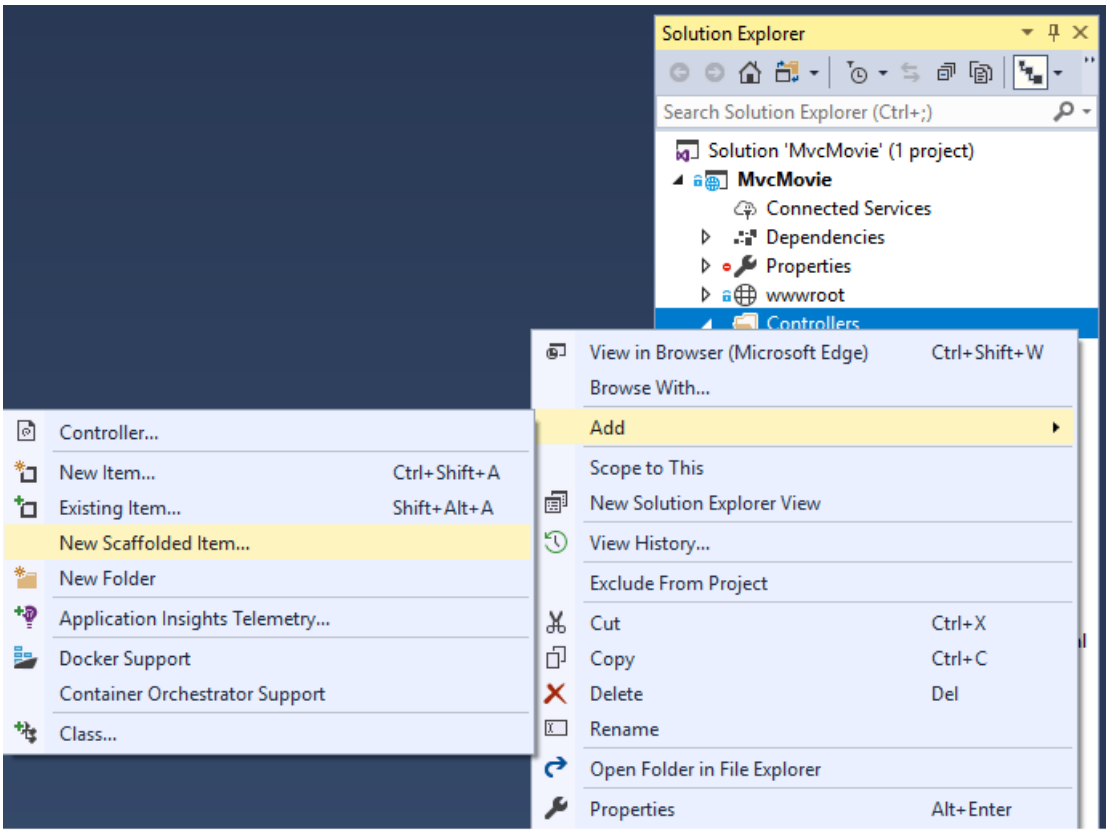
DataAnnotations are covered in a later tutorial.

Scaffold the movie model

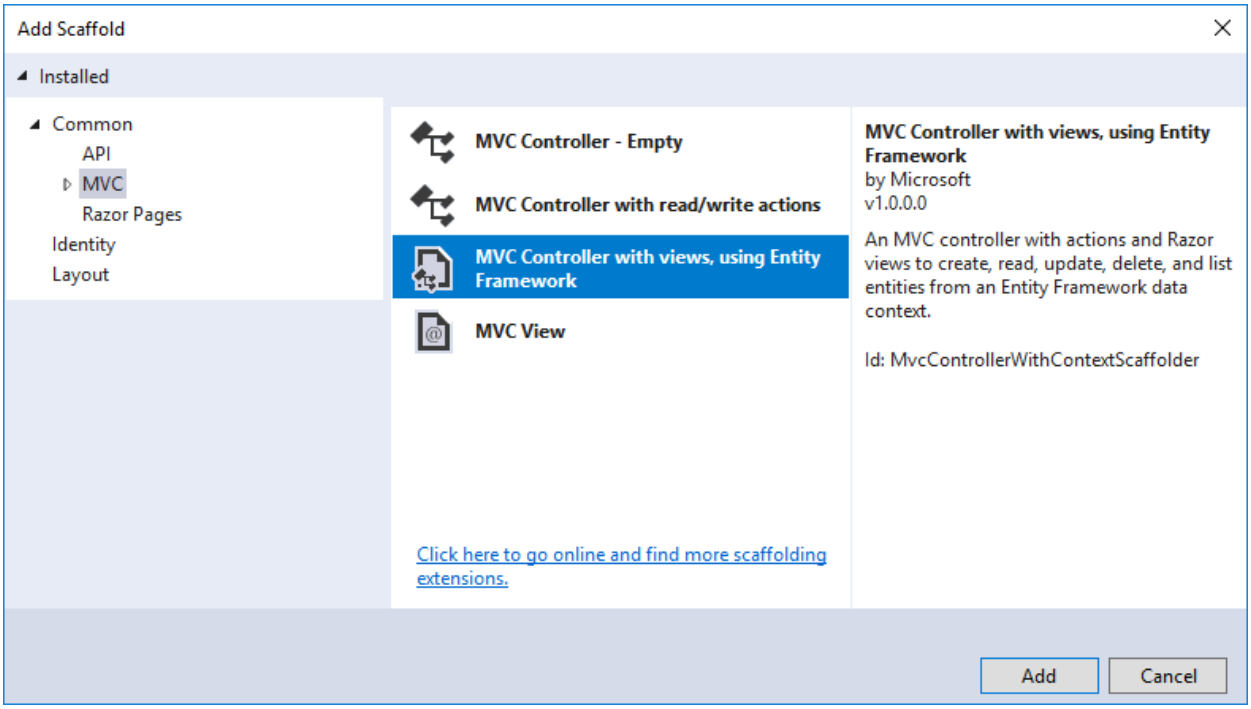
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

In **Solution Explorer**, right-click the *Controllers* folder > **Add** > **New Scaffolded Item**.

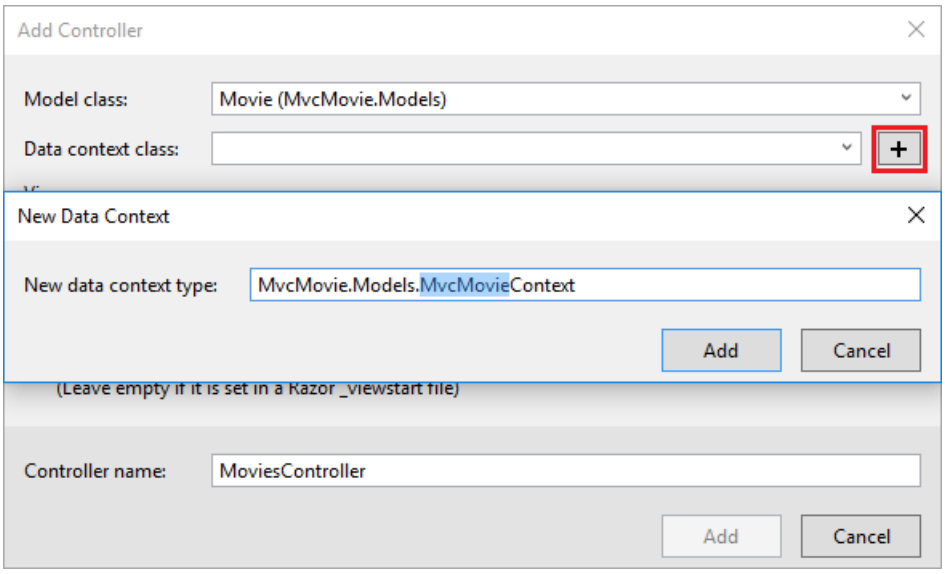


In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework > Add**.

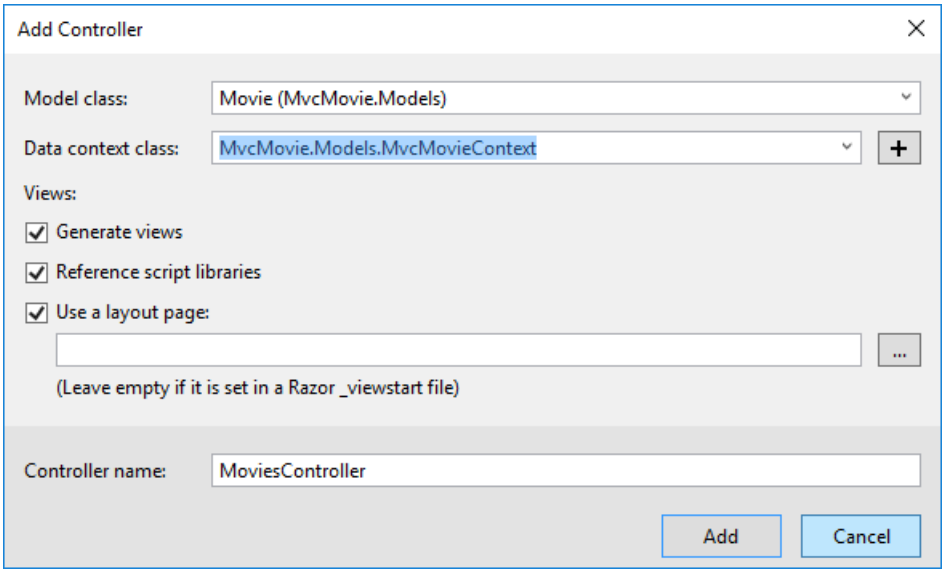


Complete the **Add Controller** dialog:

- **Model class:** *Movie (MvcMovie.Models)*
- **Data context class:** Select the + icon and add the default **MvcMovie.Models.MvcMovieContext**



- **Views:** Keep the default of each option checked
- **Controller name:** Keep the default *MoviesController*
- Select **Add**



Visual Studio creates:

- An Entity Framework Core database context class (*Data/MvcMovieContext.cs*)
- A movies controller (*Controllers/MoviesController.cs*)
- Razor view files for Create, Delete, Details, Edit, and Index pages (*Views/Movies/*.cshtml*)

The automatic creation of the database context and CRUD (create, read, update, and delete) action methods and views is known as *scaffolding*.

If you run the app and click on the **Mvc Movie** link, you get an error similar to the following:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
An unhandled exception occurred while processing the request.

SqlException: Cannot open database "MvcMovieContext-<GUID removed>" requested by the login. The login
failed.
Login failed for user 'Rick'.

System.Data.SqlClient.SqlInternalConnectionTds..ctor(DbConnectionPoolIdentity identity, SqlConnectionString
```

You need to create the database, and you use the EF Core Migrations feature to do that. Migrations lets you create a database that matches your data model and update the database schema when your data model changes.

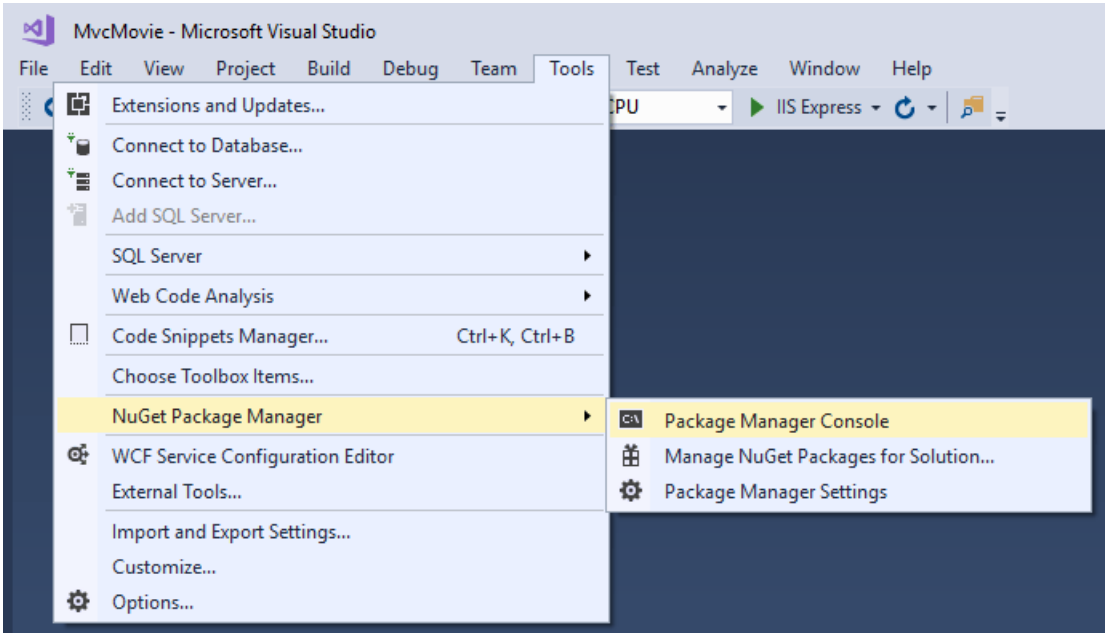
Initial migration

In this section, the following tasks are completed:

- Add an initial migration.
- Update the database with the initial migration.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

1. From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** (PMC).



2. In the PMC, enter the following commands:

```
Add-Migration Initial
Update-Database
```

The `Add-Migration` command generates code to create the initial database schema.

The database schema is based on the model specified in the `MvcMovieContext` class. The `Initial` argument is the migration name. Any name can be used, but by convention, a name that describes the migration is used. For more information, see Tutorial: Using the migrations feature - ASP.NET MVC with EF Core.

The `Update-Database` command runs the `Up` method in the *Migrations/{time-stamp}_InitialCreate.cs* file, which creates the database.

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection (DI). Services (such as the EF Core DB context) are registered with DI during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

The scaffolding tool automatically created a DB context and registered it with the DI container.

Examine the following `Startup.ConfigureServices` method. The highlighted line was added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The `MvcMovieContext` coordinates EF Core functionality (Create, Read, Update, Delete, etc.) for the `Movie` model. The data context (`MvcMovieContext`) is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model:

```
// Unused usings removed.
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;  // Enables public DbSet<Movie> Movie

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Test the app

- Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

If you get a database exception similar to the following:

```
SqlException: Cannot open database "MvcMovieContext-GUID" requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

- Test the **Create** link. Enter and submit data.

> **NOTE**
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

- Test the **Edit**, **Details**, and **Delete** links.

Examine the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The preceding highlighted code shows the movie database context being added to the Dependency Injection container:

- `services.AddDbContext<MvcMovieContext>(options =>` specifies the database to use and the connection string.
- `=>` is a lambda operator

Open the *Controllers/MoviesController.cs* file and examine the constructor:

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

Strongly typed models and the @model keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables better compile time checking of your code. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the `MoviesController` class and views when it created the methods and views.

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller (the first URL segment).
- The action to `details` (the second URL segment).
- The id to 1 (the last URL segment).

You can also pass in the `id` with a query string as follows:

`https://localhost:5001/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in case an ID value isn't provided.

A lambda expression is passed in to `FirstOrDefaultAsync` to select movie entities that match the route data or query string value.

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

```
return View(movie);
```

Examine the contents of the *Views/Movies/Details.cshtml* file:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

By including a `@model` statement at the top of the view file, you can specify the type of object that the view expects. When you created the movie controller, the following `@model` statement was automatically included at the top of the *Details.cshtml* file:

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows you to access the movie that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Details.cshtml* view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the *Index.cshtml* view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

When you created the movies controller, scaffolding automatically included the following `@model` statement at the top of the *Index.cshtml* file:

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows you to access the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the *Index.cshtml* view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each item in the loop is typed as `Movie`. Among other benefits, this means that you get compile time checking of the code:

Additional resources

- Tag Helpers
- Globalization and localization

PREVIOUS ADDING A VIEW

NEXT WORKING WITH A DATABASE

Part 5, work with a database in an ASP.NET Core MVC app

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in the *Startup.cs* file:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();

    services.AddDbContext<MvcMovieContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file:

```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

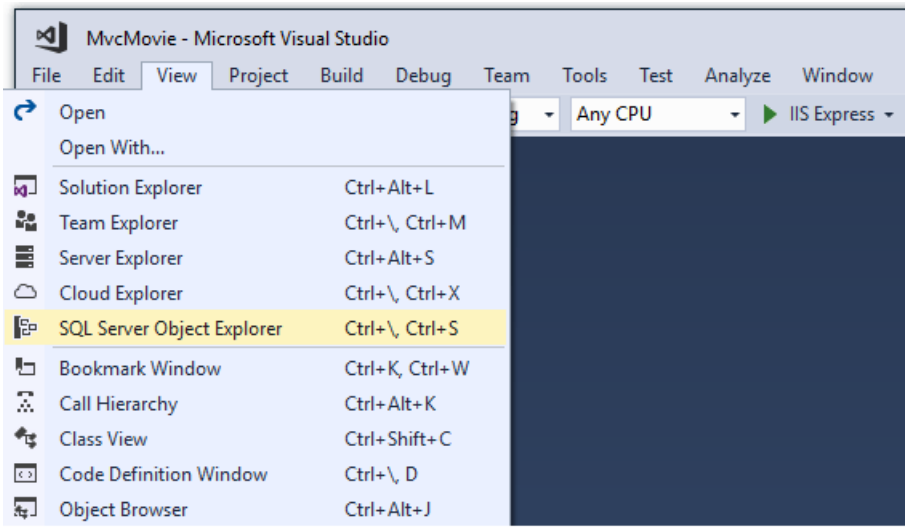
When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a production SQL Server. See Configuration for more information.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

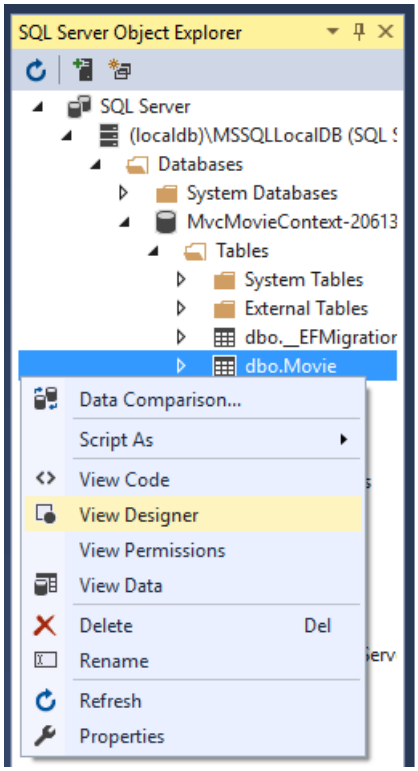
SQL Server Express LocalDB

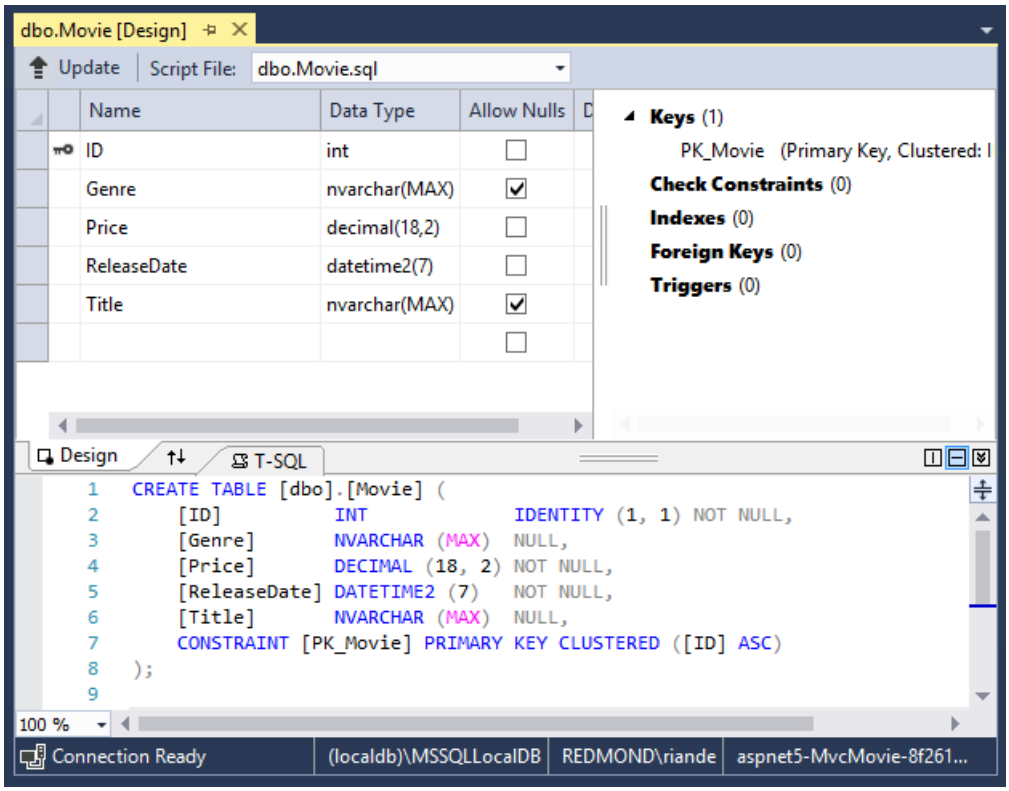
LocalDB is a lightweight version of the SQL Server Express Database Engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates *.mdf* files in the *C:/Users/{user}* directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



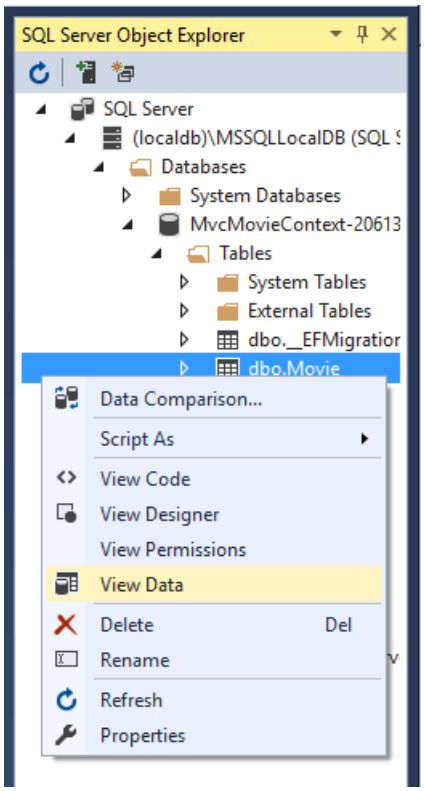
- Right click on the `Movie` table > **View Designer**

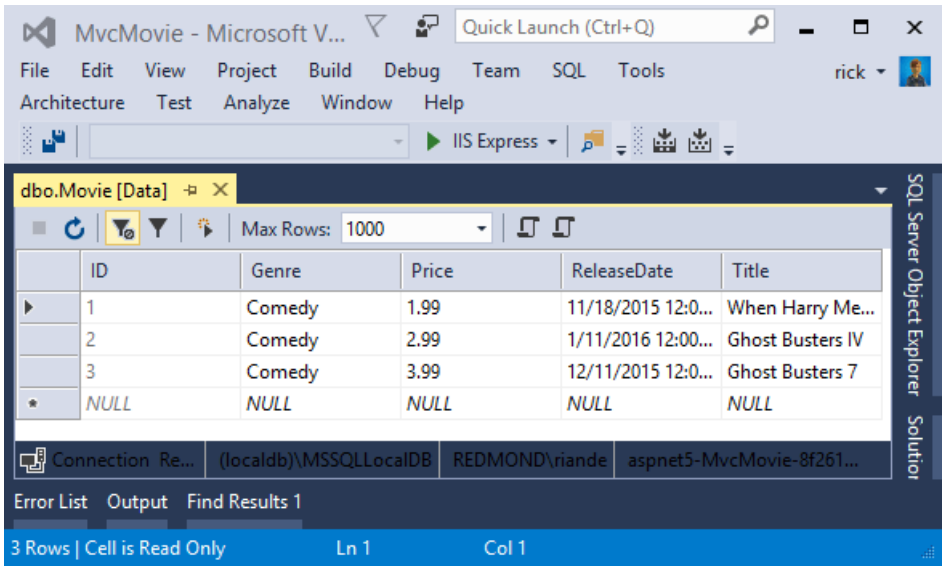




Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.

- Right click on the `Movie` table > **View Data**





Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

Replace the contents of *Program.cs* with the following code:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using MvcMovie.Data;
using MvcMovie.Models;
using System;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();

        }

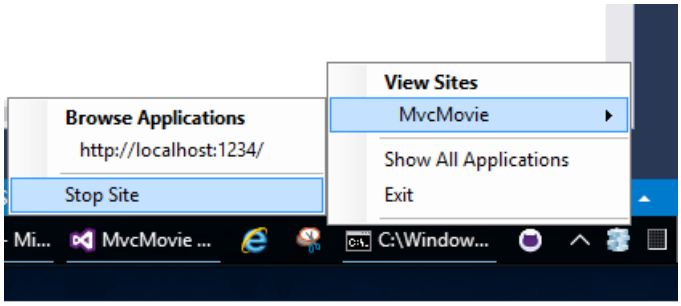
        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Test the app

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

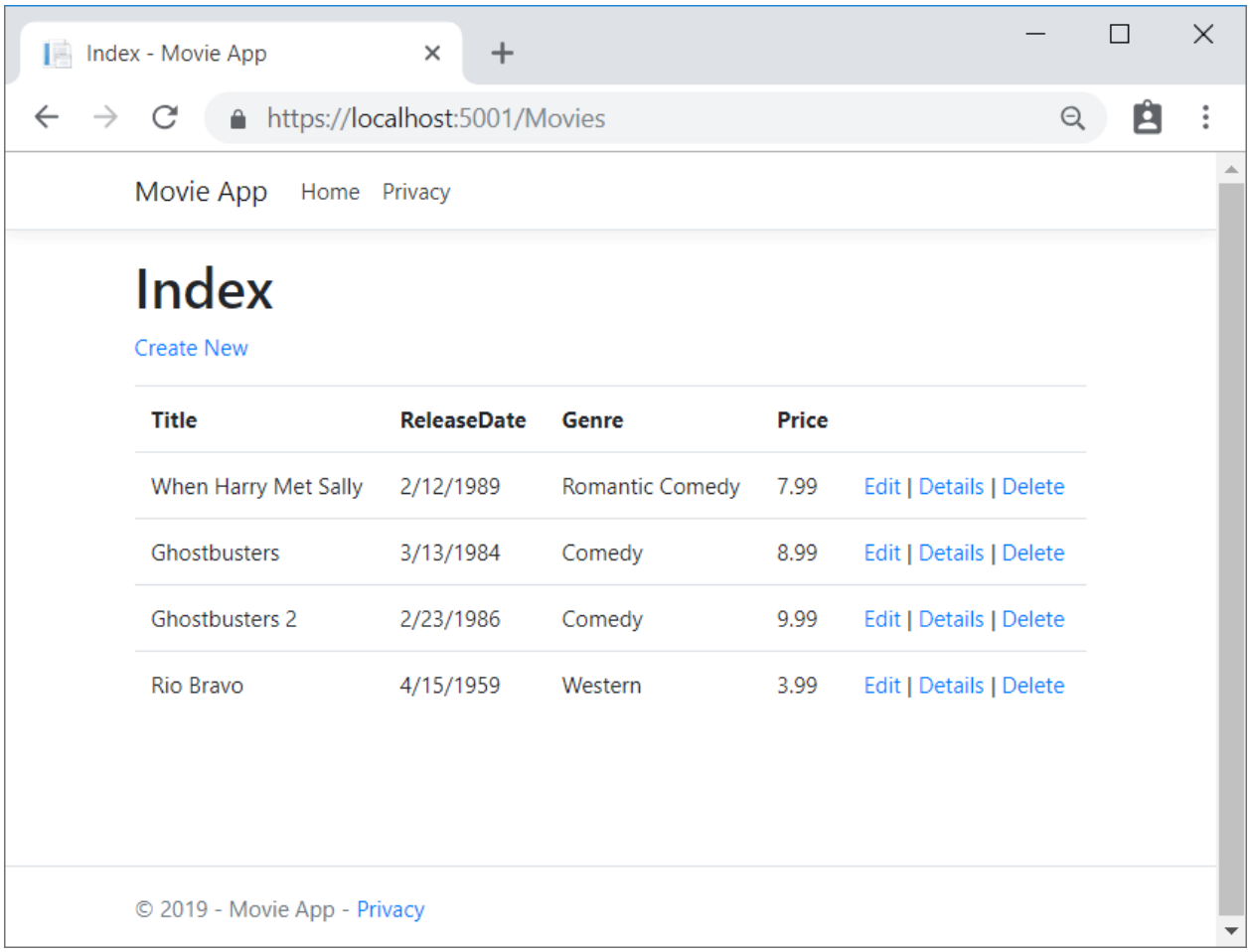
- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:

- Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**



  - If you were running VS in non-debug mode, press F5 to run in debug mode
  - If you were running VS in debug mode, stop the debugger and press F5

The app shows the seeded data.



PREVIOUS NEXT

By Rick Anderson

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to

database records. The database context is registered with the Dependency Injection container in the `ConfigureServices` method in the *Startup.cs* file:

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies
        // is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddDbContext<MvcMovieContext>(options =>
         options.UseSqlServer(Configuration.GetConnectionString("MvcMovieContext")));
}
```

The ASP.NET Core Configuration system reads the `ConnectionString`. For local development, it gets the connection string from the *appsettings.json* file:

```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

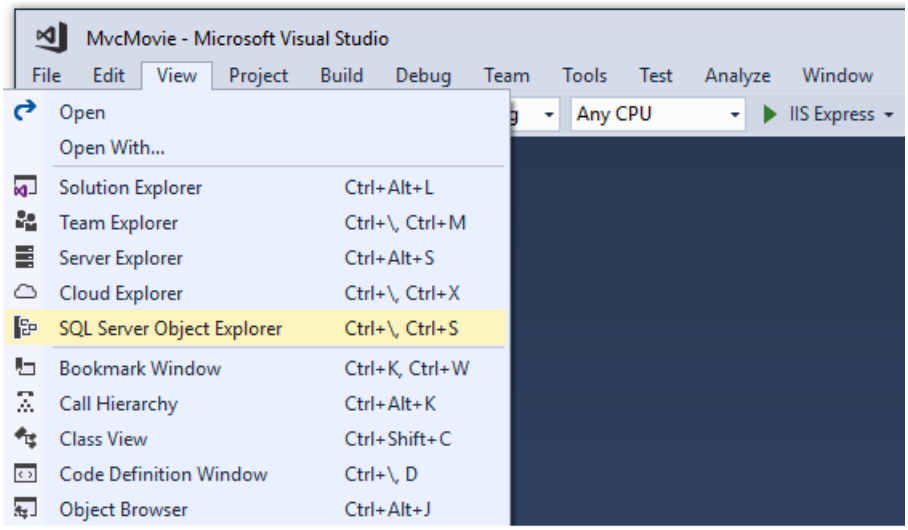
When you deploy the app to a test or production server, you can use an environment variable or another approach to set the connection string to a real SQL Server. See Configuration for more information.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

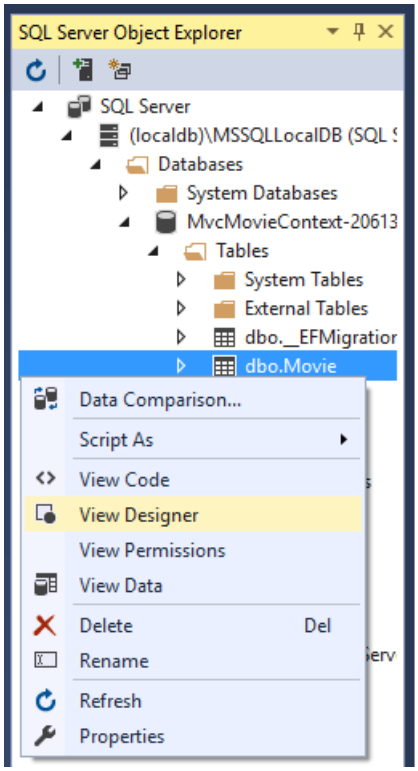
SQL Server Express LocalDB

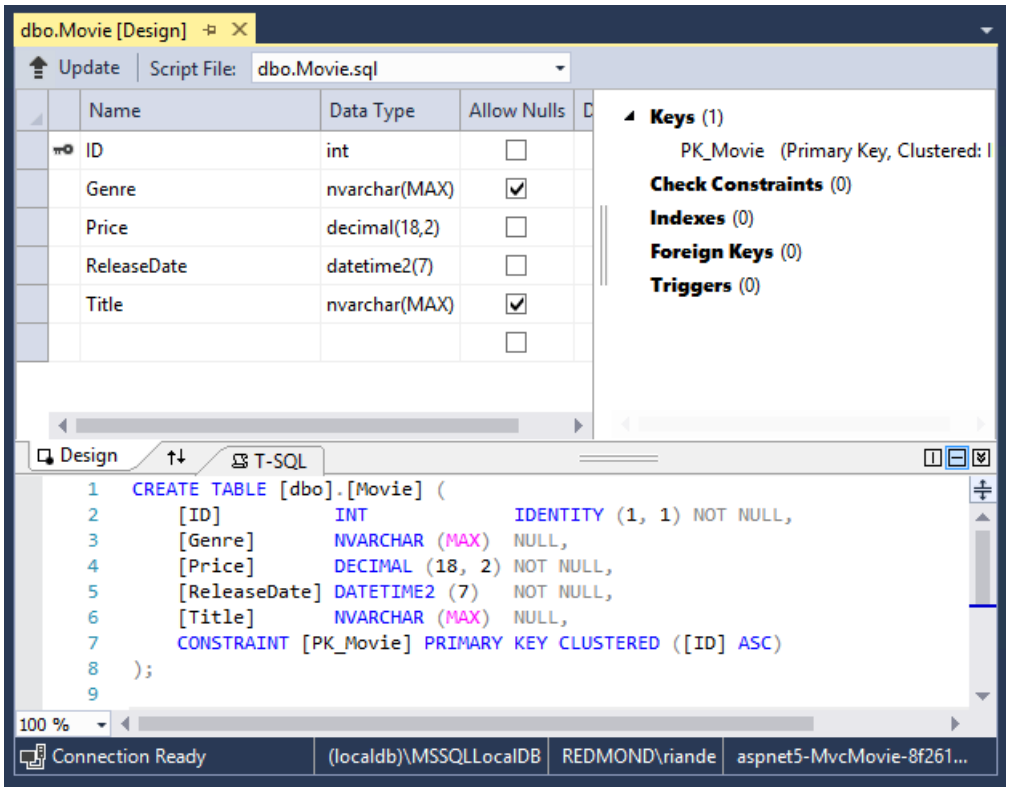
LocalDB is a lightweight version of the SQL Server Express Database Engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates *.mdf* files in the *C:/Users/{user}* directory.

- From the **View** menu, open **SQL Server Object Explorer** (SSOX).



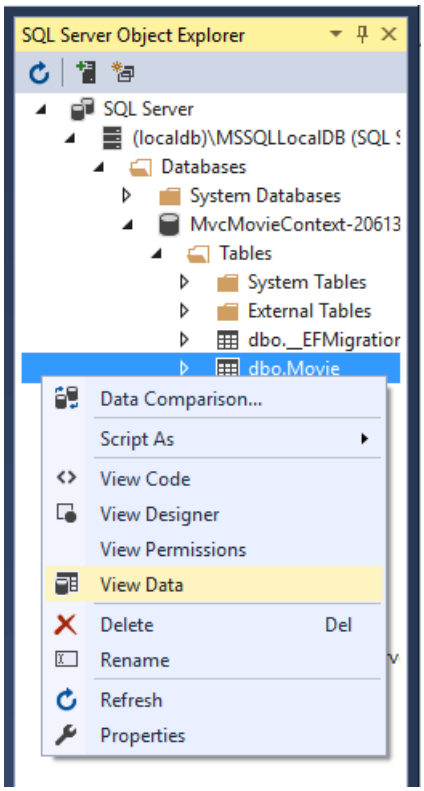
- Right click on the `Movie` table > **View Designer**

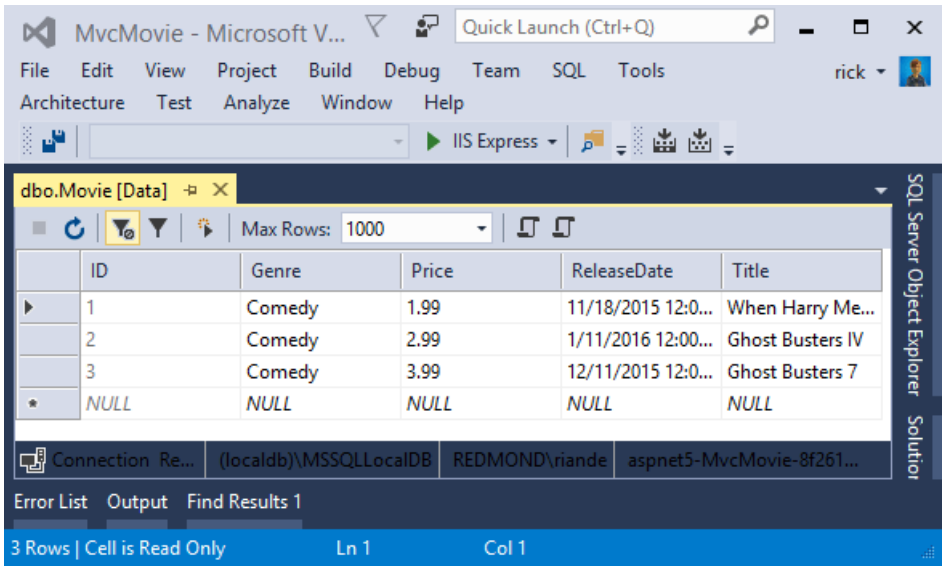




Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.

- Right click on the `Movie` table > **View Data**





Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;

namespace MvcMovie.Models
{
    public static class SeedData
    {
        public static void Initialize(IServiceProvider serviceProvider)
        {
            using (var context = new MvcMovieContext(
                serviceProvider.GetRequiredService<
                    DbContextOptions<MvcMovieContext>>()))
            {
                // Look for any movies.
                if (context.Movie.Any())
                {
                    return;   // DB has been seeded
                }

                context.Movie.AddRange(
                    new Movie
                    {
                        Title = "When Harry Met Sally",
                        ReleaseDate = DateTime.Parse("1989-2-12"),
                        Genre = "Romantic Comedy",
                        Price = 7.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters ",
                        ReleaseDate = DateTime.Parse("1984-3-13"),
                        Genre = "Comedy",
                        Price = 8.99M
                    },

                    new Movie
                    {
                        Title = "Ghostbusters 2",
                        ReleaseDate = DateTime.Parse("1986-2-23"),
                        Genre = "Comedy",
                        Price = 9.99M
                    },

                    new Movie
                    {
                        Title = "Rio Bravo",
                        ReleaseDate = DateTime.Parse("1959-4-15"),
                        Genre = "Western",
                        Price = 3.99M
                    }
                );
                context.SaveChanges();
            }
        }
    }
}
```

If there are any movies in the DB, the seed initializer returns and no movies are added.

```
if (context.Movie.Any())
{
    return;   // DB has been seeded.
}
```

Add the seed initializer

Replace the contents of *Program.cs* with the following code:

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using System;
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;
using MvcMovie;

namespace MvcMovie
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<MvcMovieContext>();
                    context.Database.Migrate();
                    SeedData.Initialize(services);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

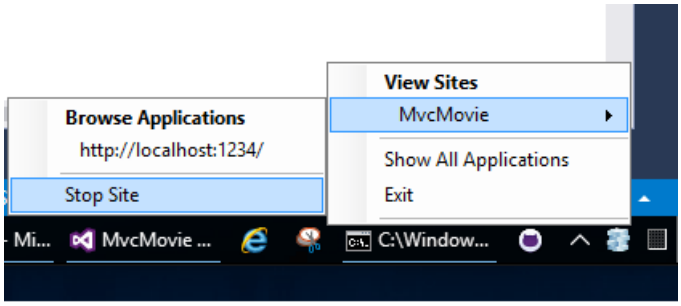
        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

Test the app

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

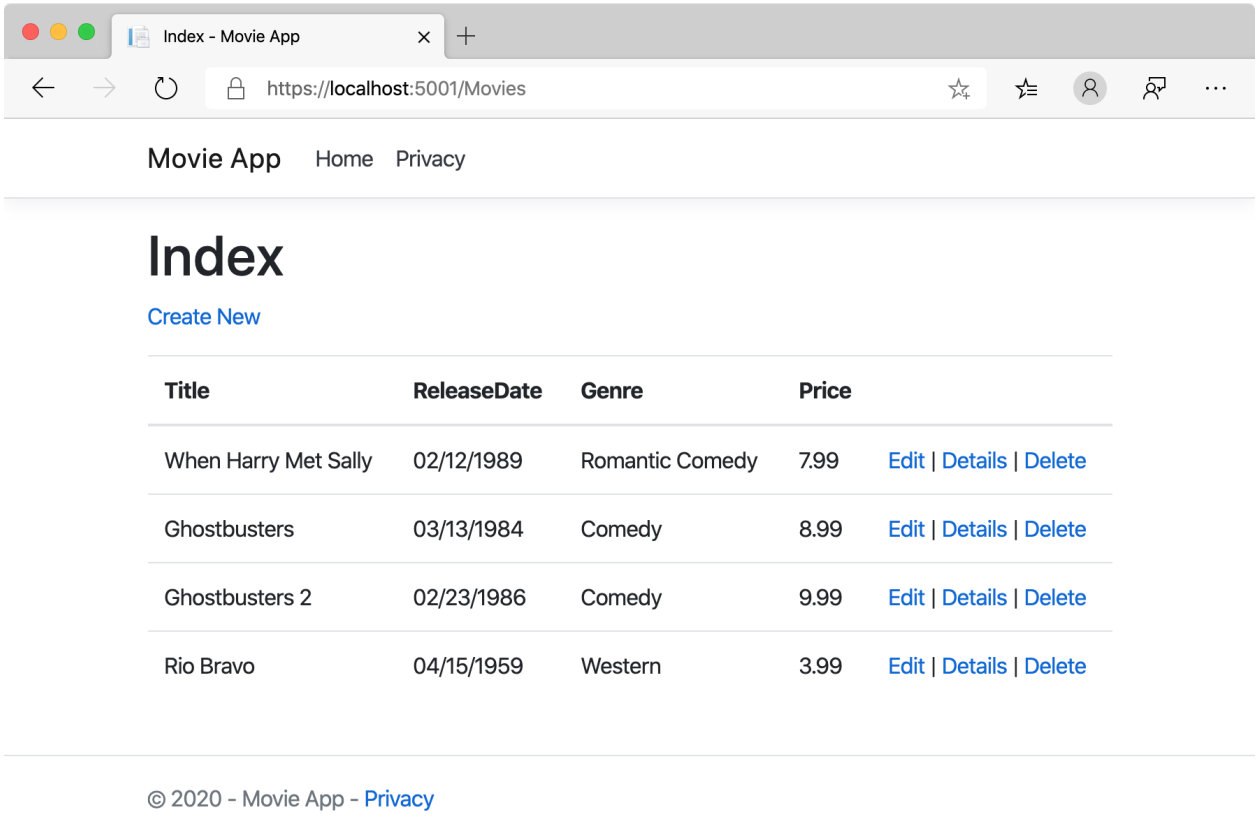
- Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.
- Force the app to initialize (call the methods in the `Startup` class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:

- Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**



  - If you were running VS in non-debug mode, press F5 to run in debug mode
  - If you were running VS in debug mode, stop the debugger and press F5

The app shows the seeded data.

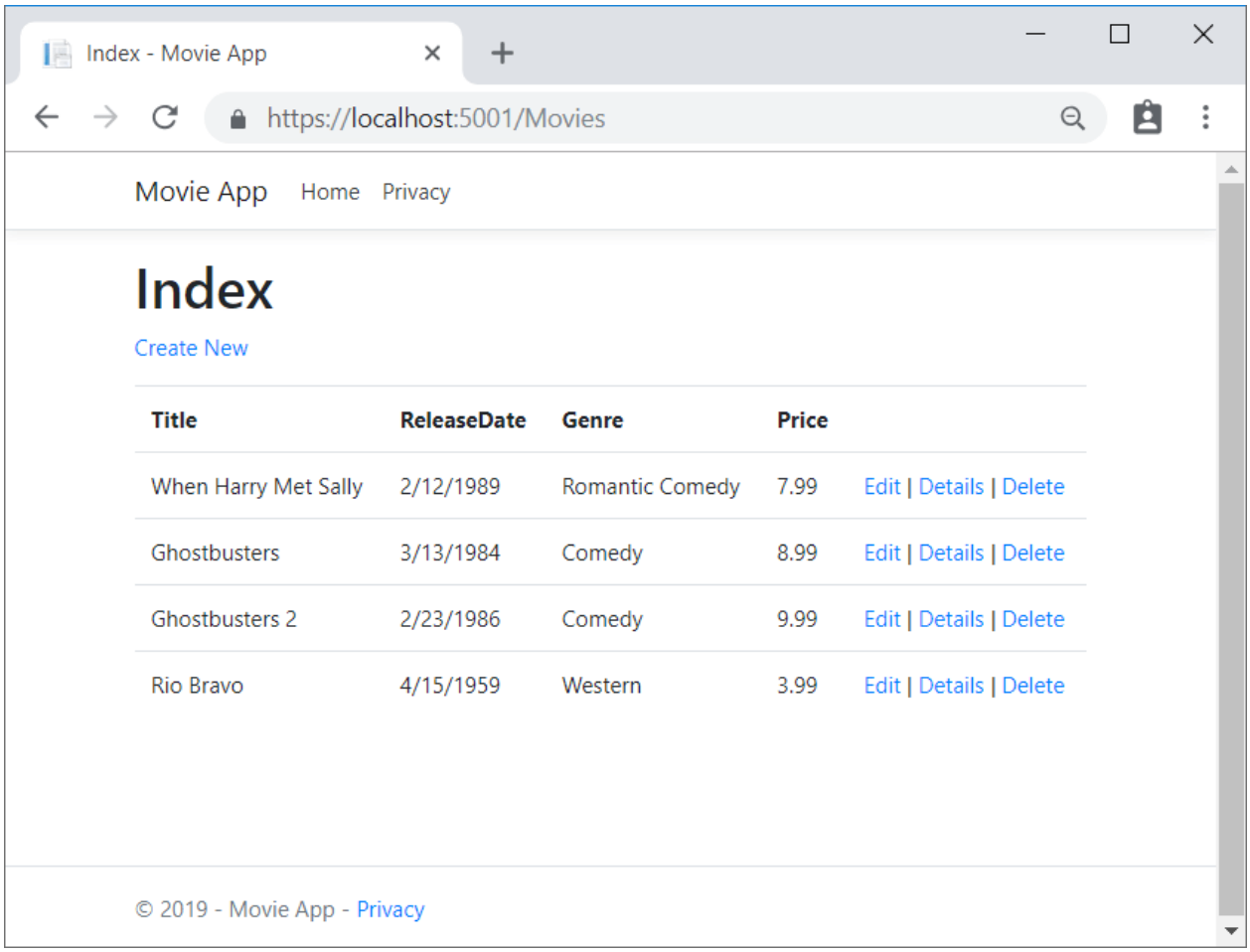


Part 6, controller methods and views in ASP.NET Core

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

We have a good start to the movie app, but the presentation isn't ideal, for example, **ReleaseDate** should be two words.



Open the *Models/Movie.cs* file and add the highlighted lines shown below:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }

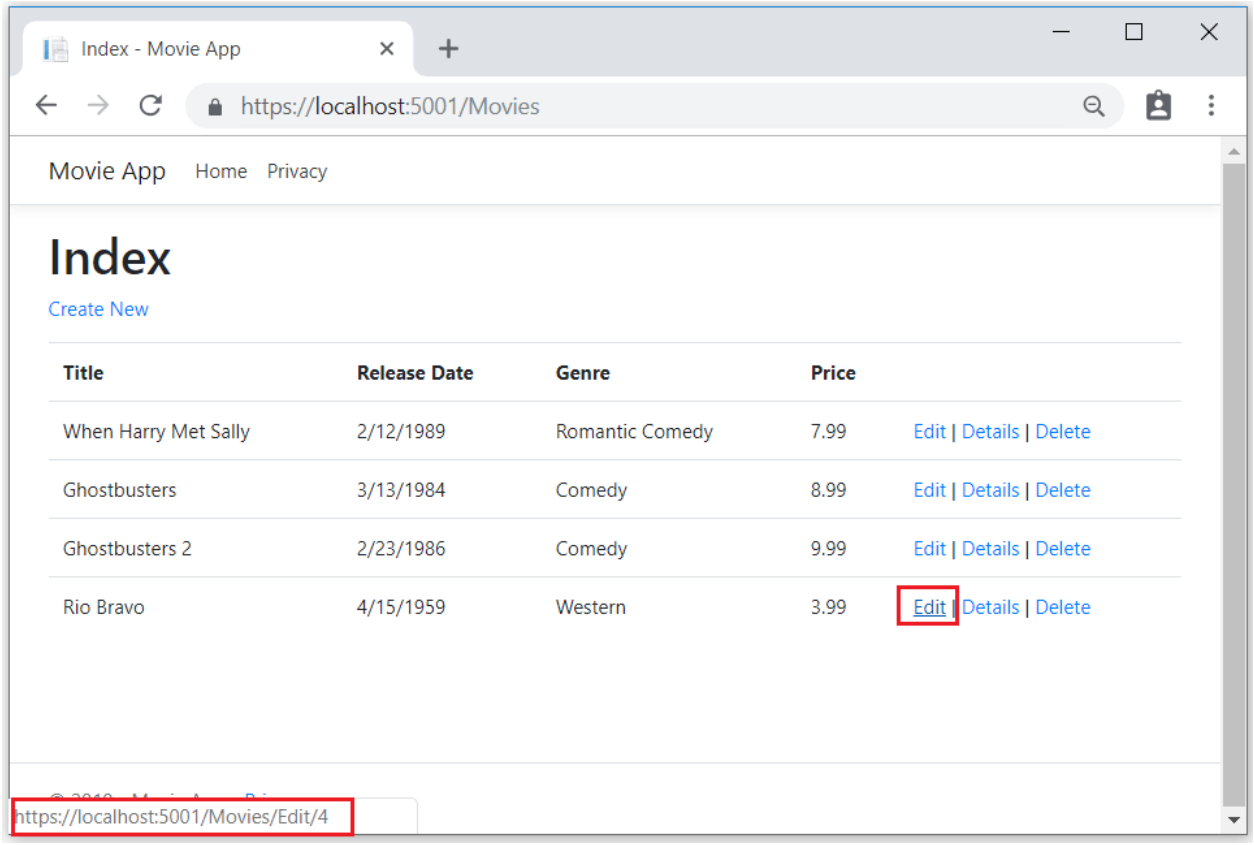
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
}
```

We cover DataAnnotations in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Core MVC Anchor Tag Helper in the *Views/Movies/Index.cshtml* file.

```
        <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
        <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
        <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
    </td>
</tr>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

```
 <td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for routing set in the *Startup.cs* file:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

ASP.NET Core translates `https://localhost:5001/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. For more information, see Additional resources.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the *Edit.cshtml* Razor file.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `[Bind]` attribute is one way to protect against over-posting. You should only include properties in the `[Bind]` attribute that you want to change. For more information, see Protect your controller from over-posting. ViewModels provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to prevent forgery of a request and is paired up with an anti-forgery token generated in the edit view file (*Views/Movies/Edit.cshtml*). The edit view file generates the anti-forgery token with the Form Tag Helper.

```
<form asp-action="Edit">
```

The Form Tag Helper generates a hidden anti-forgery token that must match the `[ValidateAntiForgeryToken]` generated anti-forgery token in the `Edit` method of the Movies controller. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `FindAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file. `@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The Label Tag Helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML `<input>` element. The Validation Tag Helper displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field is required." id="ID" name="ID"
value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre" name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Genre" data-valmsg-
replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true" data-val-number="The field Price
must be a number." data-val-required="The Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-valmsg-for="Price" data-valmsg-
replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The `<input>` elements are in an `HTML <form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the `Save` button is clicked. The last line before the closing `</form>` element shows the hidden XSRF token generated by the Form Tag Helper.

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

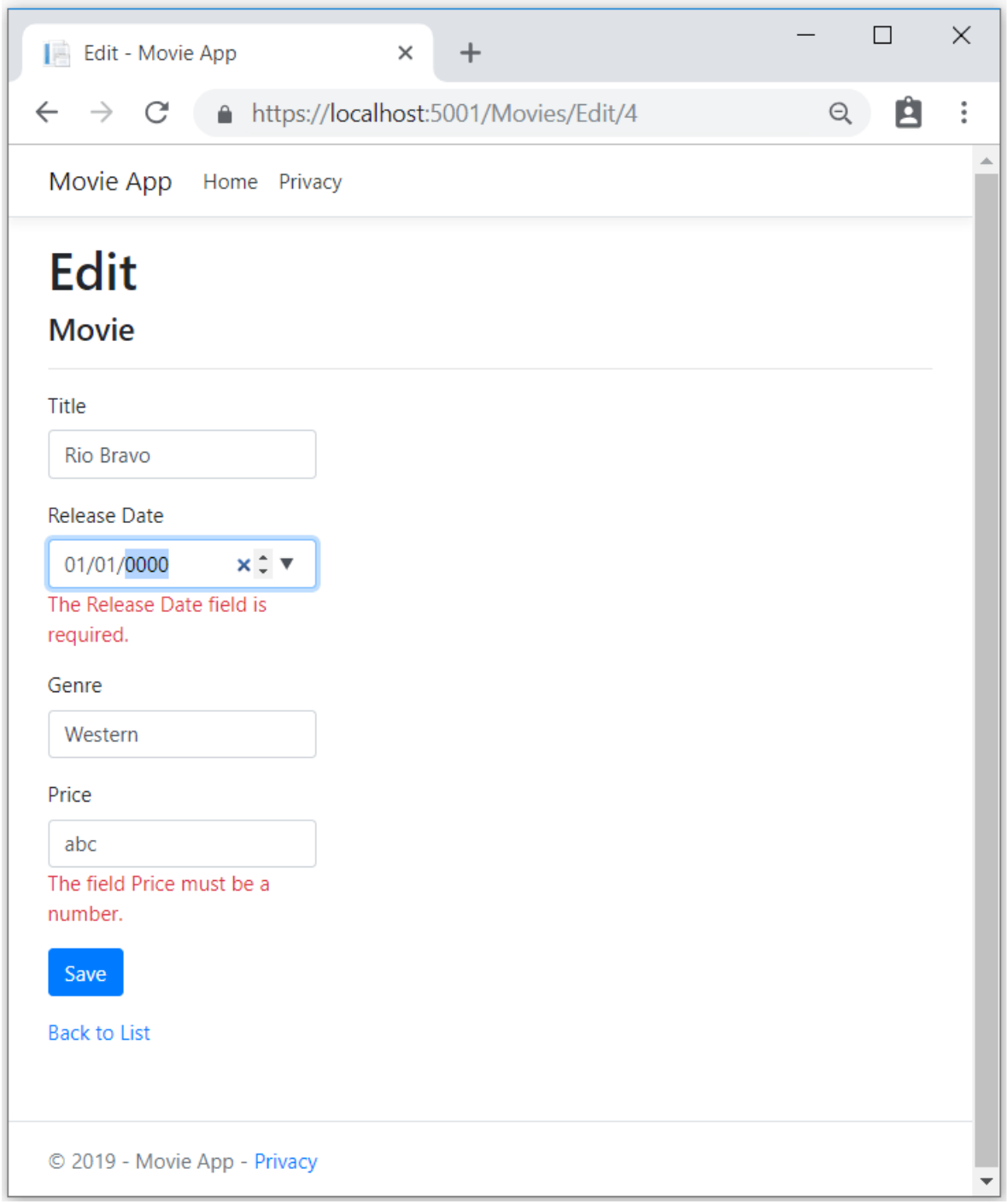
```
// POST: Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (id != movie.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the anti-forgery token generator in the Form Tag Helper

The model binding system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` method verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine Model Validation in more detail. The Validation Tag Helper in the *Views/Movies/Edit.cshtml* view template takes care of displaying appropriate error messages.



All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests shouldn't change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers
- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core
- Protect your controller from over-posting

- ViewModels
- Form Tag Helper
- Input Tag Helper
- Label Tag Helper
- Select Tag Helper
- Validation Tag Helper

PREVIOUS NEXT

Part 7, add search to an ASP.NET Core MVC app

9/22/2020 • 7 minutes to read • Edit Online

By Rick Anderson

In this section, you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method found inside *Controllers/MoviesController.cs* with the following code:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

The first line of the `Index` action method creates a LINQ query to select the movies:

```
var movies = from m in _context.Movie
             select m;
```

The query is *only* defined at this point, it has **not** been run against the database.

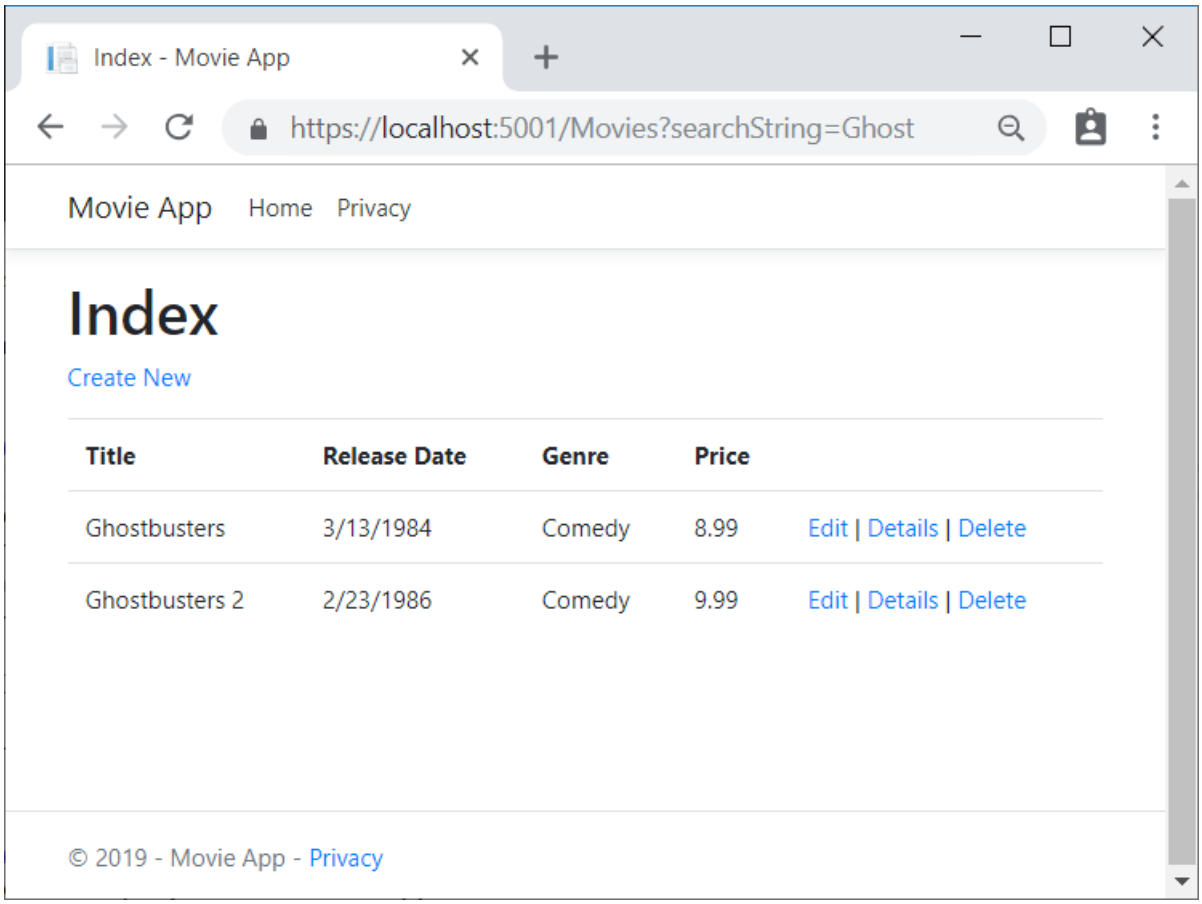
If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

The `s => s.Title.Contains()` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see Query Execution.

Note: The Contains method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, Contains maps to SQL LIKE, which is case insensitive. In SQLite, with the default collation, it's case sensitive.

Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in *Startup.cs*.

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Change the parameter to `id` and all occurrences of `searchString` change to `id`.

The previous `Index` method:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

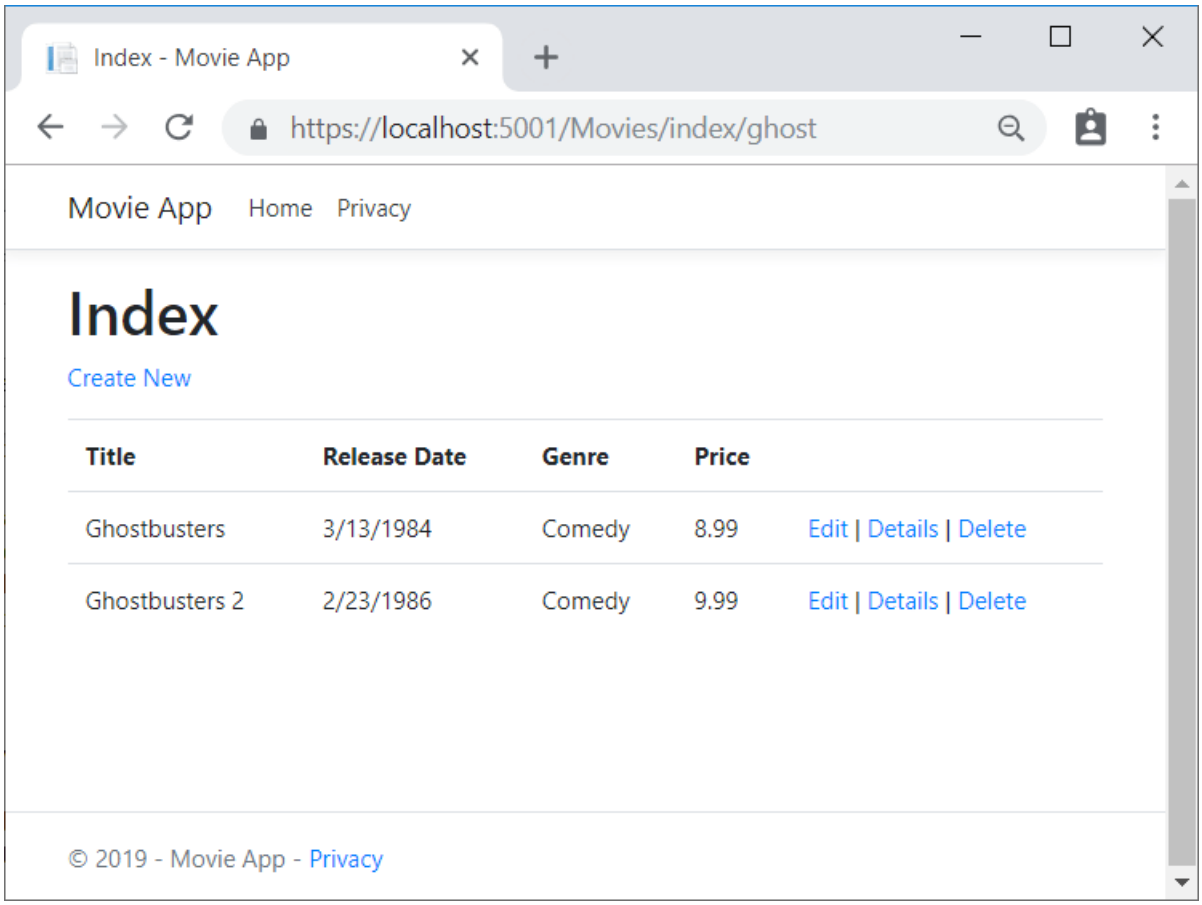
The updated `Index` method with `id` parameter:

```
public async Task<IActionResult> Index(string id)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s => s.Title.Contains(id));
    }

    return View(await movies.ToListAsync());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

```
public async Task<IActionResult> Index(string searchString)
{
    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    return View(await movies.ToListAsync());
}
```

Open the *Views/Movies/Index.cshtml* file, and add the `<form>` markup highlighted below:

```
    ViewData["Title"] = "Index";
}

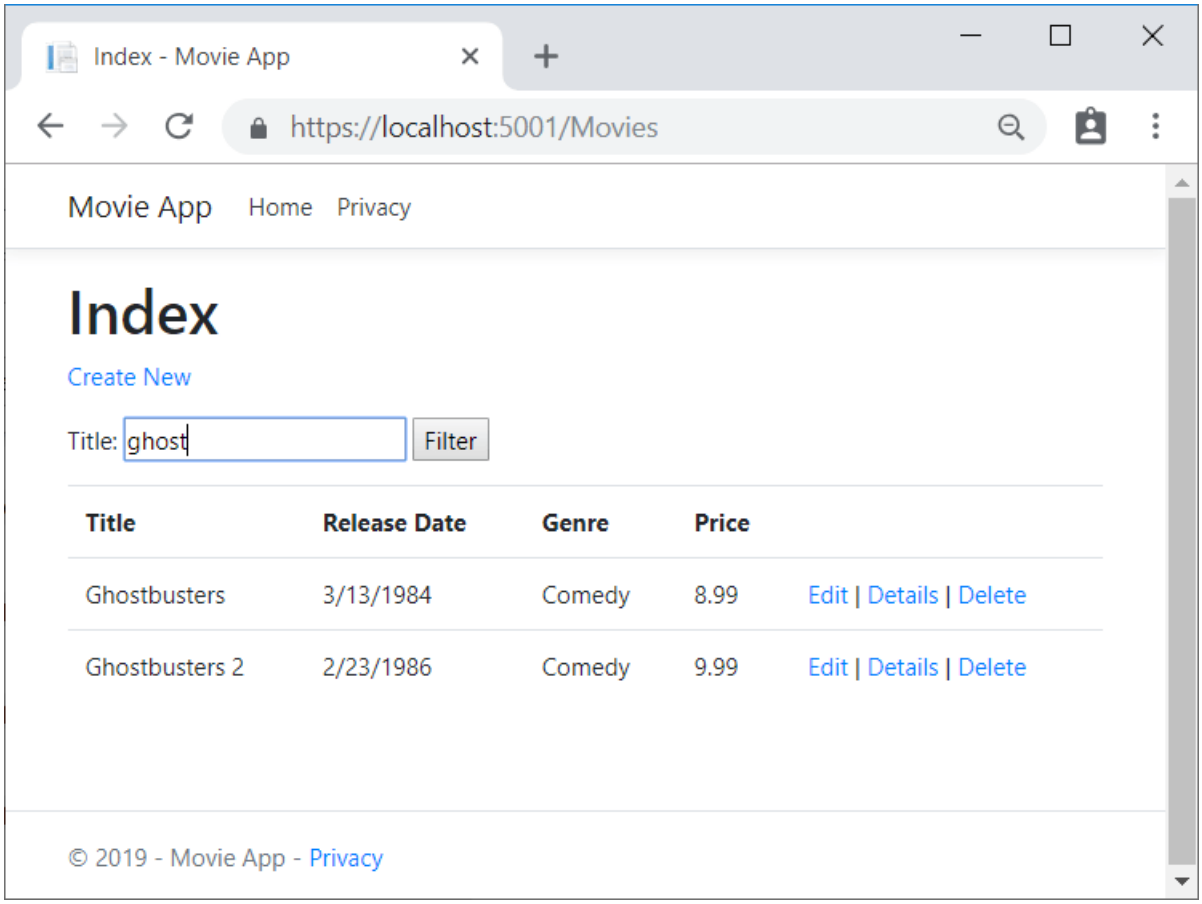
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString">
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
```

The HTML `<form>` tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.



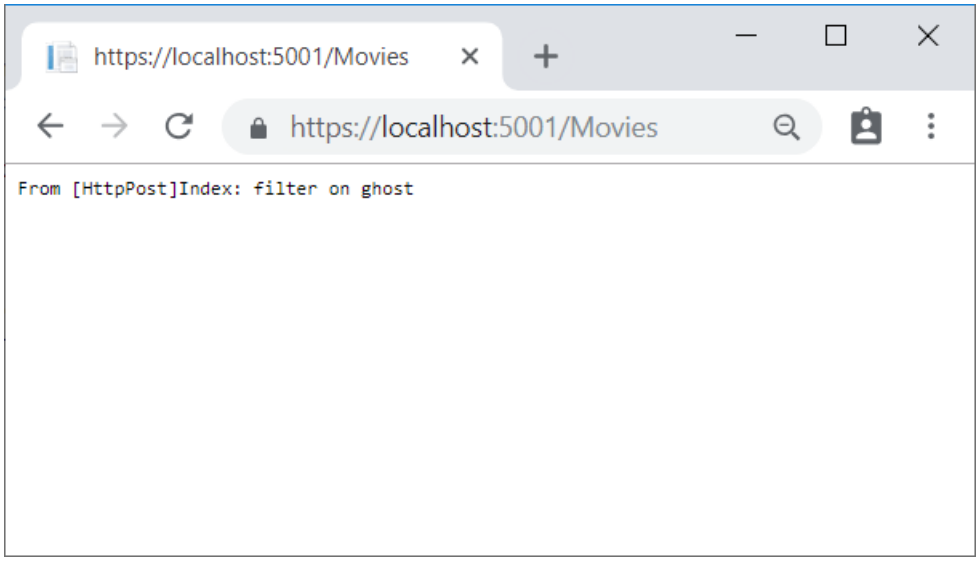
There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following `[HttpPost] Index` method.

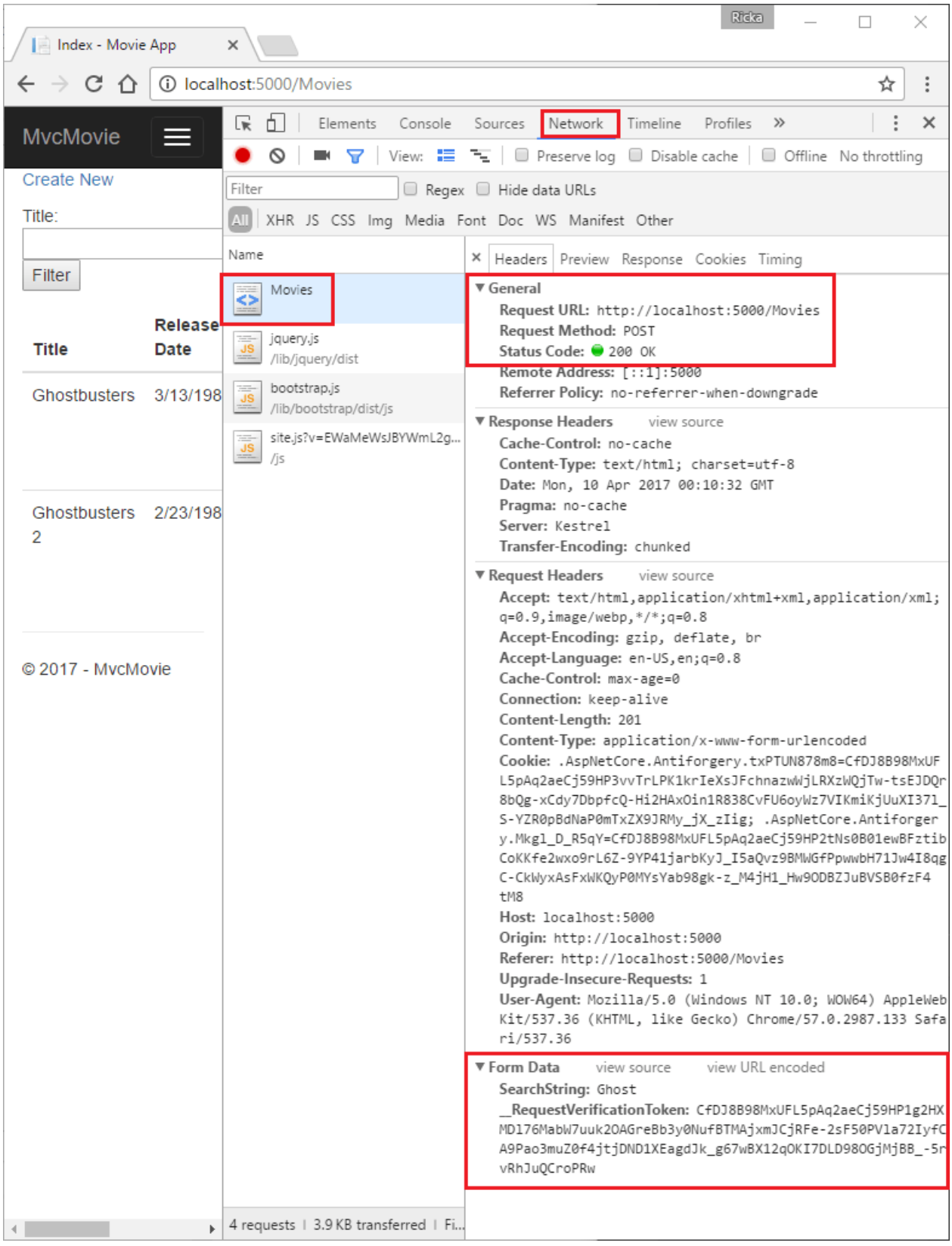
```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:{PORT}/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool. The image below shows the Chrome browser Developer tools:



You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF anti-forgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be `HTTP GET` found in the *Views/Movies/Index.cshtml* file.

```
@model IEnumerable<MvcMovie.Models.Movie>

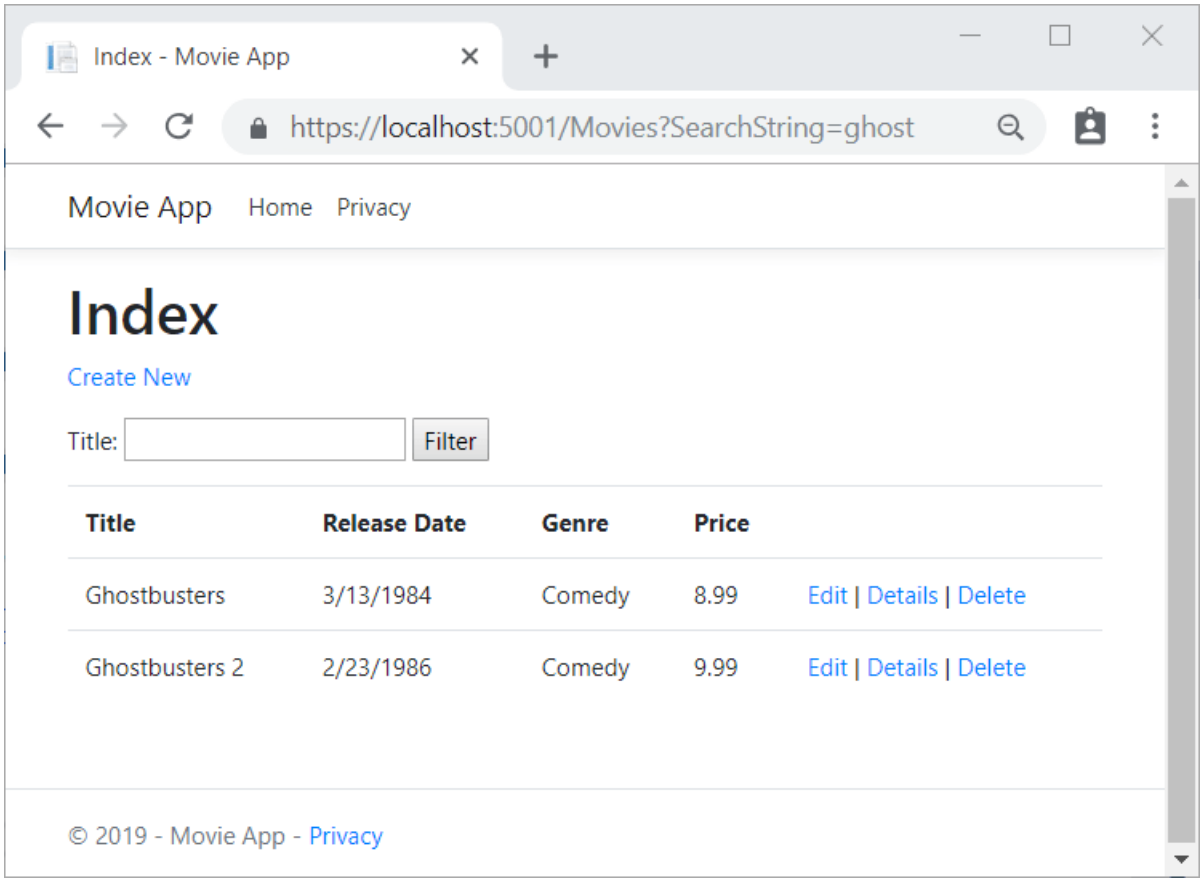
@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



The following markup shows the change to the `form` tag:

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Add Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models
{
    public class MovieGenreViewModel
    {
        public List<Movie> Movies { get; set; }
        public SelectList Genres { get; set; }
        public string MovieGenre { get; set; }
        public string SearchString { get; set; }
    }
}
```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This allows the user to select a genre from the list.
- `MovieGenre`, which contains the selected genre.
- `SearchString`, which contains the text users enter in the search text box.

Replace the `Index` method in `MoviesController.cs` with the following code:

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string searchString)
{
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s => s.Title.Contains(searchString));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

The following code is a `LINQ` query that retrieves all the genres from the database.

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

When the user searches for the item, the search value is retained in the search box.

Add search by genre to the Index view

Update `Index.cshtml` found in *Views/Movies/* as follows:

```
@model MvcMovie.Models.MovieGenreViewModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

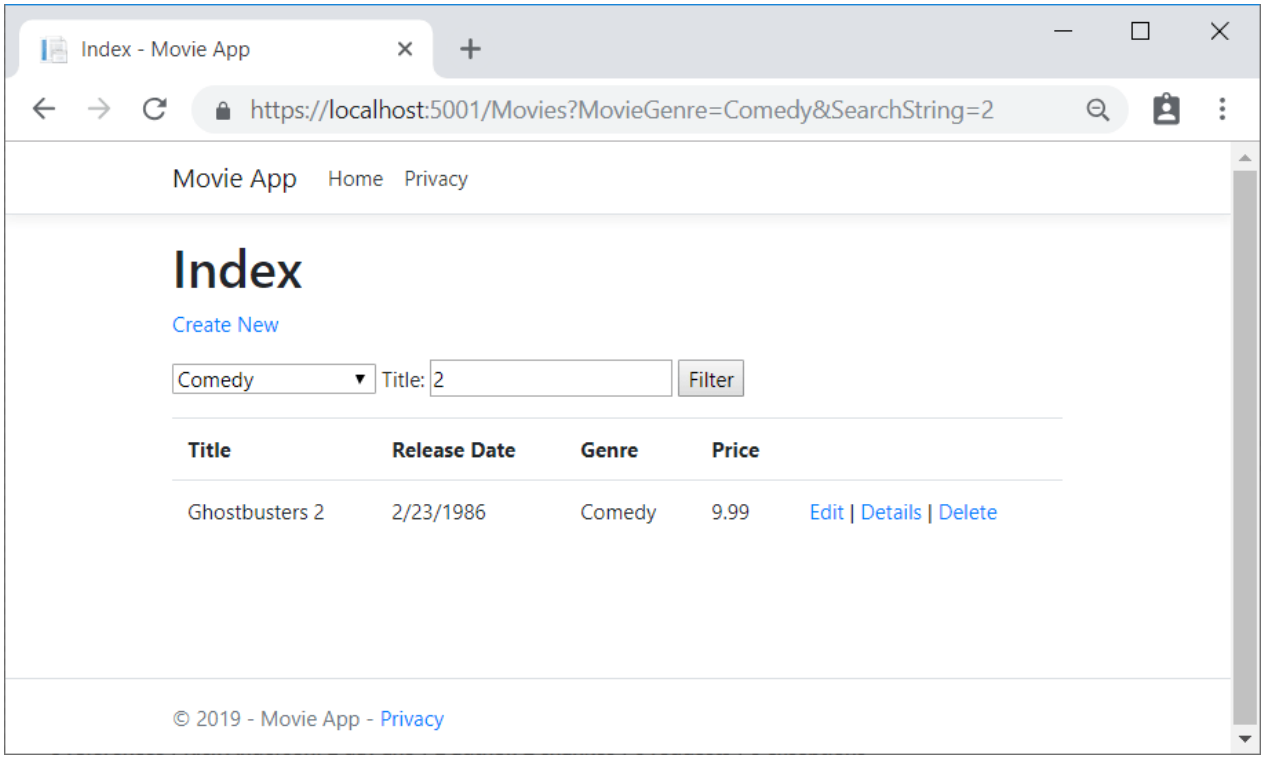
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movies[0].Title)
```

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.Movies`, or `model.Movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated.

Test the app by searching by genre, by movie title, and by both:



Part 8, add a new field to an ASP.NET Core MVC app

9/22/2020 • 4 minutes to read • Edit Online

By Rick Anderson

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When EF Code First is used to automatically create a database, Code First:

- Adds a table to the database to track the schema of the database.
- Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Add a Rating Property to the Movie Model

Add a `Rating` property to *Models/Movie.cs*:

```
public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Ctrl+Shift+B

Because you've added a new field to the `Movie` class, you need to update the binding white list so this new property will be included. In *MoviesController.cs*, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

Update the view templates in order to display, create, and edit the new `Rating` property in the browser view.

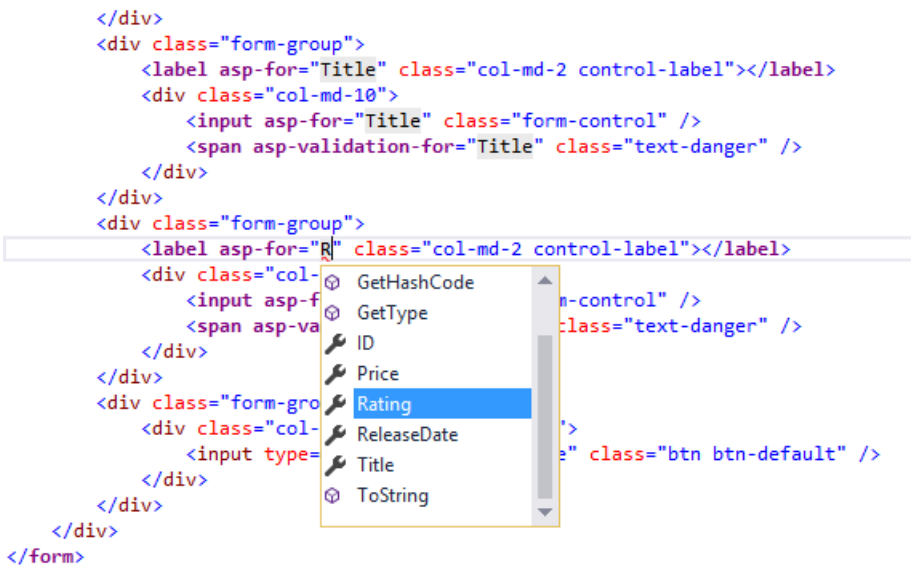
Edit the */Views/Movies/Index.cshtml* file and add a `Rating` field:

```
<thead>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Movies[0].Rating)
        </th>
        <th></th>
    </tr>
</thead>
<tbody>
    @foreach (var item in Model.Movies)
    {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
```

Update the */Views/Movies/Create.cshtml* with a `Rating` field.

- Visual Studio / Visual Studio for Mac
- Visual Studio Code

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers.



Update the remaining templates.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

The app won't work until the DB is updated to include the new field. If it's run now, the following `SqlException` is thrown:

```
SqlException: Invalid column name 'Rating'.
```

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.

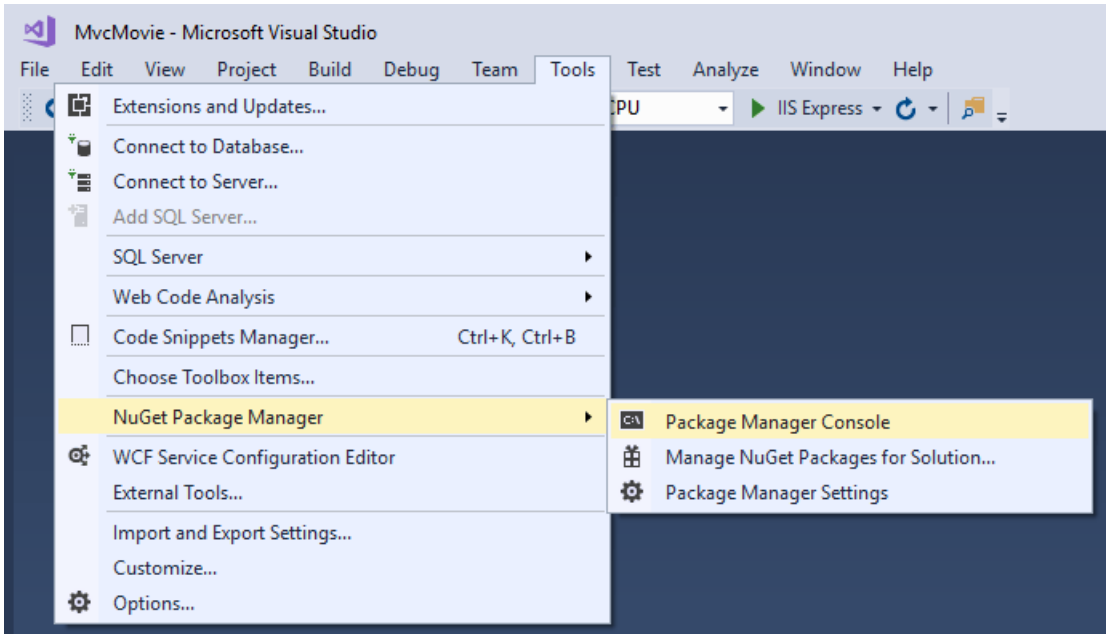
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

3. Use Code First Migrations to update the database schema.

For this tutorial, Code First Migrations is used.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the `Rating` field.

Run the app and verify you can create, edit, and display movies with a `Rating` field.

PREVIOUS NEXT

Part 9, add validation to an ASP.NET Core MVC app

9/22/2020 • 9 minutes to read • Edit Online

By Rick Anderson

In this section:

- Validation logic is added to the `Movie` model.
- You ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, `RegularExpression`, and `Range` validation attributes.

```
public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

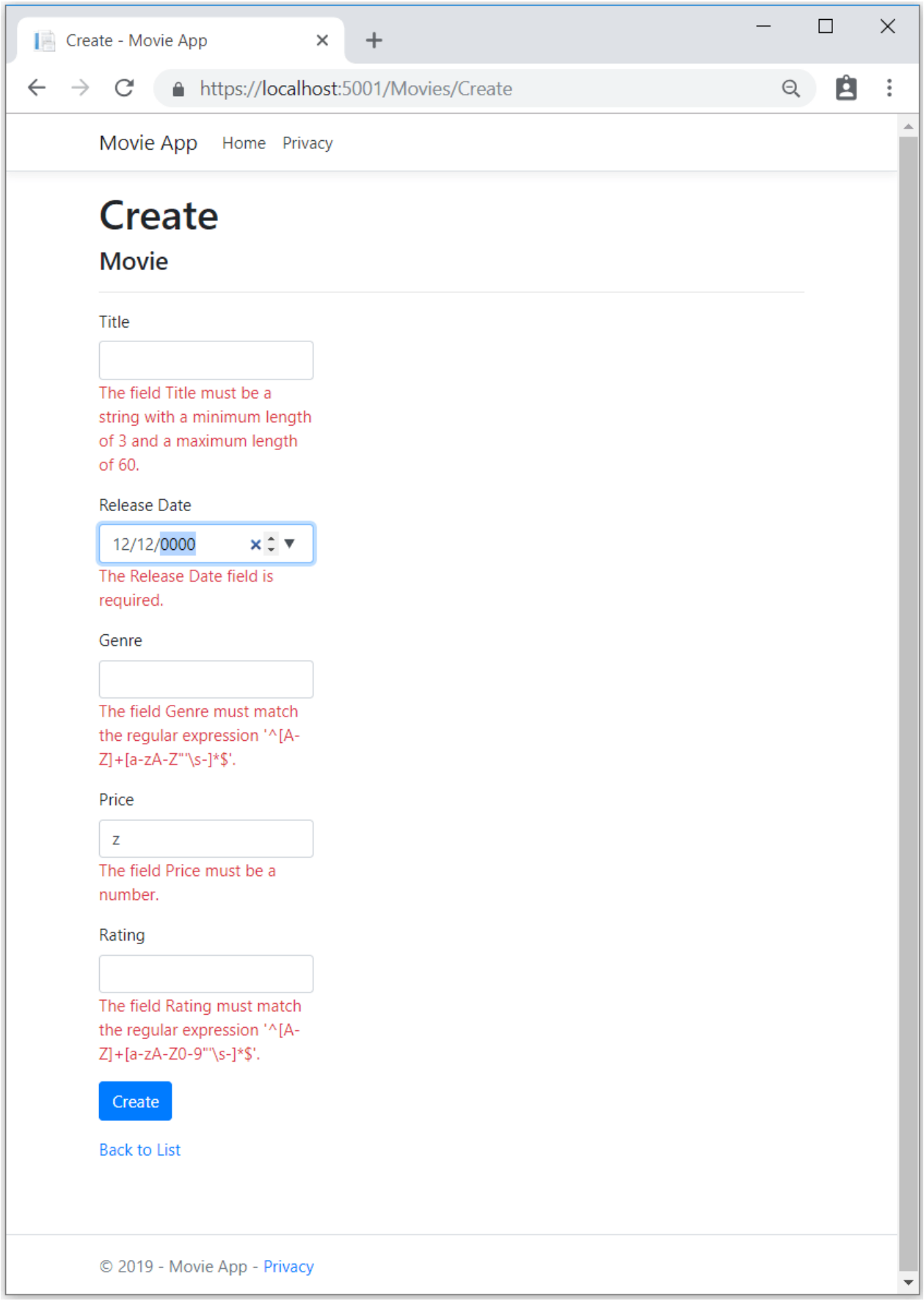
- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White space, numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI

Run the app and navigate to the Movies controller.

Tap the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



NOTE

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered an appropriate validation error message in each field containing

an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the *Create.cshtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data isn't sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the Fiddler tool , or the F12 Developer tools.

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

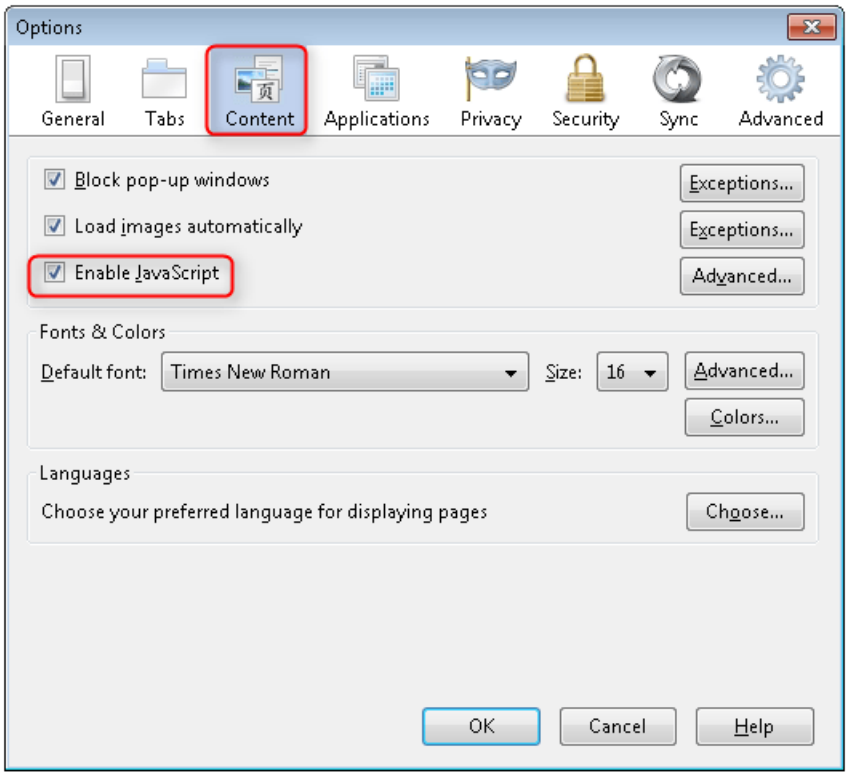
```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

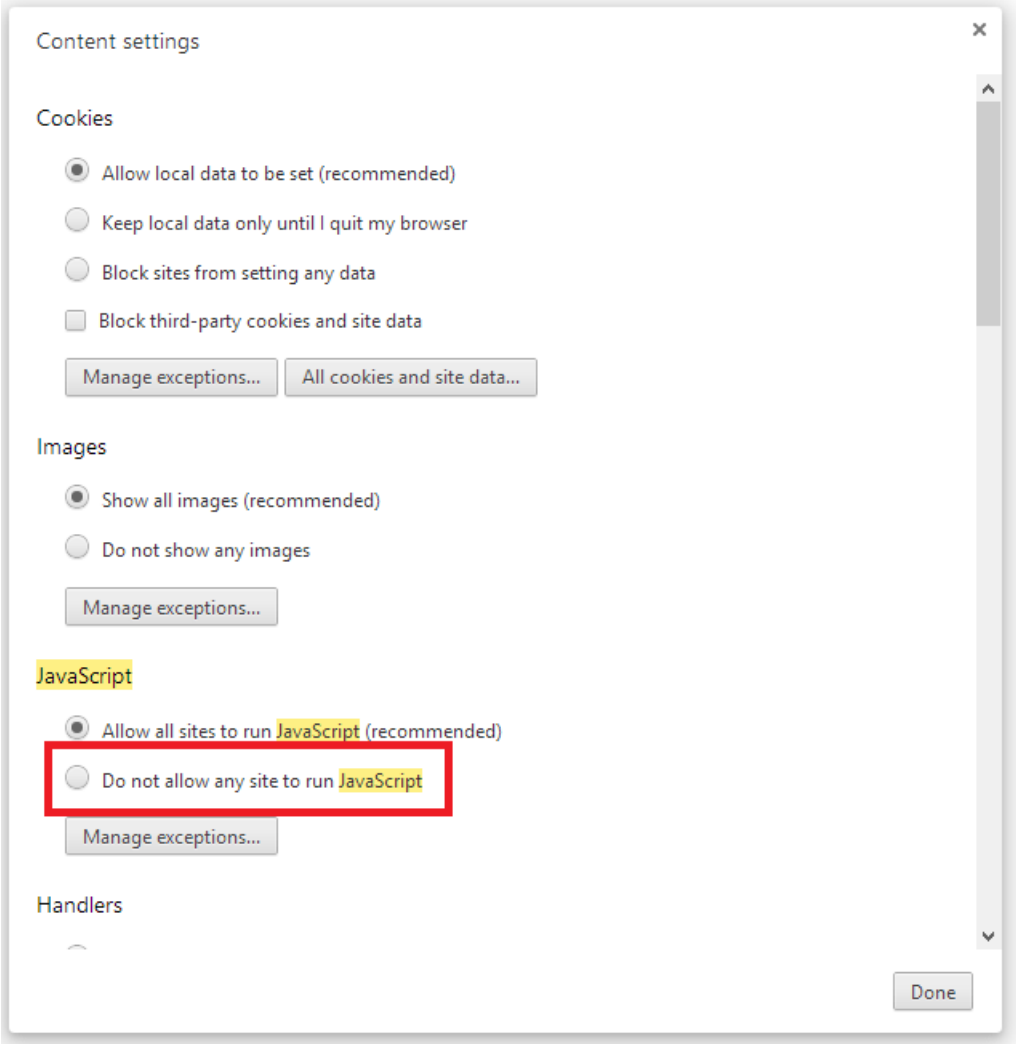
The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation won't submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

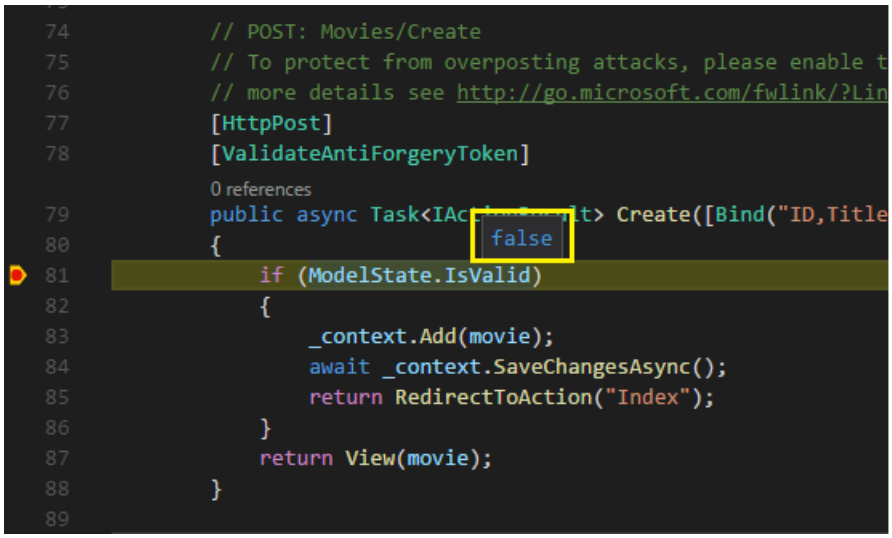
The following image shows how to disable JavaScript in the Firefox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.



The portion of the *Create.cshtml* view template is shown in the following markup:

```
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>

        @*Markup removed for brevity.*@
```

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The Validation Tag Helper displays validation errors. See Validation for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the *Movie.cs* file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data (and supplies elements/attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

> **NOTE**
> jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:
>
> `[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]`

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z]*$"), Required, StringLength(30)]
    public string Genre { get; set; }

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

In the next part of the series, we review the app and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- Working with Forms
- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers

PREVIOUS NEXT

Part 10, examine the Details and Delete methods of an ASP.NET Core app

9/22/2020 • 3 minutes to read • Edit Online

By Rick Anderson

Open the Movie controller and examine the `Details` method:

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method, and an `id` value. Recall these segments are defined in *Startup.cs*.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

EF makes it easy to search for data using the `FirstOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:{PORT}/Movies/Details/1` to something like `http://localhost:{PORT}/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

```
// POST: Movies/Delete/6
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

PREVIOUS

Views in ASP.NET Core MVC

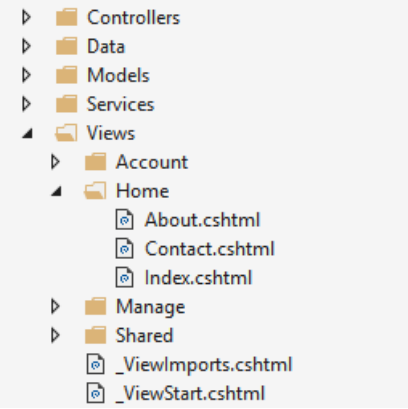
9/22/2020 • 13 minutes to read • Edit Online

By Steve Smith

This document explains views used in ASP.NET Core MVC applications. For information on Razor Pages, see Introduction to Razor Pages.

In the Model-View-Controller (MVC) pattern, the *view* handles the app's data presentation and user interaction. A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.

In ASP.NET Core MVC, views are *.cshtml* files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in a *Views* folder at the root of the app:



The *Home* controller is represented by a *Home* folder inside the *Views* folder. The *Home* folder contains the views for the *About*, *Contact*, and *Index* (homepage) webpages. When a user requests one of these three webpages, controller actions in the *Home* controller determine which of the three views is used to build and return a webpage to the user.

Use layouts to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

Partial views reduce code duplication by managing reusable parts of views. For example, a partial view is useful for an author biography on a blog website that appears in several views. An author biography is ordinary view content and doesn't require code to execute in order to produce the content for the webpage. Author biography content is available to the view by model binding alone, so using a partial view for this type of content is ideal.

View components are similar to partial views in that they allow you to reduce repetitive code, but they're appropriate for view content that requires code to run on the server in order to render the webpage. View components are useful when the rendered content requires database interaction, such as for a website shopping cart. View components aren't limited to model binding in order to produce webpage output.

Benefits of using views

Views help to establish separation of concerns within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes your app modular, which provides several benefits:

- The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
- The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
- It's easier to test the user interface parts of the app because the views are separate units.
- Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

Creating a view

Views that are specific to a controller are created in the *Views/[ControllerName]* folder. Views that are shared among controllers are placed in the *Views/Shared* folder. To create a view, add a new file and give it the same name as its associated controller action with the *.cshtml* file extension. To create a view that corresponds with the *About* action in the *Home* controller, create an *About.cshtml* file in the *Views/Home* folder:

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>Use this area to provide additional information.</p>
```

Razor markup starts with the `@` symbol. Run C# statements by placing C# code within Razor code blocks set off by curly braces (`{ ... }`). For example, see the assignment of "About" to `ViewData["Title"]` shown above. You can display values within HTML by simply referencing the value with the `@` symbol. See the contents of the `<h2>` and `<h3>` elements above.

The view content shown above is only part of the entire webpage that's rendered to the user. The rest of the page's layout and other common aspects of the view are specified in other view files. To learn more, see the Layout topic.

How controllers specify views

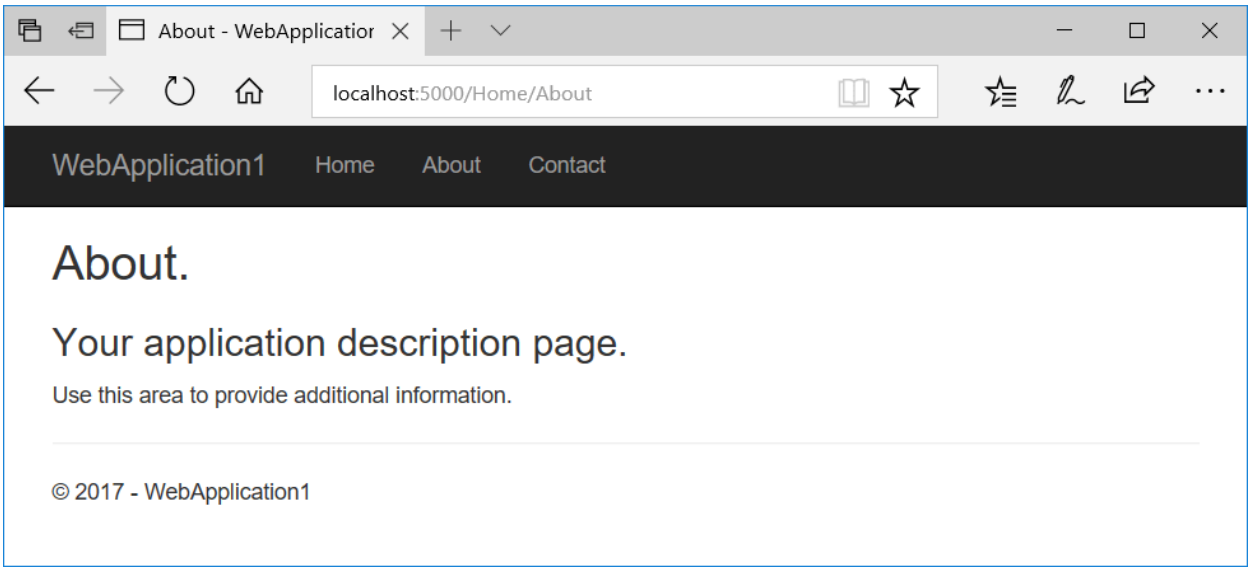
Views are typically returned from actions as a ViewResult, which is a type of ActionResult. Your action method can create and return a `ViewResult` directly, but that isn't commonly done. Since most controllers inherit from Controller, you simply use the `View` helper method to return the `ViewResult`:

HomeController.cs

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

When this action returns, the *About.cshtml* view shown in the last section is rendered as the following webpage:



The `View` helper method has several overloads. You can optionally specify:

- An explicit view to return:

```
return View("Orders");
```

- A model to pass to the view:

```
return View(Orders);
```

- Both a view and a model:

```
return View("Orders", Orders);
```

View discovery

When an action returns a view, a process called *view discovery* takes place. This process determines which view file is used based on the view name.

The default behavior of the `View` method (`return View();`) is to return a view with the same name as the action method from which it's called. For example, the *About* `ActionResult` method name of the controller is used to search for a view file named *About.cshtml*. First, the runtime looks in the *Views/[ControllerName]* folder for the view. If it doesn't find a matching view there, it searches the *Shared* folder for the view.

It doesn't matter if you implicitly return the `ViewResult` with `return View();` or explicitly pass the view name to the `View` method with `return View("<ViewName>");`. In both cases, view discovery searches for a matching view file in this order:

1. *Views/[ControllerName]/[ViewName].cshtml*
2. *Views/Shared/[ViewName].cshtml*

A view file path can be provided instead of a view name. If using an absolute path starting at the app root (optionally starting with "/" or "~/"), the *.cshtml* extension must be specified:

```
return View("Views/Home/About.cshtml");
```

You can also use a relative path to specify views in different directories without the *.cshtml* extension. Inside the `HomeController`, you can return the *Index* view of your *Manage* views with a relative path:

```
return View("../Manage/Index");
```

Similarly, you can indicate the current controller-specific directory with the "./" prefix:

```
return View("./About");
```

Partial views and view components use similar (but not identical) discovery mechanisms.

You can customize the default convention for how views are located within the app by using a custom IViewLocationExpander.

View discovery relies on finding view files by file name. If the underlying file system is case sensitive, view names are probably case sensitive. For compatibility across operating systems, match case between controller and action names and associated view folders and file names. If you encounter an error that a view file can't be found while working with a case-sensitive file system, confirm that the casing matches between the requested view file and the actual view file name.

Follow the best practice of organizing the file structure for your views to reflect the relationships among controllers, actions, and views for maintainability and clarity.

Passing data to views

Pass data to views using several approaches:

- Strongly typed data: viewmodel
- Weakly typed data
  - `ViewData` (`ViewDataAttribute`)
  - `ViewBag`

Strongly typed data (viewmodel)

The most robust approach is to specify a model type in the view. This model is commonly referred to as a *viewmodel*. You pass an instance of the viewmodel type to the view from the action.

Using a viewmodel to pass data to a view allows the view to take advantage of *strong* type checking. *Strong typing* (or *strongly typed*) means that every variable and constant has an explicitly defined type (for example, `string` , `int` , or `DateTime`). The validity of types used in a view is checked at compile time.

Visual Studio and Visual Studio Code list strongly typed class members using a feature called IntelliSense. When you want to see the properties of a viewmodel, type the variable name for the viewmodel followed by a period (`.`). This helps you write code faster with fewer errors.

Specify a model using the `@model` directive. Use the model with `@Model` :

```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

To provide the model to the view, the controller passes it as a parameter:

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

There are no restrictions on the model types that you can provide to a view. We recommend using Plain Old CLR Object (POCO) viewmodels with little or no behavior (methods) defined. Usually, viewmodel classes are either stored in the *Models* folder or a separate *ViewModels* folder at the root of the app. The *Address* viewmodel used in the example above is a POCO viewmodel stored in a file named *Address.cs*:

```
namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}
```

Nothing prevents you from using the same classes for both your viewmodel types and your business model types. However, using separate models allows your views to vary independently from the business logic and data access parts of your app. Separation of models and viewmodels also offers security benefits when models use model binding and validation for data sent to the app by the user.

Weakly typed data (ViewData, ViewData attribute, and ViewBag)

`ViewBag` *isn't available in Razor Pages.*

In addition to strongly typed views, views have access to a *weakly typed* (also called *loosely typed*) collection of data. Unlike strong types, *weak types* (or *loose types*) means that you don't explicitly declare the type of data you're using. You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views.

PASSING DATA BETWEEN A ...	EXAMPLE
Controller and a view	Populating a dropdown list with data.
View and a layout view	Setting the **<title>** element content in the layout view from a view file.
Partial view and a view	A widget that displays data based on the webpage that the user requested.

This collection can be referenced through either the `ViewData` or `ViewBag` properties on controllers and views. The `ViewData` property is a dictionary of weakly typed objects. The `ViewBag` property is a wrapper around

`ViewData` that provides dynamic properties for the underlying `ViewData` collection. Note: Key lookups are case-insensitive for both `ViewData` and `ViewBag`.

`ViewData` and `ViewBag` are dynamically resolved at runtime. Since they don't offer compile-time type checking, both are generally more error-prone than using a viewmodel. For that reason, some developers prefer to minimally or never use `ViewData` and `ViewBag`.

ViewData

`ViewData` is a ViewDataDictionary object accessed through `string` keys. String data can be stored and used directly without the need for a cast, but you must cast other `ViewData` object values to specific types when you extract them. You can use `ViewData` to pass data from controllers to views and within views, including partial views and layouts.

The following is an example that sets values for a greeting and an address using `ViewData` in an action:

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"]  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

Work with the data in a view:

```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```

ViewData attribute

Another approach that uses the ViewDataDictionary is ViewDataAttribute. Properties on controllers or Razor Page models marked with the `[ViewData]` attribute have their values stored and loaded from the dictionary.

In the following example, the Home controller contains a `Title` property marked with `[ViewData]`. The `About` method sets the title for the About view:

```
public class HomeController : Controller
{
    [ViewData]
    public string Title { get; set; }

    public IActionResult About()
    {
        Title = "About Us";
        ViewData["Message"] = "Your application description page.";

        return View();
    }
}
```

In the layout, the title is read from the ViewData dictionary:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

ViewBag

`ViewBag` *isn't available in Razor Pages.*

`ViewBag` is a DynamicViewData object that provides dynamic access to the objects stored in `ViewData`. `ViewBag` can be more convenient to work with, since it doesn't require casting. The following example shows how to use `ViewBag` with the same result as using `ViewData` above:

```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>
```

Using ViewData and ViewBag simultaneously

`ViewBag` *isn't available in Razor Pages.*

Since `ViewData` and `ViewBag` refer to the same underlying `ViewData` collection, you can use both `ViewData` and `ViewBag` and mix and match between them when reading and writing values.

Set the title using `ViewBag` and the description using `ViewData` at the top of an *About.cshtml* view:

```
@{
    Layout = "/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Contoso";
    ViewData["Description"] = "Let us tell you about Contoso's philosophy and mission.";
}
```

Read the properties but reverse the use of `ViewData` and `ViewBag`. In the *_Layout.cshtml* file, obtain the title using `ViewData` and obtain the description using `ViewBag`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"]</title>
    <meta name="description" content="@ViewBag.Description">
    ...
```

Remember that strings don't require a cast for `ViewData`. You can use `@ViewData["Title"]` without casting.

Using both `ViewData` and `ViewBag` at the same time works, as does mixing and matching reading and writing the properties. The following markup is rendered:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>About Contoso</title>
    <meta name="description" content="Let us tell you about Contoso's philosophy and mission.">
    ...
```

Summary of the differences between ViewData and ViewBag

`ViewBag` isn't available in the Razor Pages.

- `ViewData`
  - Derives from ViewDataDictionary, so it has dictionary properties that can be useful, such as `ContainsKey`, `Add`, `Remove`, and `Clear`.
  - Keys in the dictionary are strings, so whitespace is allowed. Example: `ViewData["Some Key With Whitespace"]`
  - Any type other than a `string` must be cast in the view to use `ViewData`.
- `ViewBag`
  - Derives from DynamicViewData, so it allows the creation of dynamic properties using dot notation (`@ViewBag.SomeKey = <value or object>`), and no casting is required. The syntax of `ViewBag` makes it quicker to add to controllers and views.
  - Simpler to check for null values. Example: `@ViewBag.Person?.Name`

When to use ViewData or ViewBag

Both `ViewData` and `ViewBag` are equally valid approaches for passing small amounts of data among controllers and views. The choice of which one to use is based on preference. You can mix and match `ViewData` and `ViewBag` objects, however, the code is easier to read and maintain with one approach used consistently. Both approaches are dynamically resolved at runtime and thus prone to causing runtime errors. Some development teams avoid them.

Dynamic views

Views that don't declare a model type using `@model` but that have a model instance passed to them (for example, `return View(Address);`) can reference the instance's properties dynamically:

```
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

This feature offers flexibility but doesn't offer compilation protection or IntelliSense. If the property doesn't exist, webpage generation fails at runtime.

More view features

Tag Helpers make it easy to add server-side behavior to existing HTML tags. Using Tag Helpers avoids the need to write custom code or helpers within your views. Tag helpers are applied as attributes to HTML elements and are ignored by editors that can't process them. This allows you to edit and render view markup in a variety of tools.

Generating custom HTML markup can be achieved with many built-in HTML Helpers. More complex user interface logic can be handled by View Components. View components provide the same SoC that controllers and views offer. They can eliminate the need for actions and views that deal with data used by common user interface elements.

Like many other aspects of ASP.NET Core, views support dependency injection, allowing services to be injected into views.

Partial views in ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

By Steve Smith, Maher JENDOUBI, Rick Anderson, and Scott Sauber

A partial view is a Razor markup file (*.cshtml*) that renders HTML output *within* another markup file's rendered output.

The term *partial view* is used when developing either an MVC app, where markup files are called *views*, or a Razor Pages app, where markup files are called *pages*. This topic generically refers to MVC views and Razor Pages pages as *markup files*.

View or download sample code (how to download)

When to use partial views

Partial views are an effective way to:

- Break up large markup files into smaller components.

  In a large, complex markup file composed of several logical pieces, there's an advantage to working with each piece isolated into a partial view. The code in the markup file is manageable because the markup only contains the overall page structure and references to partial views.

- Reduce the duplication of common markup content across markup files.

  When the same markup elements are used across markup files, a partial view removes the duplication of markup content into one partial view file. When the markup is changed in the partial view, it updates the rendered output of the markup files that use the partial view.

Partial views shouldn't be used to maintain common layout elements. Common layout elements should be specified in _Layout.cshtml files.

Don't use a partial view where complex rendering logic or code execution is required to render the markup. Instead of a partial view, use a view component.

Declare partial views

A partial view is a *.cshtml* markup file maintained within the *Views* folder (MVC) or *Pages* folder (Razor Pages).

In ASP.NET Core MVC, a controller's ViewResult is capable of returning either a view or a partial view. In Razor Pages, a PageModel can return a partial view represented as a PartialViewResult object. Referencing and rendering partial views is described in the Reference a partial view section.

Unlike MVC view or page rendering, a partial view doesn't run *_ViewStart.cshtml*. For more information on *_ViewStart.cshtml*, see Layout in ASP.NET Core.

Partial view file names often begin with an underscore (`_`). This naming convention isn't required, but it helps to visually differentiate partial views from views and pages.

A partial view is a *.cshtml* markup file maintained within the *Views* folder.

A controller's ViewResult is capable of returning either a view or a partial view. Referencing and rendering partial views is described in the Reference a partial view section.

Unlike MVC view rendering, a partial view doesn't run *_ViewStart.cshtml*. For more information on *_ViewStart.cshtml*, see Layout in ASP.NET Core.

Partial view file names often begin with an underscore (`_`). This naming convention isn't required, but it helps to visually differentiate partial views from views.

Reference a partial view

Use a partial view in a Razor Pages PageModel

In ASP.NET Core 2.0 or 2.1, the following handler method renders the *_AuthorPartialRP.cshtml* partial view to the response:

```
public IActionResult OnGetPartial() =>
    new PartialViewResult
    {
        ViewName = "_AuthorPartialRP",
        ViewData = ViewData,
    };
```

In ASP.NET Core 2.2 or later, a handler method can alternatively call the Partial method to produce a `PartialViewResult` object:

```
public IActionResult OnGetPartial() =>
    Partial("_AuthorPartialRP");
```

Use a partial view in a markup file

Within a markup file, there are several ways to reference a partial view. We recommend that apps use one of the following asynchronous rendering approaches:

- Partial Tag Helper
- Asynchronous HTML Helper

Within a markup file, there are two ways to reference a partial view:

- Asynchronous HTML Helper
- Synchronous HTML Helper

We recommend that apps use the Asynchronous HTML Helper.

Partial Tag Helper

The Partial Tag Helper requires ASP.NET Core 2.1 or later.

The Partial Tag Helper renders content asynchronously and uses an HTML-like syntax:

```
<partial name="_PartialName" />
```

When a file extension is present, the Tag Helper references a partial view that must be in the same folder as the markup file calling the partial view:

```
<partial name="_PartialName.cshtml" />
```

The following example references a partial view from the app root. Paths that start with a tilde-slash (`~/`) or a slash (`/`) refer to the app root:

Razor Pages

```
<partial name="~/Pages/Folder/_PartialName.cshtml" />
<partial name="/Pages/Folder/_PartialName.cshtml" />
```

MVC

```
<partial name="~/Views/Folder/_PartialName.cshtml" />
<partial name="/Views/Folder/_PartialName.cshtml" />
```

The following example references a partial view with a relative path:

```
<partial name="../Account/_PartialName.cshtml" />
```

For more information, see Partial Tag Helper in ASP.NET Core.

Asynchronous HTML Helper

When using an HTML Helper, the best practice is to use PartialAsync. `PartialAsync` returns an IHtmlContent type wrapped in a Task<TResult>. The method is referenced by prefixing the awaited call with an `@` character:

```
@await Html.PartialAsync("_PartialName")
```

When the file extension is present, the HTML Helper references a partial view that must be in the same folder as the markup file calling the partial view:

```
@await Html.PartialAsync("_PartialName.cshtml")
```

The following example references a partial view from the app root. Paths that start with a tilde-slash (`~/`) or a slash (`/`) refer to the app root:

Razor Pages

```
@await Html.PartialAsync("~/Pages/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Pages/Folder/_PartialName.cshtml")
```

MVC

```
@await Html.PartialAsync("~/Views/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Views/Folder/_PartialName.cshtml")
```

The following example references a partial view with a relative path:

```
@await Html.PartialAsync("../Account/_LoginPartial.cshtml")
```

Alternatively, you can render a partial view with RenderPartialAsync. This method doesn't return an IHtmlContent. It streams the rendered output directly to the response. Because the method doesn't return a result, it must be called within a Razor code block:

```
@{
    await Html.RenderPartialAsync("_AuthorPartial");
}
```

Since `RenderPartialAsync` streams rendered content, it provides better performance in some scenarios. In performance-critical situations, benchmark the page using both approaches and use the approach that generates a faster response.

Synchronous HTML Helper

Partial and RenderPartial are the synchronous equivalents of `PartialAsync` and `RenderPartialAsync`, respectively. The synchronous equivalents aren't recommended because there are scenarios in which they deadlock. The synchronous methods are targeted for removal in a future release.

> **IMPORTANT**
> If you need to execute code, use a view component instead of a partial view.

Calling `Partial` or `RenderPartial` results in a Visual Studio analyzer warning. For example, the presence of `Partial` yields the following warning message:

> Use of IHtmlHelper.Partial may result in application deadlocks. Consider using <partial> Tag Helper or IHtmlHelper.PartialAsync.

Replace calls to `@Html.Partial` with `@await Html.PartialAsync` or the Partial Tag Helper. For more information on Partial Tag Helper migration, see Migrate from an HTML Helper.

Partial view discovery

When a partial view is referenced by name without a file extension, the following locations are searched in the stated order:

Razor Pages

1. Currently executing page's folder
2. Directory graph above the page's folder
3. `/Shared`
4. `/Pages/Shared`
5. `/Views/Shared`

MVC

1. `/Areas/<Area-Name>/Views/<Controller-Name>`
2. `/Areas/<Area-Name>/Views/Shared`
3. `/Views/Shared`
4. `/Pages/Shared`

1. `/Areas/<Area-Name>/Views/<Controller-Name>`
2. `/Areas/<Area-Name>/Views/Shared`
3. `/Views/Shared`

The following conventions apply to partial view discovery:

- Different partial views with the same file name are allowed when the partial views are in different folders.

- When referencing a partial view by name without a file extension and the partial view is present in both the caller's folder and the *Shared* folder, the partial view in the caller's folder supplies the partial view. If the partial view isn't present in the caller's folder, the partial view is provided from the *Shared* folder. Partial views in the *Shared* folder are called *shared partial views* or *default partial views*.
- Partial views can be *chained*—a partial view can call another partial view if a circular reference isn't formed by the calls. Relative paths are always relative to the current file, not to the root or parent of the file.

> **NOTE**
>
> A Razor `section` defined in a partial view is invisible to parent markup files. The `section` is only visible to the partial view in which it's defined.

Access data from partial views

When a partial view is instantiated, it receives a *copy* of the parent's `ViewData` dictionary. Updates made to the data within the partial view aren't persisted to the parent view. `ViewData` changes in a partial view are lost when the partial view returns.

The following example demonstrates how to pass an instance of ViewDataDictionary to a partial view:

```
@await Html.PartialAsync("_PartialName", customViewData)
```

You can pass a model into a partial view. The model can be a custom object. You can pass a model with `PartialAsync` (renders a block of content to the caller) or `RenderPartialAsync` (streams the content to the output):

```
@await Html.PartialAsync("_PartialName", model)
```

Razor Pages

The following markup in the sample app is from the *Pages/ArticlesRP/ReadRP.cshtml* page. The page contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. The `ViewDataDictionary` constructor overload is used to pass a new `ViewData` dictionary while retaining the existing `ViewData` dictionary.

```
@model ReadRPModel

<h2>@Model.Article.Title</h2>
@* Pass the author's name to Pages\Shared\_AuthorPartialRP.cshtml *@
@await Html.PartialAsync("../Shared/_AuthorPartialRP", Model.Article.AuthorName)
@Model.Article.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Pages\ArticlesRP\_ArticleSectionRP.cshtml partial view. *@
@{
    var index = 0;

    foreach (var section in Model.Article.Sections)
    {
        await Html.PartialAsync("_ArticleSectionRP",
                                section,
                                new ViewDataDictionary(ViewData)
                                {
                                    { "index", index }
                                });

        index++;
    }
}
```

Pages/Shared/_AuthorPartialRP.cshtml is the first partial view referenced by the *ReadRP.cshtml* markup file:

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Pages/Shared/_AuthorPartialRP.cshtml.
</div>
```

Pages/ArticlesRP/_ArticleSectionRP.cshtml is the second partial view referenced by the *ReadRP.cshtml* markup file:

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

MVC

The following markup in the sample app shows the *Views/Articles/Read.cshtml* view. The view contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. The `ViewDataDictionary` constructor overload is used to pass a new `ViewData` dictionary while retaining the existing `ViewData` dictionary.

```
@model PartialViewsSample.ViewModels.Article

<h2>@Model.Title</h2>
@* Pass the author's name to Views\Shared\_AuthorPartial.cshtml *@
@await Html.PartialAsync("_AuthorPartial", Model.AuthorName)
@Model.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Views\Articles\_ArticleSection.cshtml partial view. *@
@{
    var index = 0;

    foreach (var section in Model.Sections)
    {
        await Html.PartialAsync("_ArticleSection",
                                section,
                                new ViewDataDictionary(ViewData)
                                {
                                    { "index", index }
                                });

        index++;
    }
}
```

Views/Shared/_AuthorPartial.cshtml is the first partial view referenced by the *Read.cshtml* markup file:

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Views/Shared/_AuthorPartial.cshtml.
</div>
```

Views/Articles/_ArticleSection.cshtml is the second partial view referenced by the *Read.cshtml* markup file:

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

At runtime, the partials are rendered into the parent markup file's rendered output, which itself is rendered within the shared *_Layout.cshtml*. The first partial view renders the article author's name and publication date:

> Abraham Lincoln
>
> This partial view from <shared partial view file path>. 11/19/1863 12:00:00 AM

The second partial view renders the article's sections:

> Section One Index: 0
>
> Four score and seven years ago ...
>
> Section Two Index: 1
>
> Now we are engaged in a great civil war, testing ...
>
> Section Three Index: 2

> But, in a larger sense, we can not dedicate ...

Additional resources

- razor syntax reference for ASP.NET Core
- Tag Helpers in ASP.NET Core
- Partial Tag Helper in ASP.NET Core
- View components in ASP.NET Core
- Areas in ASP.NET Core

- razor syntax reference for ASP.NET Core
- View components in ASP.NET Core
- Areas in ASP.NET Core

Handle requests with controllers in ASP.NET Core MVC

9/22/2020 • 4 minutes to read • Edit Online

By Steve Smith and Scott Addie

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET Core MVC.

What is a Controller?

A controller is used to define and group a set of actions. An action (or *action method*) is a method on a controller which handles requests. Controllers logically group similar actions together. This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively. Requests are mapped to actions through routing.

By convention, controller classes:

- Reside in the project's root-level *Controllers* folder.
- Inherit from `Microsoft.AspNetCore.Mvc.Controller`.

A controller is an instantiable class in which at least one of the following conditions is true:

- The class name is suffixed with `Controller`.
- The class inherits from a class whose name is suffixed with `Controller`.
- The `[Controller]` attribute is applied to the class.

A controller class must not have an associated `[NonController]` attribute.

Controllers should follow the Explicit Dependencies Principle. There are a couple of approaches to implementing this principle. If multiple controller actions require the same service, consider using constructor injection to request those dependencies. If the service is needed by only a single action method, consider using Action Injection to request the dependency.

Within the **M**odel-**V**iew-**C**ontroller pattern, a controller is responsible for the initial processing of the request and instantiation of the model. Generally, business decisions should be performed within the model.

The controller takes the result of the model's processing (if any) and returns either the proper view and its associated view data or the result of the API call. Learn more at Overview of ASP.NET Core MVC and Get started with ASP.NET Core MVC and Visual Studio.

The controller is a *UI-level* abstraction. Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned. In well-factored apps, it doesn't directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.

Defining Actions

Public methods on a controller, except those with the `[NonAction]` attribute, are actions. Parameters on actions are bound to request data and are validated using model binding. Model validation occurs for everything that's model-bound. The `ModelState.IsValid` property value indicates whether model binding and validation succeeded.

Action methods should contain logic for mapping a request to a business concern. Business concerns should typically be represented as services that the controller accesses through dependency injection. Actions then map the result of the business action to an application state.

Actions can return anything, but frequently return an instance of `IActionResult` (or `Task<IActionResult>` for async methods) that produces a response. The action method is responsible for choosing *what kind of response*. The action result *does the responding*.

Controller Helper Methods

Controllers usually inherit from Controller, although this isn't required. Deriving from `Controller` provides access to three categories of helper methods:

1. Methods resulting in an empty response body

No `Content-Type` HTTP response header is included, since the response body lacks content to describe.

There are two result types within this category: Redirect and HTTP Status Code.

- HTTP Status Code

  This type returns an HTTP status code. A couple of helper methods of this type are `BadRequest`, `NotFound`, and `Ok`. For example, `return BadRequest();` produces a 400 status code when executed. When methods such as `BadRequest`, `NotFound`, and `Ok` are overloaded, they no longer qualify as HTTP Status Code responders, since content negotiation is taking place.

- Redirect

  This type returns a redirect to an action or destination (using `Redirect`, `LocalRedirect`, `RedirectToAction`, or `RedirectToRoute`). For example, `return RedirectToAction("Complete", new {id = 123});` redirects to `Complete`, passing an anonymous object.

  The Redirect result type differs from the HTTP Status Code type primarily in the addition of a `Location` HTTP response header.

2. Methods resulting in a non-empty response body with a predefined content type

Most helper methods in this category include a `ContentType` property, allowing you to set the `Content-Type` response header to describe the response body.

There are two result types within this category: View and Formatted Response.

- View

  This type returns a view which uses a model to render HTML. For example, `return View(customer);` passes a model to the view for data-binding.

- Formatted Response

  This type returns JSON or a similar data exchange format to represent an object in a specific manner. For example, `return Json(customer);` serializes the provided object into JSON format.

  Other common methods of this type include `File` and `PhysicalFile`. For example, `return PhysicalFile(customerFilePath, "text/xml");` returns PhysicalFileResult.

3. Methods resulting in a non-empty response body formatted in a content type negotiated with the client

This category is better known as **Content Negotiation**. Content negotiation applies whenever an action returns an ObjectResult type or something other than an IActionResult implementation. An action that returns a non-`IActionResult` implementation (for example, `object`) also returns a Formatted Response.

Some helper methods of this type include `BadRequest`, `CreatedAtRoute`, and `Ok`. Examples of these methods include `return BadRequest(modelState);`, `return CreatedAtRoute("routename", values, newobject);`, and

`return Ok(value);`, respectively. Note that `BadRequest` and `Ok` perform content negotiation only when passed a value; without being passed a value, they instead serve as HTTP Status Code result types. The `CreatedAtRoute` method, on the other hand, always performs content negotiation since its overloads all require that a value be passed.

Cross-Cutting Concerns

Applications typically share parts of their workflow. Examples include an app that requires authentication to access the shopping cart, or an app that caches data on some pages. To perform logic before or after an action method, use a *filter*. Using Filters on cross-cutting concerns can reduce duplication.

Most filter attributes, such as `[Authorize]`, can be applied at the controller or action level depending upon the desired level of granularity.

Error handling and response caching are often cross-cutting concerns:

- Handle errors
- Response Caching

Many cross-cutting concerns can be handled using filters or custom middleware.

Routing to controller actions in ASP.NET Core

9/22/2020 • 68 minutes to read • Edit Online

By Ryan Nowak, Kirk Larkin, and Rick Anderson

ASP.NET Core controllers use the Routing middleware to match the URLs of incoming requests and map them to actions. Routes templates:

- Are defined in startup code or attributes.
- Describe how URL paths are matched to actions.
- Are used to generate URLs for links. The generated links are typically returned in responses.

Actions are either conventionally-routed or attribute-routed. Placing a route on the controller or action makes it attribute-routed. See Mixed routing for more information.

This document:

- Explains the interactions between MVC and routing:
  - How typical MVC apps make use of routing features.
  - Covers both:
    - Conventionally routing typically used with controllers and views.
    - *Attribute routing* used with REST APIs. If you're primarily interested in routing for REST APIs, jump to the Attribute routing for REST APIs section.
  - See Routing for advanced routing details.
- Refers to the default routing system added in ASP.NET Core 3.0, called endpoint routing. It's possible to use controllers with the previous version of routing for compatibility purposes. See the 2.2-3.0 migration guide for instructions. Refer to the 2.2 version of this document for reference material on the legacy routing system.

Set up conventional route

`Startup.Configure` typically has code similar to the following when using conventional routing:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Inside the call to UseEndpoints, MapControllerRoute is used to create a single route. The single route is named `default` route. Most apps with controllers and views use a route template similar to the `default` route. REST APIs should use attribute routing.

The route template `"{controller=Home}/{action=Index}/{id?}"`:

- Matches a URL path like `/Products/Details/5`
- Extracts the route values `{ controller = Products, action = Details, id = 5 }` by tokenizing the path. The extraction of route values results in a match if the app has a controller named `ProductsController` and a `Details` action:

```
public class ProductsController : Controller
{
    public IActionResult Details(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

- `/Products/Details/5` model binds the value of `id = 5` to set the `id` parameter to `5`. See Model Binding for more details.
- `{controller=Home}` defines `Home` as the default `controller`.
- `{action=Index}` defines `Index` as the default `action`.
- The `?` character in `{id?}` defines `id` as optional.
- Default and optional route parameters don't need to be present in the URL path for a match. See Route Template Reference for a detailed description of route template syntax.
- Matches the URL path `/`.
- Produces the route values `{ controller = Home, action = Index }`.

The values for `controller` and `action` make use of the default values. `id` doesn't produce a value since there's no corresponding segment in the URL path. `/` only matches if there exists a `HomeController` and `Index` action:

```
public class HomeController : Controller
{
  public IActionResult Index() { ... }
}
```

Using the preceding controller definition and route template, the `HomeController.Index` action is run for the following URL paths:

- `/Home/Index/17`
- `/Home/Index`
- `/Home`
- `/`

The URL path `/` uses the route template default `Home` controllers and `Index` action. The URL path `/Home` uses the route template default `Index` action.

The convenience method MapDefaultControllerRoute:

```
endpoints.MapDefaultControllerRoute();
```

Replaces:

```
endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

IMPORTANT

Routing is configured using the UseRouting and UseEndpoints middleware. To use controllers:

- Call MapControllers inside `UseEndpoints` to map attribute routed controllers.
- Call MapControllerRoute or MapAreaControllerRoute, to map both conventionally routed controllers and attribute routed controllers.

Conventional routing

Conventional routing is used with controllers and views. The `default` route:

```
endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

is an example of a *conventional routing*. It's called *conventional routing* because it establishes a *convention* for URL paths:

- The first path segment, `{controller=Home}`, maps to the controller name.
- The second segment, `{action=Index}`, maps to the action name.
- The third segment, `{id?}` is used for an optional `id`. The `?` in `{id?}` makes it optional. `id` is used to map to a model entity.

Using this `default` route, the URL path:

- `/Products/List` maps to the `ProductsController.List` action.
- `/Blog/Article/17` maps to `BlogController.Article` and typically model binds the `id` parameter to 17.

This mapping:

- Is based on the controller and action names **only**.
- Isn't based on namespaces, source file locations, or method parameters.

Using conventional routing with the default route allows creating the app without having to come up with a new URL pattern for each action. For an app with CRUD style actions, having consistency for the URLs across controllers:

- Helps simplify the code.
- Makes the UI more predictable.

WARNING

The `id` in the preceding code is defined as optional by the route template. Actions can execute without the optional ID provided as part of the URL. Generally, when `id` is omitted from the URL:

- `id` is set to `0` by model binding.
- No entity is found in the database matching `id == 0`.

Attribute routing provides fine-grained control to make the ID required for some actions and not for others. By convention, the documentation includes optional parameters like `id` when they're likely to appear in correct usage.

Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful. The default conventional route `{controller=Home}/{action=Index}/{id?}`:

- Supports a basic and descriptive routing scheme.
- Is a useful starting point for UI-based apps.
- Is the only route template needed for many web UI apps. For larger web UI apps, another route using Areas is frequently all that's needed.

MapControllerRoute and MapAreaRoute :

- Automatically assign an **order** value to their endpoints based on the order they are invoked.

Endpoint routing in ASP.NET Core 3.0 and later:

- Doesn't have a concept of routes.
- Doesn't provide ordering guarantees for the execution of extensibility, all endpoints are processed at once.

Enable Logging to see how the built-in routing implementations, such as Route, match requests.

Attribute routing is explained later in this document.

Multiple conventional routes

Multiple conventional routes can be added inside `UseEndpoints` by adding more calls to MapControllerRoute and MapAreaControllerRoute. Doing so allows defining multiple conventions, or to adding conventional routes that are dedicated to a specific action, such as:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
    endpoints.MapControllerRoute(name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

The `blog` route in the preceding code is a **dedicated conventional route**. It's called a dedicated conventional route because:

- It uses conventional routing.
- It's dedicated to a specific action.

Because `controller` and `action` don't appear in the route template `"blog/{*article}"` as parameters:

- They can only have the default values `{ controller = "Blog", action = "Article" }`.
- This route always maps to the action `BlogController.Article`.

`/Blog`, `/Blog/Article`, and `/Blog/{any-string}` are the only URL paths that match the blog route.

The preceding example:

- `blog` route has a higher priority for matches than the `default` route because it is added first.
- Is an example of Slug style routing where it's typical to have an article name as part of the URL.

WARNING

In ASP.NET Core 3.0 and later, routing doesn't:

- Define a concept called a *route*. `UseRouting` adds route matching to the middleware pipeline. The `UseRouting` middleware looks at the set of endpoints defined in the app, and selects the best endpoint match based on the request.
- Provide guarantees about the execution order of extensibility like IRouteConstraint or IActionConstraint.

See Routing for reference material on routing.

Conventional routing order

Conventional routing only matches a combination of action and controller that are defined by the app. This is intended to simplify cases where conventional routes overlap. Adding routes using MapControllerRoute, MapDefaultControllerRoute, and MapAreaControllerRoute automatically assign an order value to their endpoints based on the order they are invoked. Matches from a route that appears earlier have a higher priority. Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area. Dedicated conventional routes with catch-all route parameters like `{*article}` can make a route too greedy, meaning that it matches URLs that you intended to be matched by other routes. Put the greedy routes later in the route table to prevent greedy matches.

WARNING

A **catch-all** parameter may match routes incorrectly due to a bug in routing. Apps impacted by this bug have the following characteristics:

- A catch-all route, for example, `{**slug}"`
- The catch-all route fails to match requests it should match.
- Removing other routes makes catch-all route start working.

See GitHub bugs 18677 and 16579 for example cases that hit this bug.

An opt-in fix for this bug is contained in .NET Core 3.1.301 SDK and later. The following code sets an internal switch that fixes this bug:

```
public static void Main(string[] args)
{
   AppContext.SetSwitch("Microsoft.AspNetCore.Routing.UseCorrectCatchAllBehavior",
                        true);
   CreateHostBuilder(args).Build().Run();
}
// Remaining code removed for brevity.
```

Resolving ambiguous actions

When two endpoints match through routing, routing must do one of the following:

- Choose the best candidate.
- Throw an exception.

For example:

```
    public class Products33Controller : Controller
    {
        public IActionResult Edit(int id)
        {
            return ControllerContext.MyDisplayRouteInfo(id);
        }

        [HttpPost]
        public IActionResult Edit(int id, Product product)
        {
            return ControllerContext.MyDisplayRouteInfo(id, product.name);
        }
    }
}
```

The preceding controller defines two actions that match:

- The URL path `/Products33/Edit/17`
- Route data `{ controller = Products33, action = Edit, id = 17 }`.

This is a typical pattern for MVC controllers:

- `Edit(int)` displays a form to edit a product.
- `Edit(int, Product)` processes the posted form.

To resolve the correct route:

- `Edit(int, Product)` is selected when the request is an HTTP `POST`.
- `Edit(int)` is selected when the HTTP verb is anything else. `Edit(int)` is generally called via `GET`.

The HttpPostAttribute, `[HttpPost]`, is provided to routing so that it can choose based on the HTTP method of the request. The `HttpPostAttribute` makes `Edit(int, Product)` a better match than `Edit(int)`.

It's important to understand the role of attributes like `HttpPostAttribute`. Similar attributes are defined for other HTTP verbs. In conventional routing, it's common for actions to use the same action name when they're part of a show form, submit form workflow. For example, see Examine the two Edit action methods.

If routing can't choose a best candidate, an AmbiguousMatchException is thrown, listing the multiple matched endpoints.

Conventional route names

The strings `"blog"` and `"default"` in the following examples are conventional route names:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
    endpoints.MapControllerRoute(name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

The route names give the route a logical name. The named route can be used for URL generation. Using a named route simplifies URL creation when the ordering of routes could make URL generation complicated. Route names must be unique application wide.

Route names:

- Have no impact on URL matching or handling of requests.
- Are used only for URL generation.

The route name concept is represented in routing as IEndpointNameMetadata. The terms **route name** and **endpoint name**:

- Are interchangeable.
- Which one is used in documentation and code depends on the API being described.

Attribute routing for REST APIs

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs.

Attribute routing uses a set of attributes to map actions directly to route templates. The following `StartUp.Configure` code is typical for a REST API and is used in the next sample:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

In the preceding code, MapControllers is called inside `UseEndpoints` to map attribute routed controllers.

In the following example:

- The preceding `Configure` method is used.
- `HomeController` matches a set of URLs similar to what the default conventional route `{controller=Home}/{action=Index}/{id?}` matches.

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The `HomeController.Index` action is run for any of the URL paths `/`, `/Home`, `/Home/Index`, or `/Home/Index/3`.

This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route. The conventional default route handles routes more succinctly. However, attribute routing allows and requires precise control of which route templates apply to each action.

With attribute routing, the controller and action names play no part in which action is matched, unless token replacement is used. The following example matches the same URLs as the previous example:

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The following code uses token replacement for `action` and `controller`:

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("[controller]/[action]")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Route("[controller]/[action]")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The following code applies `[Route("[controller]/[action]")]` to the controller:

```
[Route("[controller]/[action]")]
public class HomeController : Controller
{
    [Route("~/")]
    [Route("/Home")]
    [Route("~/Home/Index")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

In the preceding code, the `Index` method templates must prepend `/` or `~/` to the route templates. Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller.

See Route template precedence for information on route template selection.

Reserved routing names

The following keywords are reserved route parameter names when using Controllers or Razor Pages:

- `action`
- `area`
- `controller`
- `handler`
- `page`

Using `page` as a route parameter with attribute routing is a common error. Doing that results in inconsistent and confusing behavior with URL generation.

```
public class MyDemo2Controller : Controller
{
    [Route("/articles/{page}")]
    public IActionResult ListArticles(int page)
    {
        return ControllerContext.MyDisplayRouteInfo(page);
    }
}
```

The special parameter names are used by the URL generation to determine if a URL generation operation refers to a Razor Page or to a Controller.

HTTP verb templates

ASP.NET Core has the following HTTP verb templates:

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]
- [HttpHead]
- [HttpPatch]

Route templates

ASP.NET Core has the following route templates:

- All the HTTP verb templates are route templates.
- [Route]

Attribute routing with Http verb attributes

Consider the following controller:

```
[Route("api/[controller]")]
[ApiController]
public class Test2Controller : ControllerBase
{
    [HttpGet]   // GET /api/test2
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]   // GET /api/test2/xyz
    public IActionResult GetProduct(string id)
    {
       return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int/{id:int}")] // GET /api/test2/int/3
    public IActionResult GetIntProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int2/{id}")]  // GET /api/test2/int2/3
    public IActionResult GetInt2Product(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

- Each action contains the `[HttpGet]` attribute, which constrains matching to HTTP GET requests only.
- The `GetProduct` action includes the `"{id}"` template, therefore `id` is appended to the `"api/[controller]"` template on the controller. The methods template is `"api/[controller]/"{id}""`. Therefore this action only matches GET requests of for the form `/api/test2/xyz`, `/api/test2/123`, `/api/test2/{any string}`, etc.

  ```
  [HttpGet("{id}")]   // GET /api/test2/xyz
  public IActionResult GetProduct(string id)
  {
     return ControllerContext.MyDisplayRouteInfo(id);
  }
  ```

- The `GetIntProduct` action contains the `"int/{id:int}")` template. The `:int` portion of the template constrains the `id` route values to strings that can be converted to an integer. A GET request to `/api/test2/int/abc`:
  - Doesn't match this action.
  - Returns a 404 Not Found error.

  ```
  [HttpGet("int/{id:int}")] // GET /api/test2/int/3
  public IActionResult GetIntProduct(int id)
  {
      return ControllerContext.MyDisplayRouteInfo(id);
  }
  ```

- The `GetInt2Product` action contains `{id}` in the template, but doesn't constrain `id` to values that can be converted to an integer. A GET request to `/api/test2/int2/abc`:
  - Matches this route.

- Model binding fails to convert `abc` to an integer. The `id` parameter of the method is integer.
- Returns a 400 Bad Request because model binding failed to convert `abc` to an integer.

```
[HttpGet("int2/{id}")]  // GET /api/test2/int2/3
public IActionResult GetInt2Product(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

Attribute routing can use HttpMethodAttribute attributes such as HttpPostAttribute, HttpPutAttribute, and HttpDeleteAttribute. All of the HTTP verb attributes accept a route template. The following example shows two actions that match the same route template:

```
[ApiController]
public class MyProductsController : ControllerBase
{
    [HttpGet("/products3")]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpPost("/products3")]
    public IActionResult CreateProduct(MyProduct myProduct)
    {
        return ControllerContext.MyDisplayRouteInfo(myProduct.Name);
    }
}
```

Using the URL path `/products3`:

- The `MyProductsController.ListProducts` action runs when the HTTP verb is `GET`.
- The `MyProductsController.CreateProduct` action runs when the HTTP verb is `POST`.

When building a REST API, it's rare that you'll need to use `[Route(...)]` on an action method because the action accepts all HTTP methods. It's better to use the more specific HTTP verb attribute to be precise about what your API supports. Clients of REST APIs are expected to know what paths and HTTP verbs map to specific logical operations.

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs. This means that many operations, for example, GET and POST on the same logical resource use the same URL. Attribute routing provides a level of control that's needed to carefully design an API's public endpoint layout.

Since an attribute route applies to a specific action, it's easy to make parameters required as part of the route template definition. In the following example, `id` is required as part of the URL path:

```
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The `Products2ApiController.GetProduct(int)` action:

- Is run with URL path like `/products2/3`
- Isn't run with the URL path `/products2`.

The [Consumes] attribute allows an action to limit the supported request content types. For more information, see Define supported request content types with the Consumes attribute.

See Routing for a full description of route templates and related options.

For more information on `[ApiController]`, see ApiController attribute.

Route name

The following code defines a route name of `Products_List`:

```
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Route names can be used to generate a URL based on a specific route. Route names:

- Have no impact on the URL matching behavior of routing.
- Are only used for URL generation.

Route names must be unique application-wide.

Contrast the preceding code with the conventional default route, which defines the `id` parameter as optional (`{id?}`). The ability to precisely specify APIs has advantages, such as allowing `/products` and `/products/5` to be dispatched to different actions.

Combining attribute routes

To make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual actions. Any route templates defined on the controller are prepended to route templates on the actions. Placing a route attribute on the controller makes **all** actions in the controller use attribute routing.

```
[ApiController]
[Route("products")]
public class ProductsApiController : ControllerBase
{
    [HttpGet]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding example:

- The URL path `/products` can match `ProductsApi.ListProducts`
- The URL path `/products/5` can match `ProductsApi.GetProduct(int)`.

Both of these actions only match HTTP `GET` because they're marked with the `[HttpGet]` attribute.

Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller. The following example matches a set of URL paths similar to the default route.

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]
    [Route("Index")]
    [Route("/")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Route("About")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The following table explains the `[Route]` attributes in the preceding code:

ATTRIBUTE	COMBINES WITH `[ROUTE("HOME")]`	DEFINES ROUTE TEMPLATE
`[Route("")]`	Yes	`"Home"`
`[Route("Index")]`	Yes	`"Home/Index"`
`[Route("/")]`	No	`""`
`[Route("About")]`	Yes	`"Home/About"`

Attribute route order

Routing builds a tree and matches all endpoints simultaneously:

- The route entries behave as if placed in an ideal ordering.
- The most specific routes have a chance to execute before the more general routes.

For example, an attribute route like `blog/search/{topic}` is more specific than an attribute route like `blog/{*article}`. The `blog/search/{topic}` route has higher priority, by default, because it's more specific. Using conventional routing, the developer is responsible for placing routes in the desired order.

Attribute routes can configure an order using the Order property. All of the framework provided route attributes include `Order` . Routes are processed according to an ascending sort of the `Order` property. The default order is `0`. Setting a route using `Order = -1` runs before routes that don't set an order. Setting a route using `Order = 1` runs after default route ordering.

Avoid depending on `Order`. If an app's URL-space requires explicit order values to route correctly, then it's likely confusing to clients as well. In general, attribute routing selects the correct route with URL matching. If the default order used for URL generation isn't working, using a route name as an override is usually simpler than applying the `Order` property.

Consider the following two controllers which both define the route matching `/home`:

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Requesting `/home` with the preceding code throws an exception similar to the following:

```
AmbiguousMatchException: The request matched multiple endpoints. Matches:

 WebMvcRouting.Controllers.HomeController.Index
 WebMvcRouting.Controllers.MyDemoController.MyIndex
```

Adding `Order` to one of the route attributes resolves the ambiguity:

```
[Route("")]
[Route("Home", Order = 2)]
[Route("Home/MyIndex")]
public IActionResult MyIndex()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

With the preceding code, `/home` runs the `HomeController.Index` endpoint. To get to the `MyDemoController.MyIndex`, request `/home/MyIndex`. **Note**:

- The preceding code is an example or poor routing design. It was used to illustrate the `Order` property.
- The `Order` property only resolves the ambiguity, that template cannot be matched. It would be better to remove the `[Route("Home")]` template.

See Razor Pages route and app conventions: Route order for information on route order with Razor Pages.

In some cases, an HTTP 500 error is returned with ambiguous routes. Use logging to see which endpoints caused the `AmbiguousMatchException`.

Token replacement in route templates [controller], [action], [area]

For convenience, attribute routes support token replacement for reserved route parameters by enclosing a token in one of the following:

- Square brackets: `[]`
- Curly braces: `{}`

The tokens `[action]`, `[area]`, and `[controller]` are replaced with the values of the action name, area name, and controller name from the action where the route is defined:

```
[Route("[controller]/[action]")]
public class Products0Controller : Controller
{
    [HttpGet]
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }


    [HttpGet("{id}")]
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

```
[HttpGet]
public IActionResult List()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

- Matches `/Products0/List`

```
[HttpGet("{id}")]
public IActionResult Edit(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

- Matches `/Products0/Edit/{id}`

Token replacement occurs as the last step of building the attribute routes. The preceding example behaves the same as the following code:

```
public class Products20Controller : Controller
{
    [HttpGet("[controller]/[action]")]  // Matches '/Products20/List'
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("[controller]/[action]/{id}")]   // Matches '/Products20/Edit/{id}'
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

Attribute routes can also be combined with inheritance. This is powerful combined with token replacement. Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` generates a unique route name for each action:

```
[ApiController]
[Route("api/[controller]/[action]", Name = "[controller]_[action]")]
public abstract class MyBase2Controller : ControllerBase
{
}

public class Products11Controller : MyBase2Controller
{
    [HttpGet]                       // /api/products11/
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]             //    /api/products11/edit/3
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` generates a unique route name for each action.

To match the literal token replacement delimiter `[` or `]`, escape it by repeating the character (`[[` or `]]`).

Use a parameter transformer to customize token replacement

Token replacement can be customized using a parameter transformer. A parameter transformer implements IOutboundParameterTransformer and transforms the value of parameters. For example, a custom `SlugifyParameterTransformer` parameter transformer changes the `SubscriptionManagement` route value to `subscription-management`:

```
public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) { return null; }

        return Regex.Replace(value.ToString(),
                             "([a-z])([A-Z])",
                             "$1-$2",
                             RegexOptions.CultureInvariant,
                             TimeSpan.FromMilliseconds(100)).ToLowerInvariant();
    }
}
```

The RouteTokenTransformerConvention is an application model convention that:

- Applies a parameter transformer to all attribute routes in an application.
- Customizes the attribute route token values as they are replaced.

```
public class SubscriptionManagementController : Controller
{
    [HttpGet("[controller]/[action]")]
    public IActionResult ListAll()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `ListAll` method matches `/subscription-management/list-all`.

The `RouteTokenTransformerConvention` is registered as an option in `ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
    {
        options.Conventions.Add(new RouteTokenTransformerConvention(
                                    new SlugifyParameterTransformer()));
    });
}
```

See MDN web docs on Slug for the definition of Slug.

> **WARNING**
>
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Multiple attribute routes

Attribute routing supports defining multiple routes that reach the same action. The most common usage of this is to mimic the behavior of the default conventional route as shown in the following example:

```
[Route("[controller]")]
public class Products13Controller : Controller
{
    [Route("")]     // Matches 'Products13'
    [Route("Index")] // Matches 'Products13/Index'
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

Putting multiple route attributes on the controller means that each one combines with each of the route attributes on the action methods:

```
[Route("Store")]
[Route("[controller]")]
public class Products6Controller : Controller
{
    [HttpPost("Buy")]       // Matches 'Products6/Buy' and 'Store/Buy'
    [HttpPost("Checkout")]  // Matches 'Products6/Checkout' and 'Store/Checkout'
    public IActionResult Buy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

All the HTTP verb route constraints implement `IActionConstraint`.

When multiple route attributes that implement IActionConstraint are placed on an action:

- Each action constraint combines with the route template applied to the controller.

```
[Route("api/[controller]")]
public class Products7Controller : ControllerBase
{
    [HttpPut("Buy")]        // Matches PUT 'api/Products7/Buy'
    [HttpPost("Checkout")]  // Matches POST 'api/Products7/Checkout'
    public IActionResult Buy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

Using multiple routes on actions might seem useful and powerful, it's better to keep your app's URL space basic and well defined. Use multiple routes on actions **only** where needed, for example, to support existing clients.

Specifying attribute route optional parameters, default values, and constraints

Attribute routes support the same inline syntax as conventional routes to specify optional parameters, default values, and constraints.

```
public class Products14Controller : Controller
{
    [HttpPost("product14/{id:int}")]
    public IActionResult ShowProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code, `[HttpPost("product/{id:int}")]` applies a route constraint. The `ProductsController.ShowProduct` action is matched only by URL paths like `/product/3`. The route template portion `{id:int}` constrains that segment to only integers.

See Route Template Reference for a detailed description of route template syntax.

Custom route attributes using IRouteTemplateProvider

All of the route attributes implement IRouteTemplateProvider. The ASP.NET Core runtime:

- Looks for attributes on controller classes and action methods when the app starts.
- Uses the attributes that implement `IRouteTemplateProvider` to build the initial set of routes.

Implement `IRouteTemplateProvider` to define custom route attributes. Each `IRouteTemplateProvider` allows you to define a single route with a custom route template, order, and name:

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
    public string Template => "api/[controller]";
    public int? Order => 2;
    public string Name { get; set; }
}

[MyApiController]
[ApiController]
public class MyTestApiController : ControllerBase
{
    // GET /api/MyTestApi
    [HttpGet]
    public IActionResult Get()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `Get` method returns `Order = 2, Template = api/MyTestApi`.

Use application model to customize attribute routes

The application model:

- Is an object model created at startup.
- Contains all of the metadata used by ASP.NET Core to route and execute the actions in an app.

The application model includes all of the data gathered from route attributes. The data from route attributes is provided by the `IRouteTemplateProvider` implementation. Conventions:

- Can be written to modify the application model to customize how routing behaves.
- Are read at app startup.

This section shows a basic example of customizing routing using application model. The following code makes routes roughly line up with the folder structure of the project.

```
public class NamespaceRoutingConvention : Attribute, IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
                                                selector.AttributeRouteModel != null);
        if (hasRouteAttributes)
        {
            return;
        }

        var namespc = controller.ControllerType.Namespace;
        if (namespc == null)
            return;
        var template = new StringBuilder();
        template.Append(namespc, _baseNamespace.Length + 1,
                        namespc.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
        template.Append("/[controller]/[action]/{id?}");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
        }
    }
}
```

The following code prevents the `namespace` convention from being applied to controllers that are attribute routed:

```
public void Apply(ControllerModel controller)
{
    var hasRouteAttributes = controller.Selectors.Any(selector =>
                                            selector.AttributeRouteModel != null);
    if (hasRouteAttributes)
    {
        return;
    }
```

For example, the following controller doesn't use `NamespaceRoutingConvention`:

```
[Route("[controller]/[action]/{id?}")]
public class ManagersController : Controller
{
    // /managers/index
    public IActionResult Index()
    {
        var template = ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        return Content($"Index- template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

The `NamespaceRoutingConvention.Apply` method:

- Does nothing if the controller is attribute routed.
- Sets the controllers template based on the `namespace`, with the base `namespace` removed.

The `NamespaceRoutingConvention` can be applied in `Startup.ConfigureServices`:

```
namespace My.Application
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews(options =>
            {
                options.Conventions.Add(
                    new NamespaceRoutingConvention(typeof(Startup).Namespace));
            });
        }
        // Remaining code ommitted for brevity.
```

For example, consider the following controller:

```
using Microsoft.AspNetCore.Mvc;

namespace My.Application.Admin.Controllers
{
    public class UsersController : Controller
    {
        // GET /admin/controllers/users/index
        public IActionResult Index()
        {
            var fullname = typeof(UsersController).FullName;
            var template =
                ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
            var path = Request.Path.Value;

            return Content($"Path: {path} fullname: {fullname}  template:{template}");
        }

        public IActionResult List(int? id)
        {
            var path = Request.Path.Value;
            return Content($"Path: {path} ID:{id}");
        }
    }
}
```

In the preceding code:

- The base `namespace` is `My.Application`.
- The full name of the preceding controller is `My.Application.Admin.Controllers.UsersController`.
- The `NamespaceRoutingConvention` sets the controllers template to `Admin/Controllers/Users/[action]/{id?`.

The `NamespaceRoutingConvention` can also be applied as an attribute on a controller:

```
[NamespaceRoutingConvention("My.Application")]
public class TestController : Controller
{
    // /admin/controllers/test/index
    public IActionResult Index()
    {
        var template = ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        var actionname = ControllerContext.ActionDescriptor.ActionName;
        return Content($"Action- {actionname} template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

Mixed routing: Attribute routing vs conventional routing

ASP.NET Core apps can mix the use of conventional routing and attribute routing. It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. Actions that define attribute routes cannot be reached through the conventional routes and vice-versa. **Any** route attribute on the controller makes **all** actions in the controller attribute routed.

Attribute routing and conventional routing use the same routing engine.

URL Generation and ambient values

Apps can use routing URL generation features to generate URL links to actions. Generating URLs eliminates hardcoding URLs, making code more robust and maintainable. This section focuses on the URL generation features provided by MVC and only cover basics of how URL generation works. See Routing for a detailed description of URL generation.

The IUrlHelper interface is the underlying element of infrastructure between MVC and routing for URL generation. An instance of `IUrlHelper` is available through the `Url` property in controllers, views, and view components.

In the following example, the `IUrlHelper` interface is used through the `Controller.Url` property to generate a URL to another action.

```
public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

If the app is using the default conventional route, the value of the `url` variable is the URL path string `/UrlGeneration/Destination`. This URL path is created by routing by combining:

- The route values from the current request, which are called **ambient values**.
- The values passed to `Url.Action` and substituting those values into the route template:

```
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action = "Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

Each route parameter in the route template has its value substituted by matching names with the values and ambient values. A route parameter that doesn't have a value can:

- Use a default value if it has one.
- Be skipped if it's optional. For example, the `id` from the route template `{controller}/{action}/{id?}`.

URL generation fails if any required route parameter doesn't have a corresponding value. If URL generation fails for a route, the next route is tried until all routes have been tried or a match is found.

The preceding example of `Url.Action` assumes conventional routing. URL generation works similarly with attribute routing, though the concepts are different. With conventional routing:

- The route values are used to expand a template.
- The route values for `controller` and `action` usually appear in that template. This works because the

URLs matched by routing adhere to a convention.

The following example uses attribute routing:

```
public class UrlGenerationAttrController : Controller
{
    [HttpGet("custom")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination()
    {
       return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The `Source` action in the preceding code generates `custom/url/to/destination`.

LinkGenerator was added in ASP.NET Core 3.0 as an alternative to `IUrlHelper`. `LinkGenerator` offers similar but more flexible functionality. Each method on `IUrlHelper` has a corresponding family of methods on `LinkGenerator` as well.

Generating URLs by action name

Url.Action, LinkGenerator.GetPathByAction, and all related overloads all are designed to generate the target endpoint by specifying a controller name and action name.

When using `Url.Action`, the current route values for `controller` and `action` are provided by the runtime:

- The value of `controller` and `action` are part of both ambient values and values. The method `Url.Action` always uses the current values of `action` and `controller` and generates a URL path that routes to the current action.

Routing attempts to use the values in ambient values to fill in information that wasn't provided when generating a URL. Consider a route like `{a}/{b}/{c}/{d}` with ambient values `{ a = Alice, b = Bob, c = Carol, d = David }`:

- Routing has enough information to generate a URL without any additional values.
- Routing has enough information because all route parameters have a value.

If the value `{ d = Donovan }` is added:

- The value `{ d = David }` is ignored.
- The generated URL path is `Alice/Bob/Carol/Donovan`.

Warning: URL paths are hierarchical. In the preceding example, if the value `{ c = Cheryl }` is added:

- Both of the values `{ c = Carol, d = David }` are ignored.
- There is no longer a value for `d` and URL generation fails.
- The desired values of `c` and `d` must be specified to generate a URL.

You might expect to hit this problem with the default route `{controller}/{action}/{id?}`. This problem is rare in practice because `Url.Action` always explicitly specifies a `controller` and `action` value.

Several overloads of Url.Action take a route values object to provide values for route parameters other than `controller` and `action`. The route values object is frequently used with `id`. For example,

`Url.Action("Buy", "Products", new { id = 17 })`. The route values object:

- By convention is usually an object of anonymous type.
- Can be an `IDictionary<>` or a POCO).

Any additional route values that don't match route parameters are put in the query string.

```
public IActionResult Index()
{
    var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
    return Content(url);
}
```

The preceding code generates `/Products/Buy/17?color=red`.

The following code generates an absolute URL:

```
public IActionResult Index2()
{
    var url = Url.Action("Buy", "Products", new { id = 17 }, protocol: Request.Scheme);
    // Returns https://localhost:5001/Products/Buy/17
    return Content(url);
}
```

To create an absolute URL, use one of the following:

- An overload that accepts a `protocol`. For example, the preceding code.
- LinkGenerator.GetUriByAction, which generates absolute URIs by default.

Generate URLs by route

The preceding code demonstrated generating a URL by passing in the controller and action name. `IUrlHelper` also provides the Url.RouteUrl family of methods. These methods are similar to Url.Action, but they don't copy the current values of `action` and `controller` to the route values. The most common usage of `Url.RouteUrl`:

- Specifies a route name to generate the URL.
- Generally doesn't specify a controller or action name.

```
public class UrlGeneration2Controller : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination2", Name = "Destination_Route")]
    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

The following Razor file generates an HTML link to the `Destination_Route`:

```
<h1>Test Links</h1>

<ul>
    <li><a href="@Url.RouteUrl("Destination_Route")">Test Destination_Route</a></li>
</ul>
```

Generate URLs in HTML and Razor

IHtmlHelper provides the HtmlHelper methods Html.BeginForm and Html.ActionLink to generate `<form>` and `<a>` elements respectively. These methods use the Url.Action method to generate a URL and they accept similar arguments. The `Url.RouteUrl` companions for `HtmlHelper` are `Html.BeginRouteForm` and `Html.RouteLink` which have similar functionality.

TagHelpers generate URLs through the `form` TagHelper and the `<a>` TagHelper. Both of these use `IUrlHelper` for their implementation. See Tag Helpers in forms for more information.

Inside views, the `IUrlHelper` is available through the `Url` property for any ad-hoc URL generation not covered by the above.

URL generation in Action Results

The preceding examples showed using `IUrlHelper` in a controller. The most common usage in a controller is to generate a URL as part of an action result.

The ControllerBase and Controller base classes provide convenience methods for action results that reference another action. One typical usage is to redirect after accepting user input:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        ViewData["Message"] = $"Successful edit of customer {id}";
        return RedirectToAction("Index");
    }
    return View(customer);
}
```

The action results factory methods such as RedirectToAction and CreatedAtAction follow a similar pattern to the methods on `IUrlHelper`.

Special case for dedicated conventional routes

Conventional routing can use a special kind of route definition called a dedicated conventional route. In the following example, the route named `blog` is a dedicated conventional route:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
    endpoints.MapControllerRoute(name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Using the preceding route definitions, `Url.Action("Index", "Home")` generates the URL path `/` using the `default` route, but why? You might guess the route values `{ controller = Home, action = Index }` would be

enough to generate a URL using `blog`, and the result would be `/blog?action=Index&controller=Home`.

Dedicated conventional routes rely on a special behavior of default values that don't have a corresponding route parameter that prevents the route from being too greedy with URL generation. In this case the default values are `{ controller = Blog, action = Article }`, and neither `controller` nor `action` appears as a route parameter. When routing performs URL generation, the values provided must match the default values. URL generation using `blog` fails because the values `{ controller = Home, action = Index }` don't match `{ controller = Blog, action = Article }`. Routing then falls back to try `default`, which succeeds.

Areas

Areas are an MVC feature used to organize related functionality into a group as a separate:

- Routing namespace for controller actions.
- Folder structure for views.

Using areas allows an app to have multiple controllers with the same name, as long as they have different areas. Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area` to `controller` and `action`. This section discusses how routing interacts with areas. See Areas for details about how areas are used with views.

The following example configures MVC to use the default conventional route and an `area` route for an `area` named `Blog`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapAreaControllerRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    endpoints.MapControllerRoute("default_route", "{controller}/{action}/{id?}");
});
```

In the preceding code, MapAreaControllerRoute is called to create the `"blog_route"`. The second parameter, `"Blog"`, is the area name.

When matching a URL path like `/Manage/Users/AddUser`, the `"blog_route"` route generates the route values `{ area = Blog, controller = Users, action = AddUser }`. The `area` route value is produced by a default value for `area`. The route created by `MapAreaControllerRoute` is equivalent to the following:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute("blog_route", "Manage/{controller}/{action}/{id?}",
        defaults: new { area = "Blog" }, constraints: new { area = "Blog" });
    endpoints.MapControllerRoute("default_route", "{controller}/{action}/{id?}");
});
```

`MapAreaControllerRoute` creates a route using both a default value and constraint for `area` using the provided area name, in this case `Blog`. The default value ensures that the route always produces `{ area = Blog, ... }`, the constraint requires the value `{ area = Blog, ... }` for URL generation.

Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area.

Using the preceding example, the route values `{ area = Blog, controller = Users, action = AddUser }` match the following action:

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The [Area] attribute is what denotes a controller as part of an area. This controller is in the `Blog` area. Controllers without an `[Area]` attribute are not members of any area, and do **not** match when the `area` route value is provided by routing. In the following example, only the first controller listed can match the route values `{ area = Blog, controller = Users, action = AddUser }`.

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        // GET /zebra/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
    {
        // GET /users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The namespace of each controller is shown here for completeness. If the preceding controllers uses the same namespace, a compiler error would be generated. Class namespaces have no effect on MVC's routing.

The first two controllers are members of areas, and only match when their respective area name is provided by the `area` route value. The third controller isn't a member of any area, and can only match when no value for `area` is provided by routing.

In terms of matching *no value*, the absence of the `area` value is the same as if the value for `area` were null or the empty string.

When executing an action inside an area, the route value for `area` is available as an ambient value for routing to use for URL generation. This means that by default areas act *sticky* for URL generation as demonstrated by the following sample.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapAreaControllerRoute(name: "duck_route",
                                     areaName: "Duck",
                                     pattern: "Manage/{controller}/{action}/{id?}");
    endpoints.MapControllerRoute(name: "default",
                                 pattern: "Manage/{controller=Home}/{action=Index}/{id?}");
});
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
    public class UsersController : Controller
    {
        // GET /Manage/users/GenerateURLInArea
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area.
            var url = Url.Action("Index", "Home");
            // Returns /Manage/Home/Index
            return Content(url);
        }

        // GET /Manage/users/GenerateURLOutsideOfArea
        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area.
            var url = Url.Action("Index", "Home", new { area = "" });
            // Returns /Manage
            return Content(url);
        }
    }
}
```

The following code generates a URL to `/Zebra/Users/AddUser`:

```
public class HomeController : Controller
{
    public IActionResult About()
    {
        var url = Url.Action("AddUser", "Users", new { Area = "Zebra" });
        return Content($"URL: {url}");
    }
```

Action definition

Public methods on a controller, except those with the NonAction attribute, are actions.

Sample code

- The MyDisplayRouteInfo method is included in the sample download and is used to display routing information.
- View or download sample code (how to download)

Debug diagnostics

For detailed routing diagnostic output, set `Logging:LogLevel:Microsoft` to `Debug`. In the development

environment, set the log level in *appsettings.Development.json*:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Debug",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

ASP.NET Core MVC uses the Routing middleware to match the URLs of incoming requests and map them to actions. Routes are defined in startup code or attributes. Routes describe how URL paths should be matched to actions. Routes are also used to generate URLs (for links) sent out in responses.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. See Mixed routing for more information.

This document will explain the interactions between MVC and routing, and how typical MVC apps make use of routing features. See Routing for details on advanced routing.

Setting up Routing Middleware

In your *Configure* method you may see code similar to:

```
app.UseMvc(routes =>
{
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

Inside the call to `UseMvc`, `MapRoute` is used to create a single route, which we'll refer to as the `default` route. Most MVC apps will use a route with a template similar to the `default` route.

The route template `"{controller=Home}/{action=Index}/{id?}"` can match a URL path like `/Products/Details/5` and will extract the route values `{ controller = Products, action = Details, id = 5 }` by tokenizing the path. MVC will attempt to locate a controller named `ProductsController` and run the action `Details`:

```
public class ProductsController : Controller
{
   public IActionResult Details(int id) { ... }
}
```

Note that in this example, model binding would use the value of `id = 5` to set the `id` parameter to `5` when invoking this action. See the Model Binding for more details.

Using the `default` route:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

The route template:

- `{controller=Home}` defines `Home` as the default `controller`
- `{action=Index}` defines `Index` as the default `action`

- `{id?}` defines `id` as optional

Default and optional route parameters don't need to be present in the URL path for a match. See Route Template Reference for a detailed description of route template syntax.

`"{controller=Home}/{action=Index}/{id?}"` can match the URL path `/` and will produce the route values `{ controller = Home, action = Index }`. The values for `controller` and `action` make use of the default values, `id` doesn't produce a value since there's no corresponding segment in the URL path. MVC would use these route values to select the `HomeController` and `Index` action:

```
public class HomeController : Controller
{
  public IActionResult Index() { ... }
}
```

Using this controller definition and route template, the `HomeController.Index` action would be executed for any of the following URL paths:

- `/Home/Index/17`
- `/Home/Index`
- `/Home`
- `/`

The convenience method `UseMvcWithDefaultRoute`:

```
app.UseMvcWithDefaultRoute();
```

Can be used to replace:

```
app.UseMvc(routes =>
{
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

`UseMvc` and `UseMvcWithDefaultRoute` add an instance of `RouterMiddleware` to the middleware pipeline. MVC doesn't interact directly with middleware, and uses routing to handle requests. MVC is connected to the routes through an instance of `MvcRouteHandler`. The code inside of `UseMvc` is similar to the following:

```
var routes = new RouteBuilder(app);

// Add connection to MVC, will be hooked up by calls to MapRoute.
routes.DefaultHandler = new MvcRouteHandler(...);

// Execute callback to register routes.
// routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");

// Create route collection and add the middleware.
app.UseRouter(routes.Build());
```

`UseMvc` doesn't directly define any routes, it adds a placeholder to the route collection for the `attribute` route. The overload `UseMvc(Action<IRouteBuilder>)` lets you add your own routes and also supports attribute routing. `UseMvc` and all of its variations add a placeholder for the attribute route - attribute routing is always available regardless of how you configure `UseMvc`. `UseMvcWithDefaultRoute` defines a default route and

supports attribute routing. The Attribute Routing section includes more details on attribute routing.

Conventional routing

The `default` route:

```
routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
```

The preceding code is an example of a conventional routing. This style is called conventional routing because it establishes a *convention* for URL paths:

- The first path segment maps to the controller name.
- The second maps to the action name.
- The third segment is used for an optional `id`. `id` maps to a model entity.

Using this `default` route, the URL path `/Products/List` maps to the `ProductsController.List` action, and `/Blog/Article/17` maps to `BlogController.Article`. This mapping is based on the controller and action names **only** and isn't based on namespaces, source file locations, or method parameters.

> **TIP**
> Using conventional routing with the default route allows you to build the application quickly without having to come up with a new URL pattern for each action you define. For an application with CRUD style actions, having consistency for the URLs across your controllers can help simplify your code and make your UI more predictable.

> **WARNING**
> The `id` is defined as optional by the route template, meaning that your actions can execute without the ID provided as part of the URL. Usually what will happen if `id` is omitted from the URL is that it will be set to `0` by model binding, and as a result no entity will be found in the database matching `id == 0`. Attribute routing can give you fine-grained control to make the ID required for some actions and not for others. By convention the documentation will include optional parameters like `id` when they're likely to appear in correct usage.

Multiple routes

You can add multiple routes inside `UseMvc` by adding more calls to `MapRoute`. Doing so allows you to define multiple conventions, or to add conventional routes that are dedicated to a specific action, such as:

```
app.UseMvc(routes =>
{
   routes.MapRoute("blog", "blog/{*article}",
            defaults: new { controller = "Blog", action = "Article" });
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

The `blog` route here is a *dedicated conventional route*, meaning that it uses the conventional routing system, but is dedicated to a specific action. Since `controller` and `action` don't appear in the route template as parameters, they can only have the default values, and thus this route will always map to the action `BlogController.Article`.

Routes in the route collection are ordered, and will be processed in the order they're added. So in this example, the `blog` route will be tried before the `default` route.

> **NOTE**
>
> *Dedicated conventional routes* often use **catch-all** route parameters like `{*article}` to capture the remaining portion of the URL path. This can make a route 'too greedy' meaning that it matches URLs that you intended to be matched by other routes. Put the 'greedy' routes later in the route table to solve this.

Fallback

As part of request processing, MVC will verify that the route values can be used to find a controller and action in your application. If the route values don't match an action then the route isn't considered a match, and the next route will be tried. This is called *fallback*, and it's intended to simplify cases where conventional routes overlap.

Disambiguating actions

When two actions match through routing, MVC must disambiguate to choose the 'best' candidate or else throw an exception. For example:

```
public class ProductsController : Controller
{
   public IActionResult Edit(int id) { ... }

   [HttpPost]
   public IActionResult Edit(int id, Product product) { ... }
}
```

This controller defines two actions that would match the URL path `/Products/Edit/17` and route data `{ controller = Products, action = Edit, id = 17 }`. This is a typical pattern for MVC controllers where `Edit(int)` shows a form to edit a product, and `Edit(int, Product)` processes the posted form. To make this possible MVC would need to choose `Edit(int, Product)` when the request is an HTTP `POST` and `Edit(int)` when the HTTP verb is anything else.

The `HttpPostAttribute` (`[HttpPost]`) is an implementation of `IActionConstraint` that will only allow the action to be selected when the HTTP verb is `POST`. The presence of an `IActionConstraint` makes the `Edit(int, Product)` a 'better' match than `Edit(int)`, so `Edit(int, Product)` will be tried first.

You will only need to write custom `IActionConstraint` implementations in specialized scenarios, but it's important to understand the role of attributes like `HttpPostAttribute` - similar attributes are defined for other HTTP verbs. In conventional routing it's common for actions to use the same action name when they're part of a `show form -> submit form` workflow. The convenience of this pattern will become more apparent after reviewing the Understanding IActionConstraint section.

If multiple routes match, and MVC can't find a 'best' route, it will throw an `AmbiguousActionException`.

Route names

The strings `"blog"` and `"default"` in the following examples are route names:

```
app.UseMvc(routes =>
{
   routes.MapRoute("blog", "blog/{*article}",
               defaults: new { controller = "Blog", action = "Article" });
   routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

The route names give the route a logical name so that the named route can be used for URL generation. This greatly simplifies URL creation when the ordering of routes could make URL generation complicated. Route

names must be unique application-wide.

Route names have no impact on URL matching or handling of requests; they're used only for URL generation. Routing has more detailed information on URL generation including URL generation in MVC-specific helpers.

Attribute routing

Attribute routing uses a set of attributes to map actions directly to route templates. In the following example, `app.UseMvc();` is used in the `Configure` method and no route is passed. The `HomeController` will match a set of URLs similar to what the default route `{controller=Home}/{action=Index}/{id?}` would match:

```
public class HomeController : Controller
{
   [Route("")]
   [Route("Home")]
   [Route("Home/Index")]
   public IActionResult Index()
   {
      return View();
   }
   [Route("Home/About")]
   public IActionResult About()
   {
      return View();
   }
   [Route("Home/Contact")]
   public IActionResult Contact()
   {
      return View();
   }
}
```

The `HomeController.Index()` action will be executed for any of the URL paths `/`, `/Home`, or `/Home/Index`.

> **NOTE**
> This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route; the conventional default route handles routes more succinctly. However, attribute routing allows (and requires) precise control of which route templates apply to each action.

With attribute routing the controller name and action names play **no** role in which action is selected. This example will match the same URLs as the previous example.

```
public class MyDemoController : Controller
{
   [Route("")]
   [Route("Home")]
   [Route("Home/Index")]
   public IActionResult MyIndex()
   {
      return View("Index");
   }
   [Route("Home/About")]
   public IActionResult MyAbout()
   {
      return View("About");
   }
   [Route("Home/Contact")]
   public IActionResult MyContact()
   {
      return View("Contact");
   }
}
```

NOTE

The route templates above don't define route parameters for `action`, `area`, and `controller`. In fact, these route parameters are not allowed in attribute routes. Since the route template is already associated with an action, it wouldn't make sense to parse the action name from the URL.

Attribute routing with Http[Verb] attributes

Attribute routing can also make use of the `Http[Verb]` attributes such as `HttpPostAttribute`. All of these attributes can accept a route template. This example shows two actions that match the same route template:

```
[HttpGet("/products")]
public IActionResult ListProducts()
{
   // ...
}

[HttpPost("/products")]
public IActionResult CreateProduct(...)
{
   // ...
}
```

For a URL path like `/products` the `ProductsApi.ListProducts` action will be executed when the HTTP verb is `GET` and `ProductsApi.CreateProduct` will be executed when the HTTP verb is `POST`. Attribute routing first matches the URL against the set of route templates defined by route attributes. Once a route template matches, `IActionConstraint` constraints are applied to determine which actions can be executed.

TIP

When building a REST API, it's rare that you will want to use `[Route(...)]` on an action method as the action will accept all HTTP methods. It's better to use the more specific `Http*Verb*Attributes` to be precise about what your API supports. Clients of REST APIs are expected to know what paths and HTTP verbs map to specific logical operations.

Since an attribute route applies to a specific action, it's easy to make parameters required as part of the route template definition. In this example, `id` is required as part of the URL path.

```
public class ProductsApiController : Controller
{
   [HttpGet("/products/{id}", Name = "Products_List")]
   public IActionResult GetProduct(int id) { ... }
}
```

The `ProductsApi.GetProduct(int)` action will be executed for a URL path like `/products/3` but not for a URL path like `/products`. See Routing for a full description of route templates and related options.

Route Name

The following code defines a *route name* of `Products_List`:

```
public class ProductsApiController : Controller
{
   [HttpGet("/products/{id}", Name = "Products_List")]
   public IActionResult GetProduct(int id) { ... }
}
```

Route names can be used to generate a URL based on a specific route. Route names have no impact on the URL matching behavior of routing and are only used for URL generation. Route names must be unique application-wide.

> **NOTE**
> Contrast this with the conventional *default route*, which defines the `id` parameter as optional (`{id?}`). This ability to precisely specify APIs has advantages, such as allowing `/products` and `/products/5` to be dispatched to different actions.

Combining routes

To make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual actions. Any route templates defined on the controller are prepended to route templates on the actions. Placing a route attribute on the controller makes **all** actions in the controller use attribute routing.

```
[Route("products")]
public class ProductsApiController : Controller
{
   [HttpGet]
   public IActionResult ListProducts() { ... }

   [HttpGet("{id}")]
   public ActionResult GetProduct(int id) { ... }
}
```

In this example the URL path `/products` can match `ProductsApi.ListProducts`, and the URL path `/products/5` can match `ProductsApi.GetProduct(int)`. Both of these actions only match HTTP `GET` because they're marked with the `HttpGetAttribute`.

Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller. This example matches a set of URL paths similar to the *default route*.

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]      // Combines to define the route template "Home"
    [Route("Index")] // Combines to define the route template "Home/Index"
    [Route("/")]     // Doesn't combine, defines the route template ""
    public IActionResult Index()
    {
        ViewData["Message"] = "Home index";
        var url = Url.Action("Index", "Home");
        ViewData["Message"] = "Home index" + "var url = Url.Action; =  " + url;
        return View();
    }

    [Route("About")] // Combines to define the route template "Home/About"
    public IActionResult About()
    {
        return View();
    }
}
```

Ordering attribute routes

In contrast to conventional routes, which execute in a defined order, attribute routing builds a tree and matches all routes simultaneously. This behaves as-if the route entries were placed in an ideal ordering; the most specific routes have a chance to execute before the more general routes.

For example, a route like `blog/search/{topic}` is more specific than a route like `blog/{*article}`. Logically speaking the `blog/search/{topic}` route 'runs' first, by default, because that's the only sensible ordering. Using conventional routing, the developer is responsible for placing routes in the desired order.

Attribute routes can configure an order, using the `Order` property of all of the framework provided route attributes. Routes are processed according to an ascending sort of the `Order` property. The default order is `0`. Setting a route using `Order = -1` will run before routes that don't set an order. Setting a route using `Order = 1` will run after default route ordering.

> **TIP**
>
> Avoid depending on `Order`. If your URL-space requires explicit order values to route correctly, then it's likely confusing to clients as well. In general attribute routing will select the correct route with URL matching. If the default order used for URL generation isn't working, using route name as an override is usually simpler than applying the `Order` property.

Razor Pages routing and MVC controller routing share an implementation. Information on route order in the Razor Pages topics is available at Razor Pages route and app conventions: Route order.

Token replacement in route templates ([controller], [action], [area])

For convenience, attribute routes support *token replacement* by enclosing a token in square-brackets (`[`, `]`). The tokens `[action]`, `[area]`, and `[controller]` are replaced with the values of the action name, area name, and controller name from the action where the route is defined. In the following example, the actions match URL paths as described in the comments:

```
[Route("[controller]/[action]")]
public class ProductsController : Controller
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

Token replacement occurs as the last step of building the attribute routes. The above example will behave the same as the following code:

```
public class ProductsController : Controller
{
    [HttpGet("[controller]/[action]")] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("[controller]/[action]/{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

Attribute routes can also be combined with inheritance. This is particularly powerful combined with token replacement.

```
[Route("api/[controller]")]
public abstract class MyBaseController : Controller { ... }

public class ProductsController : MyBaseController
{
   [HttpGet] // Matches '/api/Products'
   public IActionResult List() { ... }

   [HttpPut("{id}")] // Matches '/api/Products/{id}'
   public IActionResult Edit(int id) { ... }
}
```

Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` generates a unique route name for each action.

To match the literal token replacement delimiter `[` or `]`, escape it by repeating the character (`[[` or `]]`).

Use a parameter transformer to customize token replacement

Token replacement can be customized using a parameter transformer. A parameter transformer implements `IOutboundParameterTransformer` and transforms the value of parameters. For example, a custom `SlugifyParameterTransformer` parameter transformer changes the `SubscriptionManagement` route value to `subscription-management`.

The `RouteTokenTransformerConvention` is an application model convention that:

- Applies a parameter transformer to all attribute routes in an application.
- Customizes the attribute route token values as they are replaced.

```
public class SubscriptionManagementController : Controller
{
    [HttpGet("[controller]/[action]")] // Matches '/subscription-management/list-all'
    public IActionResult ListAll() { ... }
}
```

The `RouteTokenTransformerConvention` is registered as an option in `ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new RouteTokenTransformerConvention(
                                    new SlugifyParameterTransformer()));
    });
}

public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) { return null; }

        // Slugify value
        return Regex.Replace(value.ToString(), "([a-z])([A-Z])", "$1-$2").ToLower();
    }
}
```

Multiple Routes

Attribute routing supports defining multiple routes that reach the same action. The most common usage of this is to mimic the behavior of the *default conventional route* as shown in the following example:

```
[Route("[controller]")]
public class ProductsController : Controller
{
   [Route("")]     // Matches 'Products'
   [Route("Index")] // Matches 'Products/Index'
   public IActionResult Index()
}
```

Putting multiple route attributes on the controller means that each one will combine with each of the route attributes on the action methods.

```
[Route("Store")]
[Route("[controller]")]
public class ProductsController : Controller
{
   [HttpPost("Buy")]     // Matches 'Products/Buy' and 'Store/Buy'
   [HttpPost("Checkout")] // Matches 'Products/Checkout' and 'Store/Checkout'
   public IActionResult Buy()
}
```

When multiple route attributes (that implement `IActionConstraint`) are placed on an action, then each action constraint combines with the route template from the attribute that defined it.

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
   [HttpPut("Buy")]      // Matches PUT 'api/Products/Buy'
   [HttpPost("Checkout")] // Matches POST 'api/Products/Checkout'
   public IActionResult Buy()
}
```

TIP

While using multiple routes on actions can seem powerful, it's better to keep your application's URL space simple and well-defined. Use multiple routes on actions only where needed, for example to support existing clients.

Specifying attribute route optional parameters, default values, and constraints

Attribute routes support the same inline syntax as conventional routes to specify optional parameters, default values, and constraints.

```
[HttpPost("product/{id:int}")]
public IActionResult ShowProduct(int id)
{
   // ...
}
```

See Route Template Reference for a detailed description of route template syntax.

Custom route attributes using `IRouteTemplateProvider`

All of the route attributes provided in the framework (`[Route(...)]` , `[HttpGet(...)]` , etc.) implement the `IRouteTemplateProvider` interface. MVC looks for attributes on controller classes and action methods when the app starts and uses the ones that implement `IRouteTemplateProvider` to build the initial set of routes.

You can implement `IRouteTemplateProvider` to define your own route attributes. Each `IRouteTemplateProvider` allows you to define a single route with a custom route template, order, and name:

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
   public string Template => "api/[controller]";

   public int? Order { get; set; }

   public string Name { get; set; }
}
```

The attribute from the above example automatically sets the `Template` to `"api/[controller]"` when `[MyApiController]` is applied.

Using Application Model to customize attribute routes

The *application model* is an object model created at startup with all of the metadata used by MVC to route and execute your actions. The *application model* includes all of the data gathered from route attributes (through `IRouteTemplateProvider`). You can write *conventions* to modify the application model at startup time to customize how routing behaves. This section shows a simple example of customizing routing using application model.

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;
using System.Text;
public class NamespaceRoutingConvention : IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
                                                selector.AttributeRouteModel != null);
        if (hasRouteAttributes)
        {
            // This controller manually defined some routes, so treat this
            // as an override and not apply the convention here.
            return;
        }

        // Use the namespace and controller name to infer a route for the controller.
        //
        // Example:
        //
        //  controller.ControllerTypeInfo ->    "My.Application.Admin.UsersController"
        //  baseNamespace ->                    "My.Application"
        //
        //  template =>                         "Admin/[controller]"
        //
        // This makes your routes roughly line up with the folder structure of your project.
        //
        var namespc = controller.ControllerType.Namespace;
        if (namespc == null)
            return;
        var template = new StringBuilder();
        template.Append(namespc, _baseNamespace.Length + 1,
                        namespc.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
        template.Append("/[controller]");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
        }
    }
}
```

Mixed routing: Attribute routing vs conventional routing

MVC applications can mix the use of conventional routing and attribute routing. It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. Actions that define attribute routes cannot be reached through the conventional routes and vice-versa. **Any** route attribute on the controller makes all actions in the controller attribute routed.

NOTE

What distinguishes the two types of routing systems is the process applied after a URL matches a route template. In conventional routing, the route values from the match are used to choose the action and controller from a lookup table of all conventional routed actions. In attribute routing, each template is already associated with an action, and no further lookup is needed.

Complex segments

Complex segments (for example, `[Route("/dog{token}cat")]`), are processed by matching up literals from right to left in a non-greedy way. See the source code for a description. For more information, see this issue.

URL Generation

MVC applications can use routing's URL generation features to generate URL links to actions. Generating URLs eliminates hardcoding URLs, making your code more robust and maintainable. This section focuses on the URL generation features provided by MVC and will only cover basics of how URL generation works. See Routing for a detailed description of URL generation.

The `IUrlHelper` interface is the underlying piece of infrastructure between MVC and routing for URL generation. You'll find an instance of `IUrlHelper` available through the `Url` property in controllers, views, and view components.

In this example, the `IUrlHelper` interface is used through the `Controller.Url` property to generate a URL to another action.

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return Content($"Go check out {url}, it's really great.");
    }

    public IActionResult Destination()
    {
        return View();
    }
}
```

If the application is using the default conventional route, the value of the `url` variable will be the URL path string `/UrlGeneration/Destination`. This URL path is created by routing by combining the route values from the current request (ambient values), with the values passed to `Url.Action` and substituting those values into the route template:

```
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action = "Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

Each route parameter in the route template has its value substituted by matching names with the values and ambient values. A route parameter that doesn't have a value can use a default value if it has one, or be

skipped if it's optional (as in the case of `id` in this example). URL generation will fail if any required route parameter doesn't have a corresponding value. If URL generation fails for a route, the next route is tried until all routes have been tried or a match is found.

The example of `Url.Action` above assumes conventional routing, but URL generation works similarly with attribute routing, though the concepts are different. With conventional routing, the route values are used to expand a template, and the route values for `controller` and `action` usually appear in that template - this works because the URLs matched by routing adhere to a *convention*. In attribute routing, the route values for `controller` and `action` are not allowed to appear in the template - they're instead used to look up which template to use.

This example uses attribute routing:

```
// In Startup class
public void Configure(IApplicationBuilder app)
{
    app.UseMvc();
}
```

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination"); // Generates /custom/url/to/destination
        return Content($"Go check out {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination() {
        return View();
    }
}
```

MVC builds a lookup table of all attribute routed actions and will match the `controller` and `action` values to select the route template to use for URL generation. In the sample above, `custom/url/to/destination` is generated.

Generating URLs by action name

`Url.Action` (`IUrlHelper` . `Action`) and all related overloads all are based on that idea that you want to specify what you're linking to by specifying a controller name and action name.

> **NOTE**
>
> When using `Url.Action`, the current route values for `controller` and `action` are specified for you - the value of `controller` and `action` are part of both *ambient values* **and** *values*. The method `Url.Action`, always uses the current values of `action` and `controller` and will generate a URL path that routes to the current action.

Routing attempts to use the values in ambient values to fill in information that you didn't provide when generating a URL. Using a route like `{a}/{b}/{c}/{d}` and ambient values `{ a = Alice, b = Bob, c = Carol, d = David }`, routing has enough information to generate a URL without any additional values - since all route parameters have a value. If you added the value `{ d = Donovan }`, the value `{ d = David }` would be ignored, and the generated URL path would be `Alice/Bob/Carol/Donovan`.

WARNING

URL paths are hierarchical. In the example above, if you added the value `{ c = Cheryl }`, both of the values `{ c = Carol, d = David }` would be ignored. In this case we no longer have a value for `d` and URL generation will fail. You would need to specify the desired value of `c` and `d`. You might expect to hit this problem with the default route (`{controller}/{action}/{id?}`) - but you will rarely encounter this behavior in practice as `Url.Action` will always explicitly specify a `controller` and `action` value.

Longer overloads of `Url.Action` also take an additional *route values* object to provide values for route parameters other than `controller` and `action`. You will most commonly see this used with `id` like `Url.Action("Buy", "Products", new { id = 17 })`. By convention the *route values* object is usually an object of anonymous type, but it can also be an `IDictionary<>` or a *plain old .NET object*. Any additional route values that don't match route parameters are put in the query string.

```
using Microsoft.AspNetCore.Mvc;

public class TestController : Controller
{
    public IActionResult Index()
    {
        // Generates /Products/Buy/17?color=red
        var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
        return Content(url);
    }
}
```

TIP

To create an absolute URL, use an overload that accepts a `protocol`:
`Url.Action("Buy", "Products", new { id = 17 }, protocol: Request.Scheme)`

Generating URLs by route

The code above demonstrated generating a URL by passing in the controller and action name. `IUrlHelper` also provides the `Url.RouteUrl` family of methods. These methods are similar to `Url.Action`, but they don't copy the current values of `action` and `controller` to the route values. The most common usage is to specify a route name to use a specific route to generate the URL, generally *without* specifying a controller or action name.

```
using Microsoft.AspNetCore.Mvc;

public class UrlGenerationController : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route"); // Generates /custom/url/to/destination
        return Content($"See {url}, it's really great.");
    }

    [HttpGet("custom/url/to/destination", Name = "Destination_Route")]
    public IActionResult Destination() {
        return View();
    }
}
```

Generating URLs in HTML

`IHtmlHelper` provides the `HtmlHelper` methods `Html.BeginForm` and `Html.ActionLink` to generate `<form>` and `<a>` elements respectively. These methods use the `Url.Action` method to generate a URL and they accept similar arguments. The `Url.RouteUrl` companions for `HtmlHelper` are `Html.BeginRouteForm` and `Html.RouteLink` which have similar functionality.

TagHelpers generate URLs through the `form` TagHelper and the `<a>` TagHelper. Both of these use `IUrlHelper` for their implementation. See Working with Forms for more information.

Inside views, the `IUrlHelper` is available through the `Url` property for any ad-hoc URL generation not covered by the above.

Generating URLS in Action Results

The examples above have shown using `IUrlHelper` in a controller, while the most common usage in a controller is to generate a URL as part of an action result.

The `ControllerBase` and `Controller` base classes provide convenience methods for action results that reference another action. One typical usage is to redirect after accepting user input.

```
public IActionResult Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        return RedirectToAction("Index");
    }
    return View(customer);
}
```

The action results factory methods follow a similar pattern to the methods on `IUrlHelper`.

Special case for dedicated conventional routes

Conventional routing can use a special kind of route definition called a *dedicated conventional route*. In the example below, the route named `blog` is a dedicated conventional route.

```
app.UseMvc(routes =>
{
    routes.MapRoute("blog", "blog/{*article}",
        defaults: new { controller = "Blog", action = "Article" });
    routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
});
```

Using these route definitions, `Url.Action("Index", "Home")` will generate the URL path `/` with the `default` route, but why? You might guess the route values `{ controller = Home, action = Index }` would be enough to generate a URL using `blog`, and the result would be `/blog?action=Index&controller=Home`.

Dedicated conventional routes rely on a special behavior of default values that don't have a corresponding route parameter that prevents the route from being "too greedy" with URL generation. In this case the default values are `{ controller = Blog, action = Article }`, and neither `controller` nor `action` appears as a route parameter. When routing performs URL generation, the values provided must match the default values. URL generation using `blog` will fail because the values `{ controller = Home, action = Index }` don't match `{ controller = Blog, action = Article }`. Routing then falls back to try `default`, which succeeds.

Areas

Areas are an MVC feature used to organize related functionality into a group as a separate routing-namespace (for controller actions) and folder structure (for views). Using areas allows an application to have

multiple controllers with the same name - as long as they have different *areas*. Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area` to `controller` and `action`. This section will discuss how routing interacts with areas - see Areas for details about how areas are used with views.

The following example configures MVC to use the default conventional route and an *area route* for an area named `Blog`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapAreaControllerRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
    endpoints.MapControllerRoute("default_route", "{controller}/{action}/{id?}");
});
```

When matching a URL path like `/Manage/Users/AddUser`, the first route will produce the route values `{ area = Blog, controller = Users, action = AddUser }`. The `area` route value is produced by a default value for `area`, in fact the route created by `MapAreaRoute` is equivalent to the following:

```

```

`MapAreaRoute` creates a route using both a default value and constraint for `area` using the provided area name, in this case `Blog`. The default value ensures that the route always produces `{ area = Blog, ... }`, the constraint requires the value `{ area = Blog, ... }` for URL generation.

> **TIP**
> Conventional routing is order-dependent. In general, routes with areas should be placed earlier in the route table as they're more specific than routes without an area.

Using the above example, the route values would match the following action:

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The `AreaAttribute` is what denotes a controller as part of an area, we say that this controller is in the `Blog` area. Controllers without an `[Area]` attribute are not members of any area, and will **not** match when the `area` route value is provided by routing. In the following example, only the first controller listed can match the route values `{ area = Blog, controller = Users, action = AddUser }`.

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        // GET /zebra/users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
    {
        // GET /users/adduser
        public IActionResult AddUser()
        {
            var area = ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName = ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

NOTE

The namespace of each controller is shown here for completeness - otherwise the controllers would have a naming conflict and generate a compiler error. Class namespaces have no effect on MVC's routing.

The first two controllers are members of areas, and only match when their respective area name is provided by the `area` route value. The third controller isn't a member of any area, and can only match when no value for `area` is provided by routing.

NOTE

In terms of matching *no value*, the absence of the `area` value is the same as if the value for `area` were null or the empty string.

When executing an action inside an area, the route value for `area` will be available as an *ambient value* for routing to use for URL generation. This means that by default areas act *sticky* for URL generation as demonstrated by the following sample.

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
    public class UsersController : Controller
    {
        // GET /Manage/users/GenerateURLInArea
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area.
            var url = Url.Action("Index", "Home");
            // Returns /Manage/Home/Index
            return Content(url);
        }

        // GET /Manage/users/GenerateURLOutsideOfArea
        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area.
            var url = Url.Action("Index", "Home", new { area = "" });
            // Returns /Manage
            return Content(url);
        }
    }
}
```

Understanding IActionConstraint

NOTE

This section is a deep-dive on framework internals and how MVC chooses an action to execute. A typical application won't need a custom `IActionConstraint`

You have likely already used `IActionConstraint` even if you're not familiar with the interface. The `[HttpGet]` Attribute and similar `[Http-VERB]` attributes implement `IActionConstraint` in order to limit the execution of

an action method.

```
public class ProductsController : Controller
{
    [HttpGet]
    public IActionResult Edit() { }

    public IActionResult Edit(...) { }
}
```

Assuming the default conventional route, the URL path `/Products/Edit` would produce the values `{ controller = Products, action = Edit }`, which would match **both** of the actions shown here. In `IActionConstraint` terminology we would say that both of these actions are considered candidates - as they both match the route data.

When the `HttpGetAttribute` executes, it will say that *Edit()* is a match for *GET* and isn't a match for any other HTTP verb. The `Edit(...)` action doesn't have any constraints defined, and so will match any HTTP verb. So assuming a `POST` - only `Edit(...)` matches. But, for a `GET` both actions can still match - however, an action with an `IActionConstraint` is always considered *better* than an action without. So because `Edit()` has `[HttpGet]` it's considered more specific, and will be selected if both actions can match.

Conceptually, `IActionConstraint` is a form of *overloading*, but instead of overloading methods with the same name, it's overloading between actions that match the same URL. Attribute routing also uses `IActionConstraint` and can result in actions from different controllers both being considered candidates.

Implementing IActionConstraint

The simplest way to implement an `IActionConstraint` is to create a class derived from `System.Attribute` and place it on your actions and controllers. MVC will automatically discover any `IActionConstraint` that are applied as attributes. You can use the application model to apply constraints, and this is probably the most flexible approach as it allows you to metaprogram how they're applied.

In the following example, a constraint chooses an action based on a *country code* from the route data. The full sample on GitHub.

```
public class CountrySpecificAttribute : Attribute, IActionConstraint
{
    private readonly string _countryCode;

    public CountrySpecificAttribute(string countryCode)
    {
        _countryCode = countryCode;
    }

    public int Order
    {
        get
        {
            return 0;
        }
    }

    public bool Accept(ActionConstraintContext context)
    {
        return string.Equals(
            context.RouteContext.RouteData.Values["country"].ToString(),
            _countryCode,
            StringComparison.OrdinalIgnoreCase);
    }
}
```

You are responsible for implementing the `Accept` method and choosing an 'Order' for the constraint to execute. In this case, the `Accept` method returns `true` to denote the action is a match when the `country` route value matches. This is different from a `RouteValueAttribute` in that it allows fallback to a non-attributed action. The sample shows that if you define an `en-US` action then a country code like `fr-FR` will fall back to a more generic controller that doesn't have `[CountrySpecific(...)]` applied.

The `Order` property decides which *stage* the constraint is part of. Action constraints run in groups based on the `Order`. For example, all of the framework provided HTTP method attributes use the same `Order` value so that they run in the same stage. You can have as many stages as you need to implement your desired policies.

> **TIP**
> To decide on a value for `Order` think about whether or not your constraint should be applied before HTTP methods. Lower numbers run first.

Dependency injection into controllers in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By Shadi Namrouti, Rick Anderson, and Steve Smith

ASP.NET Core MVC controllers request dependencies explicitly via constructors. ASP.NET Core has built-in support for dependency injection (DI). DI makes apps easier to test and maintain.

View or download sample code (how to download)

Constructor Injection

Services are added as a constructor parameter, and the runtime resolves the service from the service container. Services are typically defined using interfaces. For example, consider an app that requires the current time. The following interface exposes the `IDateTime` service:

```
public interface IDateTime
{
    DateTime Now { get; }
}
```

The following code implements the `IDateTime` interface:

```
public class SystemDateTime : IDateTime
{
    public DateTime Now
    {
        get { return DateTime.Now; }
    }
}
```

Add the service to the service container:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();

    services.AddControllersWithViews();
}
```

For more information on AddSingleton, see DI service lifetimes.

The following code displays a greeting to the user based on the time of day:

```
public class HomeController : Controller
{
    private readonly IDateTime _dateTime;

    public HomeController(IDateTime dateTime)
    {
        _dateTime = dateTime;
    }

    public IActionResult Index()
    {
        var serverTime = _dateTime.Now;
        if (serverTime.Hour < 12)
        {
            ViewData["Message"] = "It's morning here - Good Morning!";
        }
        else if (serverTime.Hour < 17)
        {
            ViewData["Message"] = "It's afternoon here - Good Afternoon!";
        }
        else
        {
            ViewData["Message"] = "It's evening here - Good Evening!";
        }
        return View();
    }
```

Run the app and a message is displayed based on the time.

Action injection with FromServices

The FromServicesAttribute enables injecting a service directly into an action method without using constructor injection:

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    return Content( $"Current server time: {dateTime.Now}");
}
```

Access settings from a controller

Accessing app or configuration settings from within a controller is a common pattern. The *options pattern* described in Options pattern in ASP.NET Core is the preferred approach to manage settings. Generally, don't directly inject IConfiguration into a controller.

Create a class that represents the options. For example:

```
public class SampleWebSettings
{
    public string Title { get; set; }
    public int Updates { get; set; }
}
```

Add the configuration class to the services collection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();
    services.Configure<SampleWebSettings>(Configuration);

    services.AddControllersWithViews();
}
```

Configure the app to read the settings from a JSON-formatted file:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("samplewebsettings.json",
                    optional: false,
                    reloadOnChange: true);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code requests the `IOptions<SampleWebSettings>` settings from the service container and uses them in the `Index` method:

```
public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}
```

Additional resources

- See Test controller logic in ASP.NET Core to learn how to make code easier to test by explicitly requesting dependencies in controllers.
- Replace the default dependency injection container with a third party implementation.

By Shadi Namrouti, Rick Anderson, and Steve Smith

ASP.NET Core MVC controllers request dependencies explicitly via constructors. ASP.NET Core has built-in support

for dependency injection (DI). DI makes apps easier to test and maintain.

View or download sample code (how to download)

Constructor Injection

Services are added as a constructor parameter, and the runtime resolves the service from the service container. Services are typically defined using interfaces. For example, consider an app that requires the current time. The following interface exposes the `IDateTime` service:

```
public interface IDateTime
{
    DateTime Now { get; }
}
```

The following code implements the `IDateTime` interface:

```
public class SystemDateTime : IDateTime
{
    public DateTime Now
    {
        get { return DateTime.Now; }
    }
}
```

Add the service to the service container:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

For more information on AddSingleton, see DI service lifetimes.

The following code displays a greeting to the user based on the time of day:

```
public class HomeController : Controller
{
    private readonly IDateTime _dateTime;

    public HomeController(IDateTime dateTime)
    {
        _dateTime = dateTime;
    }

    public IActionResult Index()
    {
        var serverTime = _dateTime.Now;
        if (serverTime.Hour < 12)
        {
            ViewData["Message"] = "It's morning here - Good Morning!";
        }
        else if (serverTime.Hour < 17)
        {
            ViewData["Message"] = "It's afternoon here - Good Afternoon!";
        }
        else
        {
            ViewData["Message"] = "It's evening here - Good Evening!";
        }
        return View();
    }
```

Run the app and a message is displayed based on the time.

Action injection with FromServices

The FromServicesAttribute enables injecting a service directly into an action method without using constructor injection:

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    ViewData["Message"] = $"Current server time: {dateTime.Now}";

    return View();
}
```

Access settings from a controller

Accessing app or configuration settings from within a controller is a common pattern. The *options pattern* described in Options pattern in ASP.NET Core is the preferred approach to manage settings. Generally, don't directly inject IConfiguration into a controller.

Create a class that represents the options. For example:

```
public class SampleWebSettings
{
    public string Title { get; set; }
    public int Updates { get; set; }
}
```

Add the configuration class to the services collection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();
    services.Configure<SampleWebSettings>(Configuration);

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Configure the app to read the settings from a JSON-formatted file:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            config.AddJsonFile("samplewebsettings.json",
                               optional: false,        // File is not optional.
                               reloadOnChange: false);
        })
        .UseStartup<Startup>();
}
```

The following code requests the `IOptions<SampleWebSettings>` settings from the service container and uses them in the `Index` method:

```
public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}
```

Additional resources

- See Test controller logic in ASP.NET Core to learn how to make code easier to test by explicitly requesting dependencies in controllers.
- Replace the default dependency injection container with a third party implementation.

Dependency injection into views in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By Steve Smith

ASP.NET Core supports dependency injection into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. You should try to maintain separation of concerns between your controllers and views. Most of the data your views display should be passed in from the controller.

View or download sample code (how to download)

Configuration injection

appsettings.json values can be injected directly into a view.

Example of an *appsettings.json* file:

```
{
   "root": {
      "parent": {
         "child": "myvalue"
      }
   }
}
```

The syntax for `@inject`: `@inject <type> <name>`

An example using `@inject`:

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration
@{
   string myValue = Configuration["root:parent:child"];
   ...
}
```

Service injection

A service can be injected into a view using the `@inject` directive. You can think of `@inject` as adding a property to the view, and populating the property using DI.

```
@using System.Threading.Tasks
@using ViewInjectSample.Model
@using ViewInjectSample.Model.Services
@model IEnumerable<ToDoItem>
@inject StatisticsService StatsService
<!DOCTYPE html>
<html>
<head>
    <title>To Do Items</title>
</head>
<body>
    <div>
        <h1>To Do Items</h1>
        <ul>
            <li>Total Items: @StatsService.GetCount()</li>
            <li>Completed: @StatsService.GetCompletedCount()</li>
            <li>Avg. Priority: @StatsService.GetAveragePriority()</li>
        </ul>
        <table>
            <tr>
                <th>Name</th>
                <th>Priority</th>
                <th>Is Done?</th>
            </tr>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.Name</td>
                    <td>@item.Priority</td>
                    <td>@item.IsDone</td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

This view displays a list of `ToDoItem` instances, along with a summary showing overall statistics. The summary is populated from the injected `StatisticsService`. This service is registered for dependency injection in `ConfigureServices` in *Startup.cs*:

```
// For more information on how to configure your application, visit http://go.microsoft.com/fwlink/?
LinkID=398940
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
    services.AddTransient<StatisticsService>();
    services.AddTransient<ProfileOptionsService>();
```

The `StatisticsService` performs some calculations on the set of `ToDoItem` instances, which it accesses via a repository:

```
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _toDoItemRepository;

        public StatisticsService(IToDoItemRepository toDoItemRepository)
        {
            _toDoItemRepository = toDoItemRepository;
        }

        public int GetCount()
        {
            return _toDoItemRepository.List().Count();
        }

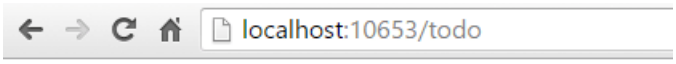
        public int GetCompletedCount()
        {
            return _toDoItemRepository.List().Count(x => x.IsDone);
        }

        public double GetAveragePriority()
        {
            if (_toDoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _toDoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

The sample repository uses an in-memory collection. The implementation shown above (which operates on all of the data in memory) isn't recommended for large, remotely accessed data sets.

The sample displays data from the model bound to the view and the service injected into the view:



To Do Items

- Total Items: 50
- Completed: 17
- Avg. Priority: 3

Name	Priority	Is Done?
Task 1	1	True
Task 2	2	False
Task 3	3	False
Task 4	4	True
Task 5	5	False

Populating Lookup Data

View injection can be useful to populate options in UI elements, such as dropdown lists. Consider a user profile form that includes options for specifying gender, state, and other preferences. Rendering such a form using a standard MVC approach would require the controller to request data access services for each of these sets of

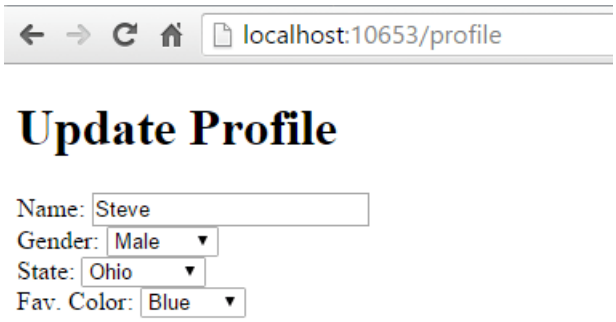
options, and then populate a model or `ViewBag` with each set of options to be bound.

An alternative approach injects services directly into the view to obtain the options. This minimizes the amount of code required by the controller, moving this view element construction logic into the view itself. The controller action to display a profile editing form only needs to pass the form the profile instance:

```
using Microsoft.AspNetCore.Mvc;
using ViewInjectSample.Model;

namespace ViewInjectSample.Controllers
{
    public class ProfileController : Controller
    {
        [Route("Profile")]
        public IActionResult Index()
        {
            // TODO: look up profile based on logged-in user
            var profile = new Profile()
            {
                Name = "Steve",
                FavColor = "Blue",
                Gender = "Male",
                State = new State("Ohio","OH")
            };
            return View(profile);
        }
    }
}
```

The HTML form used to update these preferences includes dropdown lists for three of the properties:



These lists are populated by a service that has been injected into the view:

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
    <title>Update Profile</title>
</head>
<body>
<div>
    <h1>Update Profile</h1>
    Name: @Html.TextBoxFor(m => m.Name)
    <br/>
    Gender: @Html.DropDownList("Gender",
           Options.ListGenders().Select(g =>
                new SelectListItem() { Text = g, Value = g }))
    <br/>

    State: @Html.DropDownListFor(m => m.State.Code,
           Options.ListStates().Select(s =>
                new SelectListItem() { Text = s.Name, Value = s.Code}))
    <br />

    Fav. Color: @Html.DropDownList("FavColor",
           Options.ListColors().Select(c =>
                new SelectListItem() { Text = c, Value = c }))
    </div>
</body>
</html>
```

The `ProfileOptionsService` is a UI-level service designed to provide just the data needed for this form:

```
using System.Collections.Generic;

namespace ViewInjectSample.Model.Services
{
    public class ProfileOptionsService
    {
        public List<string> ListGenders()
        {
            // keeping this simple
            return new List<string>() {"Female", "Male"};
        }

        public List<State> ListStates()
        {
            // a few states from USA
            return new List<State>()
            {
                new State("Alabama", "AL"),
                new State("Alaska", "AK"),
                new State("Ohio", "OH")
            };
        }

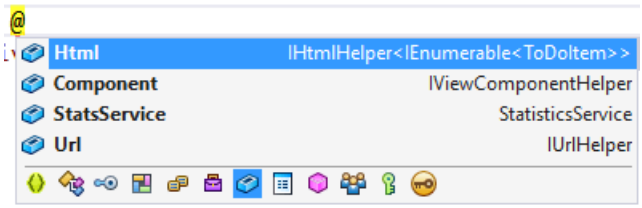
        public List<string> ListColors()
        {
            return new List<string>() { "Blue","Green","Red","Yellow" };
        }
    }
}
```

IMPORTANT

Don't forget to register types you request through dependency injection in `Startup.ConfigureServices`. An unregistered type throws an exception at runtime because the service provider is internally queried via GetRequiredService.

Overriding Services

In addition to injecting new services, this technique can also be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:



As you can see, the default fields include `Html`, `Component`, and `Url` (as well as the `StatsService` that we injected). If for instance you wanted to replace the default HTML Helpers with your own, you could easily do so using `@inject`:

```
@using System.Threading.Tasks
@using ViewInjectSample.Helpers
@inject MyHtmlHelper Html
<!DOCTYPE html>
<html>
<head>
    <title>My Helper</title>
</head>
<body>
    <div>
        Test: @Html.Value
    </div>
</body>
</html>
```

If you want to extend existing services, you can simply use this technique while inheriting from or wrapping the existing implementation with your own.

See Also

- Simon Timms Blog: Getting Lookup Data Into Your View

Unit test controller logic in ASP.NET Core

9/22/2020 • 25 minutes to read • Edit Online

By Steve Smith

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Unit testing controllers

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app.

View or download sample code (how to download)

The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new StormSessionViewModel()
        {
            Id = session.Id,
            DateCreated = session.DateCreated,
            Name = session.Name,
            IdeaCount = session.Ideas.Count
        });

        return View(model);
    }

    public class NewSessionModel
    {
        [Required]
        public string SessionName { get; set; }
    }

    [HttpPost]
    public async Task<IActionResult> Index(NewSessionModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        else
        {
            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });
        }

        return RedirectToAction(actionName: nameof(Index));
    }
}
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.

- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
  - The `Add` method on the repository is called.
  - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

```
[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> **NOTE**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (*Controllers/SessionController.cs*):

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return Ok(result);
}

[HttpPost("create")]
public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);
    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return Ok(session);
}
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).
- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState`:

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null`, so the mock repository is configured to

return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

In ASP.NET Core 2.1 or later, ActionResult<T> (ActionResult<TValue>) enables you to return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

```
[Fact]
public async Task ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>> CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id }, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

```
[Fact]
public async Task CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnValue.Ideas.LastOrDefault().Description);
}
```

Controllers play a central role in any ASP.NET Core MVC app. As such, you should have confidence that controllers behave as intended. Automated tests can detect errors before the app is deployed to a production environment.

View or download sample code (how to download)

Unit tests of controller logic

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app. The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new StormSessionViewModel()
        {
            Id = session.Id,
            DateCreated = session.DateCreated,
            Name = session.Name,
            IdeaCount = session.Ideas.Count
        });

        return View(model);
    }

    public class NewSessionModel
    {
        [Required]
        public string SessionName { get; set; }
    }

    [HttpPost]
    public async Task<IActionResult> Index(NewSessionModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        else
        {
            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });
        }

        return RedirectToAction(actionName: nameof(Index));
    }
}
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.

- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
  - The `Add` method on the repository is called.
  - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

```
[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> **NOTE**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (*Controllers/SessionController.cs*):

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return Ok(result);
}

[HttpPost("create")]
public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);
    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return Ok(session);
}
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).
- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState`:

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null`, so the mock repository is configured to

return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

In ASP.NET Core 2.1 or later, ActionResult<T> (ActionResult<TValue>) enables you to return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

```
[Fact]
public async Task ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>> CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id }, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

```
[Fact]
public async Task CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnValue.Ideas.LastOrDefault().Description);
}
```

Additional resources

- Integration tests in ASP.NET Core
- Create and run unit tests with Visual Studio
- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)
- JustMockLite: A mocking framework for .NET developers. (*Not maintained or supported by Microsoft.*)

Introduction to ASP.NET Core Blazor

9/22/2020 • 5 minutes to read • Edit Online

By Daniel Roth and Luke Latham

Welcome to Blazor!

Blazor is a framework for building interactive client-side web UI with .NET:

- Create rich interactive UIs using C# instead of JavaScript.
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.
- Integrate with modern hosting platforms, such as Docker.

Using .NET for client-side web development offers the following advantages:

- Write code in C# instead of JavaScript.
- Leverage the existing .NET ecosystem of .NET libraries.
- Share app logic across server and client.
- Benefit from .NET's performance, reliability, and security.
- Stay productive with Visual Studio on Windows, Linux, and macOS.
- Build on a common set of languages, frameworks, and tools that are stable, feature-rich, and easy to use.

Components

Blazor apps are based on *components*. A component in Blazor is an element of UI, such as a page, dialog, or data entry form.

Components are .NET classes built into .NET assemblies that:

- Define flexible UI rendering logic.
- Handle user events.
- Can be nested and reused.
- Can be shared and distributed as Razor class libraries or NuGet packages.

The component class is usually written in the form of a Razor markup page with a `.razor` file extension. Components in Blazor are formally referred to as *Razor components*. Razor is a syntax for combining HTML markup with C# code designed for developer productivity. Razor allows you to switch between HTML markup and C# in the same file with IntelliSense support. Razor Pages and MVC also use Razor. Unlike Razor Pages and MVC, which are built around a request/response model, components are used specifically for client-side UI logic and composition.

The following Razor markup demonstrates a component (`Dialog.razor`), which can be nested within another component:

```
<div>
    <h1>@Title</h1>

    @ChildContent

    <button @onclick="OnYes">Yes!</button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    private void OnYes()
    {
        Console.WriteLine("Write to the console in C#! 'Yes' button was selected.");
    }
}
```

The dialog's body content (`ChildContent`) and title (`Title`) are provided by the component that uses this component in its UI. `OnYes` is a C# method triggered by the button's `onclick` event.

Blazor uses natural HTML tags for UI composition. HTML elements specify components, and a tag's attributes pass values to a component's properties.

In the following example, the `Index` component uses the `Dialog` component. `ChildContent` and `Title` are set by the attributes and content of the `<Dialog>` element.

`Pages/Index.razor`:

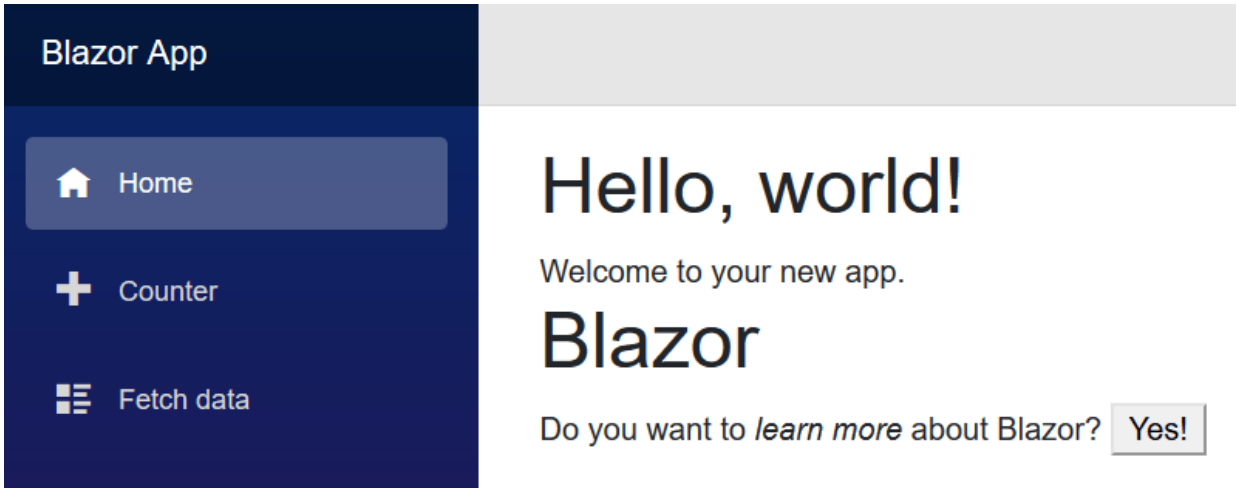
```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<Dialog Title="Blazor">
    Do you want to <i>learn more</i> about Blazor?
</Dialog>
```

The dialog is rendered when the parent (`Pages/Index.razor`) is accessed in a browser:



When this component is used in the app, IntelliSense in Visual Studio and Visual Studio Code speeds development with syntax and parameter completion.

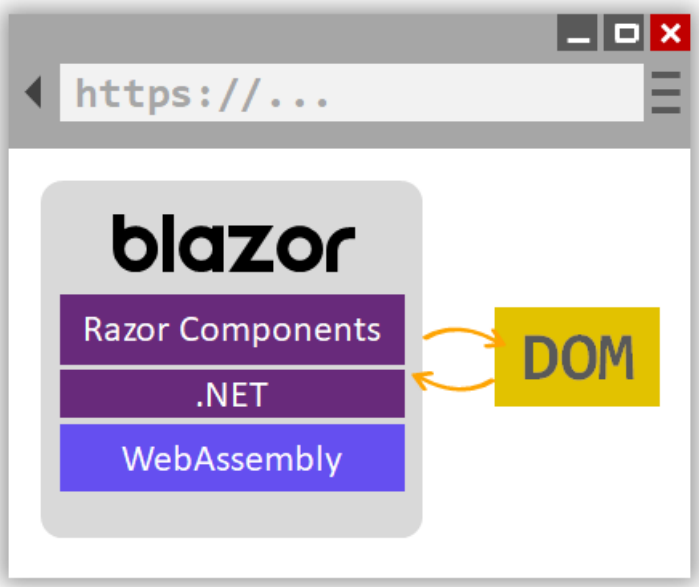
Components render into an in-memory representation of the browser's Document Object Model (DOM) called a *render tree*, which is used to update the UI in a flexible and efficient way.

Blazor WebAssembly

Blazor WebAssembly is a single-page app framework for building interactive client-side web apps with .NET. Blazor WebAssembly uses open web standards without plugins or code transpilation and works in all modern web browsers, including mobile browsers.

Running .NET code inside web browsers is made possible by WebAssembly (abbreviated `wasm`). WebAssembly is a compact bytecode format optimized for fast download and maximum execution speed. WebAssembly is an open web standard and supported in web browsers without plugins.

WebAssembly code can access the full functionality of the browser via JavaScript, called *JavaScript interoperability* (or *JavaScript interop*). .NET code executed via WebAssembly in the browser runs in the browser's JavaScript sandbox with the protections that the sandbox provides against malicious actions on the client machine.



When a Blazor WebAssembly app is built and run in a browser:

- C# code files and Razor files are compiled into .NET assemblies.
- The assemblies and the .NET runtime are downloaded to the browser.
- Blazor WebAssembly bootstraps the .NET runtime and configures the runtime to load the assemblies for the app. The Blazor WebAssembly runtime uses JavaScript interop to handle DOM manipulation and browser API calls.

The size of the published app, its *payload size*, is a critical performance factor for an app's useability. A large app takes a relatively long time to download to a browser, which diminishes the user experience. Blazor WebAssembly optimizes payload size to reduce download times:

- Unused code is stripped out of the app when it's published by the Intermediate Language (IL) Trimmer.
- HTTP responses are compressed.
- The .NET runtime and assemblies are cached in the browser.

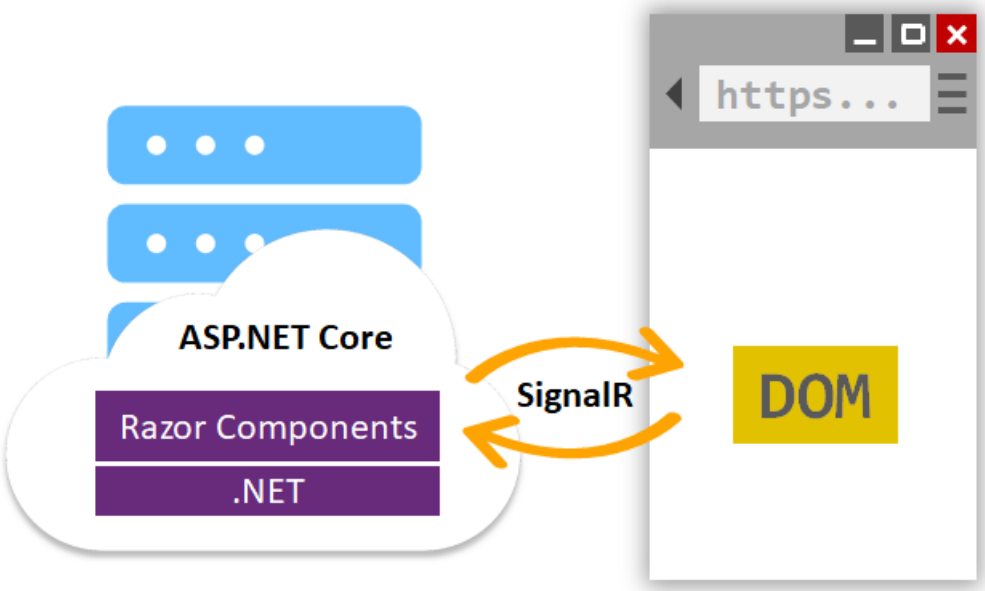
- Unused code is stripped out of the app when it's published by the Intermediate Language (IL) Linker.
- HTTP responses are compressed.
- The .NET runtime and assemblies are cached in the browser.

Blazor Server

Blazor decouples component rendering logic from how UI updates are applied. Blazor Server provides support for hosting Razor components on the server in an ASP.NET Core app. UI updates are handled over a SignalR connection.

The runtime handles sending UI events from the browser to the server and applies UI updates sent by the server back to the browser after running the components.

The connection used by Blazor Server to communicate with the browser is also used to handle JavaScript interop calls.



JavaScript interop

For apps that require third-party JavaScript libraries and access to browser APIs, components interoperate with JavaScript. Components are capable of using any library or API that JavaScript is able to use. C# code can call into JavaScript code, and JavaScript code can call into C# code. For more information, see the following articles:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Code sharing and .NET Standard

Blazor implements .NET Standard 2.1, which enables Blazor projects to reference libraries that conform to .NET Standard 2.1 or earlier specifications. .NET Standard is a formal specification of .NET APIs that are common across .NET implementations. .NET Standard class libraries can be shared across different .NET platforms, such as Blazor, .NET Framework, .NET Core, Xamarin, Mono, and Unity.

APIs that aren't applicable inside of a web browser (for example, accessing the file system, opening a socket, and threading) throw a PlatformNotSupportedException.

Additional resources

- WebAssembly
- ASP.NET Core Blazor hosting models
- Use ASP.NET Core SignalR with Blazor WebAssembly
- C# Guide

- razor syntax reference for ASP.NET Core
- HTML
- Awesome Blazor community links

ASP.NET Core Blazor supported platforms

9/22/2020 • 2 minutes to read • Edit Online

By Luke Latham

Browser requirements

Blazor WebAssembly

BROWSER	VERSION
Microsoft Edge	Current
Mozilla Firefox	Current
Google Chrome, including Android	Current
Safari, including iOS	Current
Microsoft Internet Explorer	Not Supported†

†Microsoft Internet Explorer doesn't support WebAssembly.

Blazor Server

BROWSER	VERSION
Microsoft Edge	Current
Mozilla Firefox	Current
Google Chrome, including Android	Current
Safari, including iOS	Current
Microsoft Internet Explorer	11†

†Additional polyfills are required (for example, promises can be added via a `Polyfill.io` bundle).

Additional resources

- ASP.NET Core Blazor hosting models

Tooling for ASP.NET Core Blazor

9/22/2020 • 2 minutes to read • Edit Online

By Daniel Roth and Luke Latham

1. Install the latest version of Visual Studio 2019 with the **ASP.NET and web development** workload.
2. Create a new project.
3. Select **Blazor App**. Select **Next**.
4. Provide a project name in the **Project name** field or accept the default project name. Confirm the **Location** entry is correct or provide a location for the project. Select **Create**.
5. For a Blazor WebAssembly experience, choose the **Blazor WebAssembly App** template. For a Blazor Server experience, choose the **Blazor Server App** template. Select **Create**.

   For information on the two Blazor hosting models, *Blazor WebAssembly* and *Blazor Server*, see ASP.NET Core Blazor hosting models.
6. Press `Ctrl+F5` to run the app.

For more information on trusting the ASP.NET Core HTTPS development certificate, see Enforce HTTPS in ASP.NET Core.

1. Install the latest version of the .NET Core 3.1 SDK. If you previously installed the SDK, you can determine your installed version by executing the following command in a command shell:

   ```
   dotnet --version
   ```
2. Install the latest version of Visual Studio Code.
3. Install the latest C# for Visual Studio Code extension.
4. For a Blazor WebAssembly experience, execute the following command in a command shell:

   ```
   dotnet new blazorwasm -o WebApplication1
   ```

   For a Blazor Server experience, execute the following command in a command shell:

   ```
   dotnet new blazorserver -o WebApplication1
   ```

   For information on the two Blazor hosting models, *Blazor WebAssembly* and *Blazor Server*, see ASP.NET Core Blazor hosting models.
5. Open the `WebApplication1` folder in Visual Studio Code.
6. The IDE requests that you add assets to build and debug the project. Select **Yes**.
7. Press `Ctrl+F5` to run the app.

Trust a development certificate

There's no centralized way to trust a certificate on Linux. Typically, one of the following approaches is adopted:

- Exclude the app's URL in browser's exclude list.
- Trust all self-signed certificates for `localhost`.
- Add the certificate to the list of trusted certificates in the browser.

For more information, see the guidance provided by your browser and Linux distribution.

1. Install Visual Studio for Mac.
2. Select **File** > **New Solution** or create a **New** project from the **Start Window**.
3. In the sidebar, select **Web and Console** > **App**.

   For a Blazor WebAssembly experience, choose the **Blazor WebAssembly App** template. For a Blazor Server experience, choose the **Blazor Server App** template. Select **Next**.

   For information on the two Blazor hosting models, *Blazor WebAssembly* and *Blazor Server*, see ASP.NET Core Blazor hosting models.
4. Confirm that **Authentication** is set to **No Authentication**. Select **Next**.
5. In the **Project Name** field, name the app `WebApplication1`. Select **Create**.
6. Select **Run** > **Start Without Debugging** to run the app *without the debugger*. Run the app with **Run** > **Start Debugging** or the Run (▶) button to run the app *with the debugger*.

If a prompt appears to trust the development certificate, trust the certificate and continue. The user and keychain passwords are required to trust the certificate. For more information on trusting the ASP.NET Core HTTPS development certificate, see Enforce HTTPS in ASP.NET Core.

ASP.NET Core Blazor hosting models

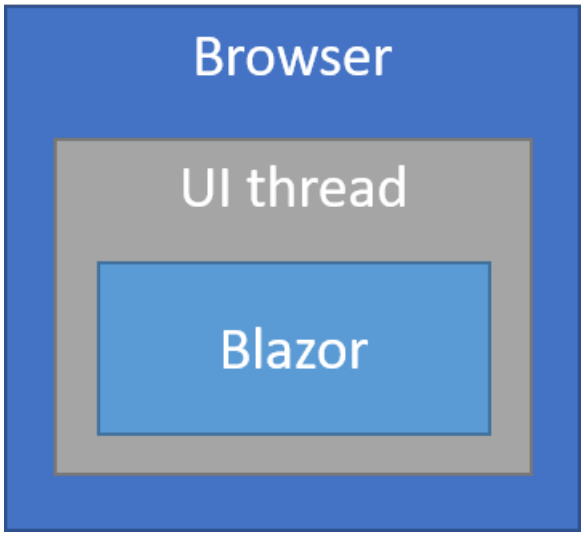
9/22/2020 • 9 minutes to read • Edit Online

By Daniel Roth

Blazor is a web framework designed to run client-side in the browser on a WebAssembly-based .NET runtime (*Blazor WebAssembly*) or server-side in ASP.NET Core (*Blazor Server*). Regardless of the hosting model, the app and component models *are the same*.

Blazor WebAssembly

The principal hosting model for Blazor is running client-side in the browser on WebAssembly. The Blazor app, its dependencies, and the .NET runtime are downloaded to the browser. The app is executed directly on the browser UI thread. UI updates and event handling occur within the same process. The app's assets are deployed as static files to a web server or service capable of serving static content to clients. Because the app is created for deployment without a backend ASP.NET Core app, it's called a *standalone Blazor WebAssembly app*.



To create a Blazor app using the client-side hosting model, use the **Blazor WebAssembly App** template (`dotnet new blazorwasm`).

After selecting the **Blazor WebAssembly App** template, you have the option of configuring the app to use an ASP.NET Core backend by selecting the **ASP.NET Core hosted** check box (`dotnet new blazorwasm --hosted`). The ASP.NET Core app serves the Blazor app to clients. An app with an ASP.NET Core backend is called a *hosted Blazor WebAssembly app*. The Blazor WebAssembly app can interact with the server over the network using web API calls or SignalR (Use ASP.NET Core SignalR with Blazor WebAssembly).

The `blazor.webassembly.js` script is provided by the framework and handles:

- Downloading the .NET runtime, the app, and the app's dependencies.
- Initialization of the runtime to run the app.

The Blazor WebAssembly hosting model offers several benefits:

- There's no .NET server-side dependency. The app is fully functioning after it's downloaded to the client.
- Client resources and capabilities are fully leveraged.
- Work is offloaded from the server to the client.
- An ASP.NET Core web server isn't required to host the app. Serverless deployment scenarios are possible (for

example, serving the app from a CDN).

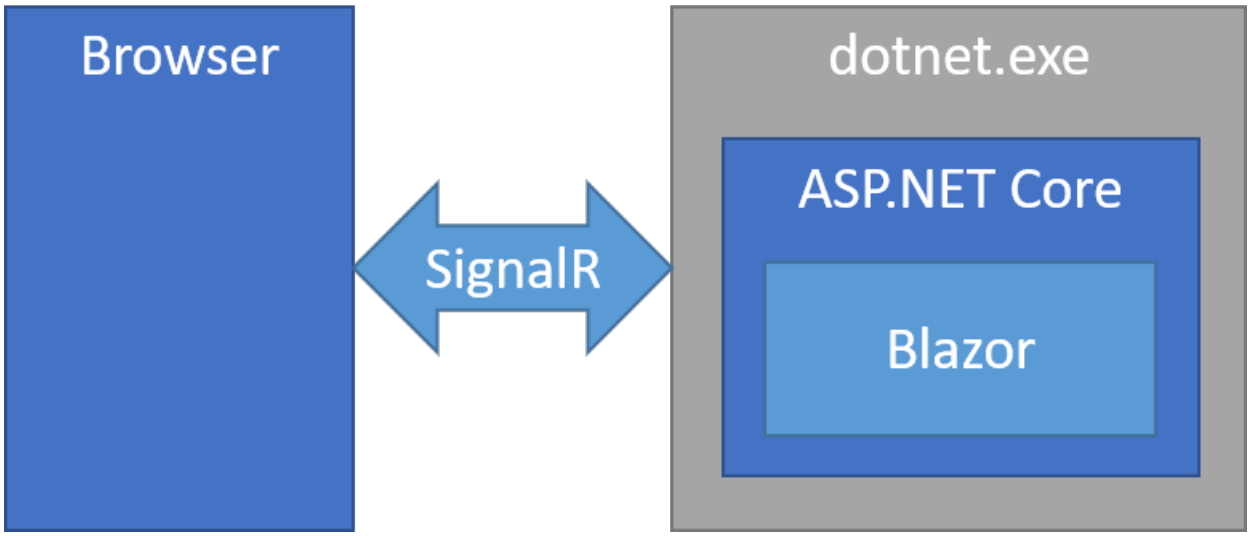
There are downsides to Blazor WebAssembly hosting:

- The app is restricted to the capabilities of the browser.
- Capable client hardware and software (for example, WebAssembly support) is required.
- Download size is larger, and apps take longer to load.
- .NET runtime and tooling support is less mature. For example, limitations exist in .NET Standard support and debugging.

The hosted Blazor app model supports Docker containers. Right-click on the Server project in Visual Studio and select **Add** > **Docker Support**.

Blazor Server

With the Blazor Server hosting model, the app is executed on the server from within an ASP.NET Core app. UI updates, event handling, and JavaScript calls are handled over a SignalR connection.



To create a Blazor app using the Blazor Server hosting model, use the ASP.NET Core **Blazor Server App** template (`dotnet new blazorserver`). The ASP.NET Core app hosts the Blazor Server app and creates the SignalR endpoint where clients connect.

The ASP.NET Core app references the app's `Startup` class to add:

- Server-side services.
- The app to the request handling pipeline.

On the client, the `blazor.server.js` script establishes the SignalR connection with the server. The script is served to the client-side app from an embedded resource in the ASP.NET Core shared framework. The client-side app is responsible for persisting and restoring app state as required.

The Blazor Server hosting model offers several benefits:

- Download size is significantly smaller than a Blazor WebAssembly app, and the app loads much faster.
- The app takes full advantage of server capabilities, including use of any .NET Core compatible APIs.
- .NET Core on the server is used to run the app, so existing .NET tooling, such as debugging, works as expected.
- Thin clients are supported. For example, Blazor Server apps work with browsers that don't support WebAssembly and on resource-constrained devices.
- The app's .NET/C# code base, including the app's component code, isn't served to clients.

> **IMPORTANT**
>
> A Blazor Server app prerenders in response to the first client request, which sets up the UI state on the server. When the client attempts to create a SignalR connection, **the client must reconnect to the same server**. Blazor Server apps that use more than one backend server should implement *sticky sessions* for SignalR connections. For more information, see the Connection to the server section.

There are downsides to Blazor Server hosting:

- Higher latency usually exists. Every user interaction involves a network hop.
- There's no offline support. If the client connection fails, the app stops working.
- Scalability is challenging for apps with many users. The server must manage multiple client connections and handle client state.
- An ASP.NET Core server is required to serve the app. Serverless deployment scenarios aren't possible (for example, serving the app from a CDN).

The Blazor Server app model supports Docker containers. Right-click on the project in Visual Studio and select **Add** > **Docker Support**.

Comparison to server-rendered UI

One way to understand Blazor Server apps is to understand how it differs from traditional models for rendering UI in ASP.NET Core apps using Razor views or Razor Pages. Both models use the Razor language to describe HTML content, but they significantly differ in how markup is rendered.

When a Razor Page or view is rendered, every line of Razor code emits HTML in text form. After rendering, the server disposes of the page or view instance, including any state that was produced. When another request for the page occurs, for instance when server validation fails and the validation summary is displayed:

- The entire page is rerendered to HTML text again.
- The page is sent to the client.

A Blazor app is composed of reusable elements of UI called *components*. A component contains C# code, markup, and other components. When a component is rendered, Blazor produces a graph of the included components similar to an HTML or XML Document Object Model (DOM). This graph includes component state held in properties and fields. Blazor evaluates the component graph to produce a binary representation of the markup. The binary format can be:

- Turned into HTML text (during prerendering†).
- Used to efficiently update the markup during regular rendering.

†*Prerendering*: The requested Razor component is compiled on the server into static HTML and sent to the client, where it's rendered to the user. After the connection is made between the client and the server, the component's static prerendered elements are replaced with interactive elements. Prerendering makes the app feel more responsive to the user.

A UI update in Blazor is triggered by:

- User interaction, such as selecting a button.
- App triggers, such as a timer.

The graph is rerendered, and a UI *diff* (difference) is calculated. This diff is the smallest set of DOM edits required to update the UI on the client. The diff is sent to the client in a binary format and applied by the browser.

A component is disposed after the user navigates away from it on the client. While a user is interacting with a component, the component's state (services, resources) must be held in the server's memory. Because the state of many components might be maintained by the server concurrently, memory exhaustion is a concern that must be

addressed. For guidance on how to author a Blazor Server app to ensure the best use of server memory, see Threat mitigation guidance for ASP.NET Core Blazor Server.

Circuits

A Blazor Server app is built on top of ASP.NET Core SignalR. Each client communicates to the server over one or more SignalR connections called a *circuit*. A circuit is Blazor's abstraction over SignalR connections that can tolerate temporary network interruptions. When a Blazor client sees that the SignalR connection is disconnected, it attempts to reconnect to the server using a new SignalR connection.

Each browser screen (browser tab or iframe) that is connected to a Blazor Server app uses a SignalR connection. This is yet another important distinction compared to typical server-rendered apps. In a server-rendered app, opening the same app in multiple browser screens typically doesn't translate into additional resource demands on the server. In a Blazor Server app, each browser screen requires a separate circuit and separate instances of component state to be managed by the server.

Blazor considers closing a browser tab or navigating to an external URL a *graceful* termination. In the event of a graceful termination, the circuit and associated resources are immediately released. A client may also disconnect non-gracefully, for instance due to a network interruption. Blazor Server stores disconnected circuits for a configurable interval to allow the client to reconnect.

Blazor Server allows code to define a *circuit handler*, which allows running code on changes to the state of a user's circuit. For more information, see ASP.NET Core Blazor advanced scenarios.

UI Latency

UI latency is the time it takes from an initiated action to the time the UI is updated. Smaller values for UI latency are imperative for an app to feel responsive to a user. In a Blazor Server app, each action is sent to the server, processed, and a UI diff is sent back. Consequently, UI latency is the sum of network latency and the server latency in processing the action.

For a line of business app that's limited to a private corporate network, the effect on user perceptions of latency due to network latency are usually imperceptible. For an app deployed over the Internet, latency may become noticeable to users, particularly if users are widely distributed geographically.

Memory usage can also contribute to app latency. Increased memory usage results in frequent garbage collection or paging memory to disk, both of which degrade app performance and consequently increase UI latency.

Blazor Server apps should be optimized to minimize UI latency by reducing network latency and memory usage. For an approach to measuring network latency, see Host and deploy ASP.NET Core Blazor Server. For more information on SignalR and Blazor, see:

- Host and deploy ASP.NET Core Blazor Server
- Threat mitigation guidance for ASP.NET Core Blazor Server

Connection to the server

Blazor Server apps require an active SignalR connection to the server. If the connection is lost, the app attempts to reconnect to the server. As long as the client's state is still in memory, the client session resumes without losing state.

A Blazor Server app prerenders in response to the first client request, which sets up the UI state on the server. When the client attempts to create a SignalR connection, the client must reconnect to the same server. Blazor Server apps that use more than one backend server should implement *sticky sessions* for SignalR connections.

We recommend using the Azure SignalR Service for Blazor Server apps. The service allows for scaling up a Blazor Server app to a large number of concurrent SignalR connections. Sticky sessions are enabled for the Azure SignalR Service by setting the service's `ServerStickyMode` option or configuration value to `Required`. For more information, see Host and deploy ASP.NET Core Blazor Server.

When using IIS, sticky sessions are enabled with Application Request Routing. For more information, see HTTP Load Balancing using Application Request Routing.

Additional resources

- Introduction to ASP.NET Core SignalR
- ASP.NET Core Blazor hosting model configuration
- Use ASP.NET Core SignalR with Blazor WebAssembly

Build a Blazor todo list app

9/22/2020 • 4 minutes to read • Edit Online

By Daniel Roth and Luke Latham

This tutorial shows you how to build and modify a Blazor app. You learn how to:

- Create a todo list Blazor app project
- Modify Razor components
- Use event handling and data binding in components
- Use routing in a Blazor app

At the end of this tutorial, you'll have a working todo list app.

Prerequisites

.NET Core 3.1 SDK or later

Create a todo list Blazor app

1. Create a new Blazor app named `TodoList` in a command shell:

   ```
   dotnet new blazorserver -o TodoList
   ```

   The preceding command creates a folder named `TodoList` to hold the app. The `TodoList` folder is the *root folder* of the project. Change directories to the `TodoList` folder with the following command:

   ```
   cd TodoList
   ```

2. Add a new `Todo` Razor component to the app in the `Pages` folder using the following command:

   ```
   dotnet new razorcomponent -n Todo -o Pages
   ```

   IMPORTANT

   Razor component file names require a capitalized first letter. Open the `Pages` folder and confirm that the `Todo` component file name starts with a capital letter `T`. The file name should be `Todo.razor`.

3. In `Pages/Todo.razor` provide the initial markup for the component:

   ```
   @page "/todo"

   <h3>Todo</h3>
   ```

4. Add the `Todo` component to the navigation bar.

   The `NavMenu` component (`Shared/NavMenu.razor`) is used in the app's layout. Layouts are components that allow you to avoid duplication of content in the app.

Add a `<NavLink>` element for the `Todo` component by adding the following list item markup below the existing list items in the `Shared/NavMenu.razor` file:

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="todo">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Todo
    </NavLink>
</li>
```

5. Build and run the app by executing the `dotnet run` command in the command shell from the `TodoList` folder. Visit the new Todo page to confirm that the link to the `Todo` component works.

6. Add a `TodoItem.cs` file to the root of the project (the `TodoList` folder) to hold a class that represents a todo item. Use the following C# code for the `TodoItem` class:

```
public class TodoItem
{
    public string Title { get; set; }
    public bool IsDone { get; set; }
}
```

7. Return to the `Todo` component (`Pages/Todo.razor`):

   - Add a field for the todo items in an `@code` block. The `Todo` component uses this field to maintain the state of the todo list.
   - Add unordered list markup and a `foreach` loop to render each todo item as a list item (`<li>`).

```
@page "/todo"

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
}
```

8. The app requires UI elements for adding todo items to the list. Add a text input (`<input>`) and a button (`<button>`) below the unordered list (`<ul>...</ul>`):

```
@page "/todo"

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" />
<button>Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
}
```

9. Stop the running app in the command shell. Many command shells accept the keyboard command `Ctrl+c` to stop an app. Rebuild and run the app with the `dotnet run` command. When the `Add todo` button is selected, nothing happens because an event handler isn't wired up to the button.

10. Add an `AddTodo` method to the `Todo` component and register it for button selections using the `@onclick` attribute. The `AddTodo` C# method is called when the button is selected:

```
<input placeholder="Something todo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();

    private void AddTodo()
    {
        // Todo: Add the todo
    }
}
```

11. To get the title of the new todo item, add a `newTodo` string field at the top of the `@code` block and bind it to the value of the text input using the `bind` attribute in the `<input>` element:

```
private IList<TodoItem> todos = new List<TodoItem>();
private string newTodo;
```

```
<input placeholder="Something todo" @bind="newTodo" />
```

12. Update the `AddTodo` method to add the `TodoItem` with the specified title to the list. Clear the value of the text input by setting `newTodo` to an empty string:

```
@page "/todo"

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
    private string newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

13. Stop the running app in the command shell. Rebuild and run the app with the `dotnet run` command. Add some todo items to the todo list to test the new code.

14. The title text for each todo item can be made editable, and a check box can help the user keep track of completed items. Add a check box input for each todo item and bind its value to the `IsDone` property. Change `@todo.Title` to an `<input>` element bound to `@todo.Title`:

```
<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
        </li>
    }
</ul>
```

15. To verify that these values are bound, update the `<h3>` header to show a count of the number of todo items that aren't complete (`IsDone` is `false`).

```
<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>
```

16. The completed `Todo` component (`Pages/Todo.razor`):

```
@page "/todo"

<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
        </li>
    }
</ul>

<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private IList<TodoItem> todos = new List<TodoItem>();
    private string newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

17. Stop the running app in the command shell. Rebuild and run the app with the `dotnet run` command. Add todo items to test the new code.

Next steps

In this tutorial, you learned how to:

- Create a todo list Blazor app project
- Modify Razor components
- Use event handling and data binding in components
- Use routing in a Blazor app

Learn about tooling for ASP.NET Core Blazor:

Tooling for ASP.NET Core Blazor

Use ASP.NET Core SignalR with Blazor WebAssembly

9/22/2020 • 7 minutes to read • Edit Online

By Daniel Roth and Luke Latham

This tutorial teaches the basics of building a real-time app using SignalR with Blazor WebAssembly. You learn how to:

- Create a Blazor WebAssembly Hosted app project
- Add the SignalR client library
- Add a SignalR hub
- Add SignalR services and an endpoint for the SignalR hub
- Add Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

- Visual Studio 2019 16.6 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a hosted Blazor WebAssembly app project

Follow the guidance for your choice of tooling:

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

> **NOTE**
> Visual Studio 16.6 or later and .NET Core SDK 3.1.300 or later are required.

1. Create a new project.
2. Select **Blazor App** and select **Next**.
3. Type `BlazorSignalRApp` in the **Project name** field. Confirm the **Location** entry is correct or provide a location for the project. Select **Create**.
4. Choose the **Blazor WebAssembly App** template.
5. Under **Advanced**, select the **ASP.NET Core hosted** check box.

6. Select **Create**.

Add the SignalR client library

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

1. In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.
2. In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.
3. With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.
4. In the search results, select the `Microsoft.AspNetCore.SignalR.Client` package and select **Install**.
5. If the **Preview Changes** dialog appears, select **OK**.
6. If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the `BlazorSignalRApp.Server` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Server.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

Add services and an endpoint for the SignalR hub

1. In the `BlazorSignalRApp.Server` project, open the `Startup.cs` file.
2. Add the namespace for the `ChatHub` class to the top of the file:

   ```
   using BlazorSignalRApp.Server.Hubs;
   ```

3. Add SignalR and Response Compression Middleware services to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR();
    services.AddControllersWithViews();
    services.AddResponseCompression(opts =>
    {
        opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "application/octet-stream" });
    });
}
```

4. In `Startup.Configure`:

   - Use Response Compression Middleware at the top of the processing pipeline's configuration.
   - Between the endpoints for controllers and the client-side fallback, add an endpoint for the hub.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseResponseCompression();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseWebAssemblyDebugging();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseBlazorFrameworkFiles();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapHub<ChatHub>("/chathub");
        endpoints.MapFallbackToFile("index.html");
    });
}
```

Add Razor component code for chat

1. In the `BlazorSignalRApp.Client` project, open the `Pages/Index.razor` file.

2. Replace the markup with the following code:

```
@page "/"
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager NavigationManager
@implements IDisposable

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection hubConnection;
    private List<string> messages = new List<string>();
    private string userInput;
    private string messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(NavigationManager.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message) =>
        {
            var encodedMsg = $"{user}: {message}";
            messages.Add(encodedMsg);
            StateHasChanged();
        });

        await hubConnection.StartAsync();
    }

    Task Send() =>
        hubConnection.SendAsync("SendMessage", userInput, messageInput);

    public bool IsConnected =>
        hubConnection.State == HubConnectionState.Connected;

    public void Dispose()
    {
        _ = hubConnection.DisposeAsync();
    }
}
```

Run the app

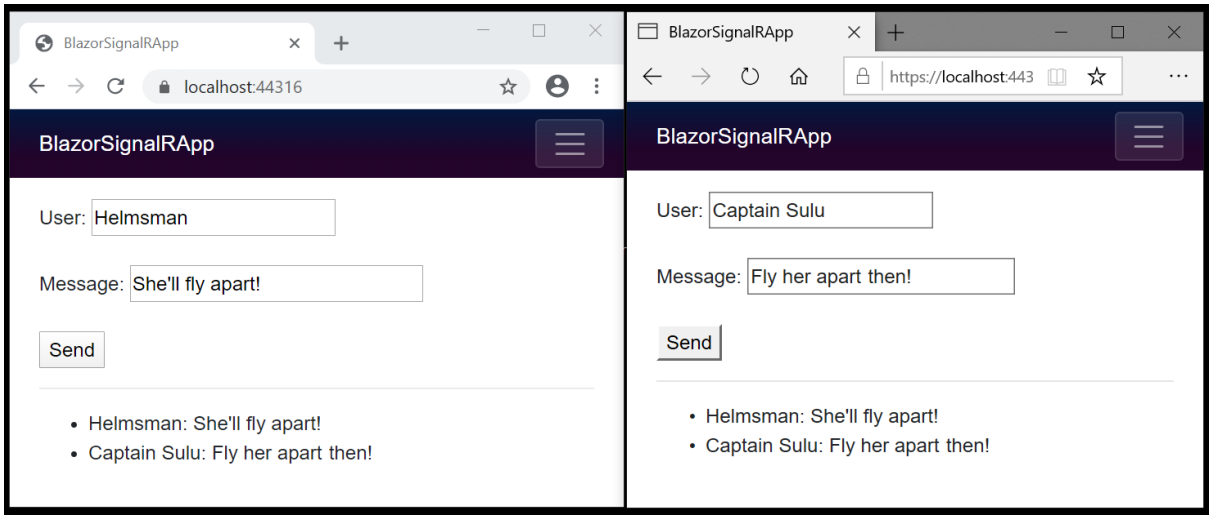
1. Follow the guidance for your tooling:

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

1. In **Solution Explorer**, select the `BlazorSignalRApp.Server` project. Press `F5` to run the app with debugging or `Ctrl+F5` to run the app without debugging.

2. Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

3. Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* © 1991 Paramount

Next steps

In this tutorial, you learned how to:

- Create a Blazor WebAssembly Hosted app project
- Add the SignalR client library
- Add a SignalR hub
- Add SignalR services and an endpoint for the SignalR hub
- Add Razor component code for chat

To learn more about building Blazor apps, see the Blazor documentation:

Introduction to ASP.NET Core Blazor

Additional resources

- Introduction to ASP.NET Core SignalR
- SignalR cross-origin negotiation for authentication

ASP.NET Core Blazor templates

9/22/2020 • 3 minutes to read • Edit Online

By Daniel Roth and Luke Latham

The Blazor framework provides templates to develop apps for each of the Blazor hosting models:

- Blazor WebAssembly (`blazorwasm`)
- Blazor Server (`blazorserver`)

For more information on Blazor's hosting models, see ASP.NET Core Blazor hosting models.

Template options are available by passing the `--help` option to the `dotnet new` CLI command:

```
dotnet new blazorwasm --help
dotnet new blazorserver --help
```

Blazor project structure

The following files and folders make up a Blazor app generated from a Blazor project template:

- `Program.cs` : The app's entry point that sets up the:
  - ASP.NET Core host (Blazor Server)
  - WebAssembly host (Blazor WebAssembly): The code in this file is unique to apps created from the Blazor WebAssembly template (`blazorwasm`).
    - The `App` component is the root component of the app. The `App` component is specified as the `app` DOM element (`<app>...</app>`) to the root component collection (`builder.RootComponents.Add<App>("app")`).
    - Services are added and configured (for example, `builder.Services.AddSingleton<IMyDependency, MyDependency>()`).
- `Startup.cs` (Blazor Server): Contains the app's startup logic. The `Startup` class defines two methods:
  - `ConfigureServices` : Configures the app's dependency injection (DI) services. In Blazor Server apps, services are added by calling AddServerSideBlazor, and the `WeatherForecastService` is added to the service container for use by the example `FetchData` component.
  - `Configure` : Configures the app's request handling pipeline:
    - MapBlazorHub is called to set up an endpoint for the real-time connection with the browser. The connection is created with SignalR, which is a framework for adding real-time web functionality to apps.
    - `MapFallbackToPage("/_Host")` is called to set up the root page of the app (`Pages/_Host.cshtml`) and enable navigation.
- `wwwroot/index.html` (Blazor WebAssembly): The root page of the app implemented as an HTML page:
  - When any page of the app is initially requested, this page is rendered and returned in the response.
  - The page specifies where the root `App` component is rendered. The component is rendered at the location of the `app` DOM element (`<app>...</app>`).
  - The `_framework/blazor.webassembly.js` JavaScript file is loaded, which:
    - Downloads the .NET runtime, the app, and the app's dependencies.

    - Initializes the runtime to run the app.
- `App.razor`: The root component of the app that sets up client-side routing using the Router component. The Router component intercepts browser navigation and renders the page that matches the requested address.
- `Pages` folder: Contains the routable components/pages (`.razor`) that make up the Blazor app and the root Razor page of a Blazor Server app. The route for each page is specified using the `@page` directive. The template includes the following:
    - `_Host.cshtml` (Blazor Server): The root page of the app implemented as a Razor Page:
        - When any page of the app is initially requested, this page is rendered and returned in the response.
        - The `_framework/blazor.server.js` JavaScript file is loaded, which sets up the real-time SignalR connection between the browser and the server.
        - The Host page specifies where the root `App` component (`App.razor`) is rendered.
    - `Counter` (`Pages/Counter.razor`): Implements the Counter page.
    - `Error` (`Error.razor`, Blazor Server app only): Rendered when an unhandled exception occurs in the app.
    - `FetchData` (`Pages/FetchData.razor`): Implements the Fetch data page.
    - `Index` (`Pages/Index.razor`): Implements the Home page.
- `Properties/launchSettings.json`: Holds development environment configuration.
- `Shared` folder: Contains other UI components (`.razor`) used by the app:
    - `MainLayout` (`MainLayout.razor`): The app's layout component.
    - `NavMenu` (`NavMenu.razor`): Implements sidebar navigation. Includes the `NavLink` component (NavLink), which renders navigation links to other Razor components. The NavLink component automatically indicates a selected state when its component is loaded, which helps the user understand which component is currently displayed.
- `_Imports.razor`: Includes common Razor directives to include in the app's components (`.razor`), such as `@using` directives for namespaces.
- `Data` folder (Blazor Server): Contains the `WeatherForecast` class and implementation of the `WeatherForecastService` that provide example weather data to the app's `FetchData` component.
- `wwwroot`: The Web Root folder for the app containing the app's public static assets.
- `appsettings.json`: Holds configuration settings for the app. In a Blazor WebAssembly app, the app settings file is located in the `wwwroot` folder. In a Blazor Server app, the app settings file is located at the project root.

ASP.NET Core Blazor routing

9/22/2020 • 9 minutes to read • Edit Online

By Luke Latham

Learn how to route requests and how to use the NavLink component to create navigation links in Blazor apps.

ASP.NET Core endpoint routing integration

Blazor Server is integrated into ASP.NET Core Endpoint Routing. An ASP.NET Core app is configured to accept incoming connections for interactive components with MapBlazorHub in `Startup.Configure`:

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

The most typical configuration is to route all requests to a Razor page, which acts as the host for the server-side part of the Blazor Server app. By convention, the *host* page is usually named `_Host.cshtml`. The route specified in the host file is called a *fallback route* because it operates with a low priority in route matching. The fallback route is considered when other routes don't match. This allows the app to use others controllers and pages without interfering with the Blazor Server app.

For information on configuring MapFallbackToPage for non-root URL server hosting, see Host and deploy ASP.NET Core Blazor.

Route templates

The Router component enables routing to each component with a specified route. The Router component appears in the `App.razor` file:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

When a `.razor` file with an `@page` directive is compiled, the generated class is provided a RouteAttribute specifying the route template.

At runtime, the RouteView component:

- Receives the RouteData from the Router along with any desired parameters.
- Renders the specified component with its layout (or an optional default layout) using the specified parameters.

You can optionally specify a DefaultLayout parameter with a layout class to use for components that don't specify a layout. The default Blazor templates specify the `MainLayout` component. `MainLayout.razor` is in the template

project's `Shared` folder. For more information on layouts, see ASP.NET Core Blazor layouts.

Multiple route templates can be applied to a component. The following component responds to requests for `/BlazorRoute` and `/DifferentBlazorRoute`:

```
@page "/BlazorRoute"
@page "/DifferentBlazorRoute"

<h1>Blazor routing</h1>
```

IMPORTANT

For URLs to resolve correctly, the app must include a `<base>` tag in its `wwwroot/index.html` file (Blazor WebAssembly) or `Pages/_Host.cshtml` file (Blazor Server) with the app base path specified in the `href` attribute (`<base href="/">`). For more information, see Host and deploy ASP.NET Core Blazor.

Provide custom content when content isn't found

The Router component allows the app to specify custom content if content isn't found for the requested route.

In the `App.razor` file, set custom content in the NotFound template parameter of the Router component:

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <h1>Sorry</h1>
        <p>Sorry, there's nothing at this address.</p> b
    </NotFound>
</Router>
```

The content of `<NotFound>` tags can include arbitrary items, such as other interactive components. To apply a default layout to NotFound content, see ASP.NET Core Blazor layouts.

Route to components from multiple assemblies

Use the AdditionalAssemblies parameter to specify additional assemblies for the Router component to consider when searching for routable components. Specified assemblies are considered in addition to the `AppAssembly`-specified assembly. In the following example, `Component1` is a routable component defined in a referenced class library. The following AdditionalAssemblies example results in routing support for `Component1`:

```
<Router
    AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="new[] { typeof(Component1).Assembly }">
    ...
</Router>
```

Route parameters

The router uses route parameters to populate the corresponding component parameters with the same name (case insensitive):

```
@page "/RouteParameter"
@page "/RouteParameter/{text}"

<h1>Blazor is @Text!</h1>

@code {
    [Parameter]
    public string Text { get; set; }

    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

Optional parameters aren't supported. Two `@page` directives are applied in the previous example. The first permits navigation to the component without a parameter. The second `@page` directive takes the `{text}` route parameter and assigns the value to the `Text` property.

Route constraints

A route constraint enforces type matching on a route segment to a component.

In the following example, the route to the `Users` component only matches if:

- An `Id` route segment is present on the request URL.
- The `Id` segment is an integer (`int`).

```
@page "/Users/{Id:int}"

<h1>The user Id is @Id!</h1>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

The route constraints shown in the following table are available. For the route constraints that match with the invariant culture, see the warning below the table for more information.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`bool`	`{active:bool}`	`true`, `FALSE`	No
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Yes
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Yes
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Yes
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Yes

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	No
`int`	`{id:int}`	`123456789`, `-123456789`	Yes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Yes

WARNING

Route constraints that verify the URL and are converted to a CLR type (such as `int` or DateTime) always use the invariant culture. These constraints assume that the URL is non-localizable.

Routing with URLs that contain dots

For hosted Blazor WebAssembly and Blazor Server apps, the server-side default route template assumes that if the last segment of a request URL contains a dot (`.`) that a file is requested (for example, `https://localhost.com:5001/example/some.thing`). Without additional configuration, an app returns a *404 - Not Found* response if this was meant to route to a component. To use a route with one or more parameters that contains a dot, the app must configure the route with a custom template.

Consider the following `Example` component that can receive a route parameter from the last segment of the URL:

```
@page "/example"
@page "/example/{param}"

<p>
    Param: @Param
</p>

@code {
    [Parameter]
    public string Param { get; set; }
}
```

To permit the *Server* app of a hosted Blazor WebAssembly solution to route the request with a dot in the `param` parameter, add a fallback file route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToFile("/example/{param?}", "index.html");
```

To configure a Blazor Server app to route the request with a dot in the `param` parameter, add a fallback page route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToPage("/example/{param?}", "/_Host");
```

For more information, see Routing in ASP.NET Core.

Catch-all route parameters

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components. The catch-all route parameter must be:

- Named to match the route segment name. Naming isn't case sensitive.
- A `string` type. The framework doesn't provide automatic casting.
- At the end of the URL.

```
@page "/page/{*pageRoute}"

@code {
    [Parameter]
    public string PageRoute { get; set; }
}
```

For the URL `/page/this/is/a/test` with a route template of `/page/{*pageRoute}`, the value of `PageRoute` is set to `this/is/a/test`.

Slashes and segments of the captured path are decoded. For a route template of `/page/{*pageRoute}`, the URL `/page/this/is/a%2Ftest%2A` yields `this/is/a/test*`.

Catch-all route parameters are supported in ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.*

NavLink component

Use a NavLink component in place of HTML hyperlink elements (`<a>`) when creating navigation links. A NavLink component behaves like an `<a>` element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed. Optionally, assign a CSS class name to NavLink.ActiveClass to apply a custom CSS class to the rendered link when the current route matches the `href`.

The following `NavMenu` component creates a `Bootstrap` navigation bar that demonstrates how to use NavLink components:

```
<div class="@NavMenuCssClass" @onclick="@ToggleNavMenu">
    <ul class="nav flex-column">
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="MyComponent" Match="NavLinkMatch.Prefix">
                <span class="oi oi-plus" aria-hidden="true"></span> My Component
            </NavLink>
        </li>
    </ul>
</div>
```

There are two NavLinkMatch options that you can assign to the `Match` attribute of the `<NavLink>` element:

- NavLinkMatch.All: The NavLink is active when it matches the entire current URL.
- NavLinkMatch.Prefix (*default*): The NavLink is active when it matches any prefix of the current URL.

In the preceding example, the Home NavLink `href=""` matches the home URL and only receives the `active` CSS class at the app's default base path URL (for example, `https://localhost:5001/`). The second NavLink receives the `active` class when the user visits any URL with a `MyComponent` prefix (for example,

`https://localhost:5001/MyComponent` and `https://localhost:5001/MyComponent/AnotherSegment`).

Additional NavLink component attributes are passed through to the rendered anchor tag. In the following example, the NavLink component includes the `target` attribute:

```
<NavLink href="my-page" target="_blank">My page</NavLink>
```

The following HTML markup is rendered:

```
<a href="my-page" target="_blank">My page</a>
```

WARNING

Due to the way that Blazor renders child content, rendering `NavLink` components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the `NavLink` (child) component's content:

```
@for (int c = 0; c < 10; c++)
{
    var current = c;
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @current
        </NavLink>
    </li>
}
```

Using an index variable in this scenario is a requirement for **any** child component that uses a loop variable in its child content, not just the `NavLink` component.

Alternatively, use a `foreach` loop with Enumerable.Range:

```
@foreach(var c in Enumerable.Range(0,10))
{
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @c
        </NavLink>
    </li>
}
```

URI and navigation state helpers

Use NavigationManager to work with URIs and navigation in C# code. NavigationManager provides the event and methods shown in the following table.

MEMBER	DESCRIPTION
Uri	Gets the current absolute URI.
BaseUri	Gets the base URI (with a trailing slash) that can be prepended to relative URI paths to produce an absolute URI. Typically, BaseUri corresponds to the `href` attribute on the document's `<base>` element in `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server).

MEMBER	DESCRIPTION
NavigateTo	Navigates to the specified URI. If `forceLoad` is `true`: • Client-side routing is bypassed. • The browser is forced to load the new page from the server, whether or not the URI is normally handled by the client-side router.
LocationChanged	An event that fires when the navigation location has changed.
ToAbsoluteUri	Converts a relative URI into an absolute URI.
ToBaseRelativePath	Given a base URI (for example, a URI previously returned by BaseUri), converts an absolute URI into a URI relative to the base URI prefix.

The following component navigates to the app's `Counter` component when the button is selected:

```
@page "/navigate"
@inject NavigationManager NavigationManager

<h1>Navigate in Code Example</h1>

<button class="btn btn-primary" @onclick="NavigateToCounterComponent">
    Navigate to the Counter component
</button>

@code {
    private void NavigateToCounterComponent()
    {
        NavigationManager.NavigateTo("counter");
    }
}
```

The following component handles a location changed event by subscribing to NavigationManager.LocationChanged. The `HandleLocationChanged` method is unhooked when `Dispose` is called by the framework. Unhooking the method permits garbage collection of the component.

```
@implements IDisposable
@inject NavigationManager NavigationManager

...

protected override void OnInitialized()
{
    NavigationManager.LocationChanged += HandleLocationChanged;
}

private void HandleLocationChanged(object sender, LocationChangedEventArgs e)
{
    ...
}

public void Dispose()
{
    NavigationManager.LocationChanged -= HandleLocationChanged;
}
```

LocationChangedEventArgs provides the following information about the event:

- Location: The URL of the new location.
- IsNavigationIntercepted: If `true`, Blazor intercepted the navigation from the browser. If `false`, NavigationManager.NavigateTo caused the navigation to occur.

For more information on component disposal, see ASP.NET Core Blazor lifecycle.

Query string and parse parameters

The query string of a request can be obtained from the NavigationManager's Uri property:

```
@inject NavigationManager Navigation

...

var query = new Uri(Navigation.Uri).Query;
```

To parse a query string's parameters:

- Add a package reference for Microsoft.AspNetCore.WebUtilities.
- Obtain the value after parsing the query string with QueryHelpers.ParseQuery.

```
@page "/"
@using Microsoft.AspNetCore.WebUtilities
@inject NavigationManager NavigationManager

<h1>Query string parse example</h1>

<p>Value: @queryValue</p>

@code {
    private string queryValue = "Not set";

    protected override void OnInitialized()
    {
        var query = new Uri(NavigationManager.Uri).Query;

        if (QueryHelpers.ParseQuery(query).TryGetValue("{KEY}", out var value))
        {
            queryValue = value;
        }
    }
}
```

The placeholder `{KEY}` in the preceding example is the query string parameter key. For example, the URL key-value pair `?ship=Tardis` uses a key of `ship`.

ASP.NET Core Blazor configuration

9/22/2020 • 3 minutes to read • Edit Online

> **NOTE**
> This topic applies to Blazor WebAssembly. For general guidance on ASP.NET Core app configuration, see Configuration in ASP.NET Core.

Blazor WebAssembly loads configuration from app settings files by default:

- `wwwroot/appsettings.json`
- `wwwroot/appsettings.{ENVIRONMENT}.json`

Other configuration providers registered by the app can also provide configuration.

Not all providers or provider features are appropriate for Blazor WebAssembly apps:

- Azure Key Vault configuration provider: The provider isn't supported for managed identity and application ID (client ID) with client secret scenarios. Application ID with a client secret isn't recommended for any ASP.NET Core app, especially Blazor WebAssembly apps because the client secret can't be secured client-side to access to the service.
- Azure App configuration provider: The provider isn't appropriate for Blazor WebAssembly apps because Blazor WebAssembly apps don't run on a server in Azure.

> **WARNING**
> Configuration in a Blazor WebAssembly app is visible to users. **Don't store app secrets or credentials in configuration.**

For more information on configuration providers, see Configuration in ASP.NET Core.

App settings configuration

`wwwroot/appsettings.json`:

```
{
  "message": "Hello from config!"
}
```

Inject an IConfiguration instance into a component to access the configuration data:

```
@page "/"
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<h1>Configuration example</h1>

<p>Message: @Configuration["message"]</p>
```

Custom configuration provider with EF Core

The custom configuration provider with EF Core demonstrated in Configuration in ASP.NET Core works with Blazor WebAssembly apps.

Add the example's configuration provider with the following code in `Program.Main` (`Program.cs`):

```
builder.Configuration.AddEFConfiguration(
    options => options.UseInMemoryDatabase("InMemoryDb"));
```

Inject an IConfiguration instance into a component to access the configuration data:

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<ul>
    <li>@Configuration["quote1"]</li>
    <li>@Configuration["quote2"]</li>
    <li>@Configuration["quote3"]</li>
</ul>
```

Memory Configuration Source

The following example uses a MemoryConfigurationSource to supply additional configuration:

`Program.Main`:

```
using Microsoft.Extensions.Configuration.Memory;

...

var vehicleData = new Dictionary<string, string>()
{
    { "color", "blue" },
    { "type", "car" },
    { "wheels:count", "3" },
    { "wheels:brand", "Blazin" },
    { "wheels:brand:type", "rally" },
    { "wheels:year", "2008" },
};

var memoryConfig = new MemoryConfigurationSource { InitialData = vehicleData };

...

builder.Configuration.Add(memoryConfig);
```

Inject an IConfiguration instance into a component to access the configuration data:

```
@page "/"
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

<h1>Configuration example</h1>

<h2>Wheels</h2>

<ul>
    <li>Count: @Configuration["wheels:count"]</li>
    <li>Brand: @Configuration["wheels:brand"]</li>
    <li>Type: @Configuration["wheels:brand:type"]</li>
    <li>Year: @Configuration["wheels:year"]</li>
</ul>

@code {
    protected override void OnInitialized()
    {
        var wheelsSection = Configuration.GetSection("wheels");

        ...
    }
}
```

To read other configuration files from the `wwwroot` folder into configuration, use an HttpClient to obtain the file's content. When using this approach, the existing HttpClient service registration can use the local client created to read the file, as the following example shows:

`wwwroot/cars.json`:

```
{
    "size": "tiny"
}
```

`Program.Main`:

```
using Microsoft.Extensions.Configuration;

...

var client = new HttpClient()
{
    BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
};

builder.Services.AddScoped(sp => client);

using var response = await client.GetAsync("cars.json");
using var stream = await response.Content.ReadAsStreamAsync();

builder.Configuration.AddJsonStream(stream);
```

Authentication configuration

`wwwroot/appsettings.json`:

```
{
  "Local": {
    "Authority": "{AUTHORITY}",
    "ClientId": "{CLIENT ID}"
  }
}
```

`Program.Main`:

```
builder.Services.AddOidcAuthentication(options =>
    builder.Configuration.Bind("Local", options.ProviderOptions));
```

Logging configuration

Add a package reference for `Microsoft.Extensions.Logging.Configuration`:

```
<PackageReference Include="Microsoft.Extensions.Logging.Configuration" Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

`wwwroot/appsettings.json`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

`Program.Main`:

```
using Microsoft.Extensions.Logging;

...

builder.Logging.AddConfiguration(
    builder.Configuration.GetSection("Logging"));
```

Host builder configuration

`Program.Main`:

```
var hostname = builder.Configuration["HostName"];
```

Cached configuration

Configuration files are cached for offline use. With Progressive Web Applications (PWAs), you can only update configuration files when creating a new deployment. Editing configuration files between deployments has no effect because:

- Users have cached versions of the files that they continue to use.
- The PWA's `service-worker.js` and `service-worker-assets.js` files must be rebuilt on compilation, which signal to the app on the user's next online visit that the app has been redeployed.

For more information on how background updates are handled by PWAs, see Build Progressive Web Applications with ASP.NET Core Blazor WebAssembly.

ASP.NET Core Blazor dependency injection

9/22/2020 • 11 minutes to read • Edit Online

By Rainer Stropek and Mike Rousos

Blazor supports dependency injection (DI). Apps can use built-in services by injecting them into components. Apps can also define and register custom services and make them available throughout the app via DI.

DI is a technique for accessing services configured in a central location. This can be useful in Blazor apps to:

- Share a single instance of a service class across many components, known as a *singleton* service.
- Decouple components from concrete service classes by using reference abstractions. For example, consider an interface `IDataAccess` for accessing data in the app. The interface is implemented by a concrete `DataAccess` class and registered as a service in the app's service container. When a component uses DI to receive an `IDataAccess` implementation, the component isn't coupled to the concrete type. The implementation can be swapped, perhaps for a mock implementation in unit tests.

Default services

Default services are automatically added to the app's service collection.

SERVICE	LIFETIME	DESCRIPTION
HttpClient	Scoped	Provides methods for sending HTTP requests and receiving HTTP responses from a resource identified by a URI. The instance of HttpClient in a Blazor WebAssembly app uses the browser for handling the HTTP traffic in the background. Blazor Server apps don't include an HttpClient configured as a service by default. Provide an HttpClient to a Blazor Server app. For more information, see Call a web API from ASP.NET Core Blazor WebAssembly.
IJSRuntime	Singleton (Blazor WebAssembly) Scoped (Blazor Server)	Represents an instance of a JavaScript runtime where JavaScript calls are dispatched. For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.
NavigationManager	Singleton (Blazor WebAssembly) Scoped (Blazor Server)	Contains helpers for working with URIs and navigation state. For more information, see URI and navigation state helpers.

A custom service provider doesn't automatically provide the default services listed in the table. If you use a custom service provider and require any of the services shown in the table, add the required services to the new service provider.

Add services to an app

Blazor WebAssembly

Configure services for the app's service collection in the `Main` method of `Program.cs`. In the following example, the `MyDependency` implementation is registered for `IMyDependency`:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.Services.AddSingleton<IMyDependency, MyDependency>();
        builder.RootComponents.Add<App>("app");

        builder.Services.AddScoped(sp =>
            new HttpClient
            {
                BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
            });

        await builder.Build().RunAsync();
    }
}
```

Once the host is built, services can be accessed from the root DI scope before any components are rendered. This can be useful for running initialization logic before rendering content:

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.Services.AddSingleton<WeatherService>();
        builder.RootComponents.Add<App>("app");

        builder.Services.AddScoped(sp =>
            new HttpClient
            {
                BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
            });

        var host = builder.Build();

        var weatherService = host.Services.GetRequiredService<WeatherService>();
        await weatherService.InitializeWeatherAsync();

        await host.RunAsync();
    }
}
```

The host also provides a central configuration instance for the app. Building on the preceding example, the weather service's URL is passed from a default configuration source (for example, `appsettings.json`) to `InitializeWeatherAsync`:

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.Services.AddSingleton<WeatherService>();
        builder.RootComponents.Add<App>("app");

        builder.Services.AddScoped(sp =>
            new HttpClient
            {
                BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
            });

        var host = builder.Build();

        var weatherService = host.Services.GetRequiredService<WeatherService>();
        await weatherService.InitializeWeatherAsync(
            host.Configuration["WeatherServiceUrl"]);

        await host.RunAsync();
    }
}
```

Blazor Server

After creating a new app, examine the `Startup.ConfigureServices` method:

```
using Microsoft.Extensions.DependencyInjection;

...

public void ConfigureServices(IServiceCollection services)
{
    ...
}
```

The ConfigureServices method is passed an IServiceCollection, which is a list of service descriptor objects (ServiceDescriptor). Services are added in the `ConfigureServices` method by providing service descriptors to the service collection. The following example demonstrates the concept with the `IDataAccess` interface and its concrete implementation `DataAccess`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDataAccess, DataAccess>();
}
```

Service lifetime

Services can be configured with the lifetimes shown in the following table.

LIFETIME	DESCRIPTION

LIFETIME	DESCRIPTION
Scoped	Blazor WebAssembly apps don't currently have a concept of DI scopes. `Scoped`-registered services behave like `Singleton` services. However, the Blazor Server hosting model supports the `Scoped` lifetime. In Blazor Server apps, a scoped service registration is scoped to the *connection*. For this reason, using scoped services is preferred for services that should be scoped to the current user, even if the current intent is to run client-side in the browser.
Singleton	DI creates a *single instance* of the service. All components requiring a `Singleton` service receive an instance of the same service.
Transient	Whenever a component obtains an instance of a `Transient` service from the service container, it receives a *new instance* of the service.

The DI system is based on the DI system in ASP.NET Core. For more information, see Dependency injection in ASP.NET Core.

Request a service in a component

After services are added to the service collection, inject the services into the components using the @inject Razor directive. `@inject` has two parameters:

- Type: The type of the service to inject.
- Property: The name of the property receiving the injected app service. The property doesn't require manual creation. The compiler creates the property.

For more information, see Dependency injection into views in ASP.NET Core.

Use multiple `@inject` statements to inject different services.

The following example shows how to use `@inject`. The service implementing `Services.IDataAccess` is injected into the component's property `DataRepository`. Note how the code is only using the `IDataAccess` abstraction:

```
@page "/customer-list"
@using Services
@inject IDataAccess DataRepository

@if (customers != null)
{
    <ul>
        @foreach (var customer in customers)
        {
            <li>@customer.FirstName @customer.LastName</li>
        }
    </ul>
}

@code {
    private IReadOnlyList<Customer> customers;

    protected override async Task OnInitializedAsync()
    {
        customers = await DataRepository.GetAllCustomersAsync();
    }
}
```

Internally, the generated property (`DataRepository`) uses the `[Inject]` attribute. Typically, this attribute isn't used directly. If a base class is required for components and injected properties are also required for the base class, manually add the `[Inject]` attribute:

```
using Microsoft.AspNetCore.Components;

public class ComponentBase : IComponent
{
    [Inject]
    protected IDataAccess DataRepository { get; set; }

    ...
}
```

In components derived from the base class, the `@inject` directive isn't required. The InjectAttribute of the base class is sufficient:

```
@page "/demo"
@inherits ComponentBase

<h1>Demo Component</h1>
```

Use DI in services

Complex services might require additional services. In the prior example, `DataAccess` might require the HttpClient default service. `@inject` (or the `[Inject]` attribute) isn't available for use in services. *Constructor injection* must be used instead. Required services are added by adding parameters to the service's constructor. When DI creates the service, it recognizes the services it requires in the constructor and provides them accordingly. In the following example, the constructor receives an HttpClient via DI. HttpClient is a default service.

```
public class DataAccess : IDataAccess
{
    public DataAccess(HttpClient client)
    {
        ...
    }
}
```

Prerequisites for constructor injection:

- One constructor must exist whose arguments can all be fulfilled by DI. Additional parameters not covered by DI are allowed if they specify default values.
- The applicable constructor must be `public`.
- One applicable constructor must exist. In case of an ambiguity, DI throws an exception.

Utility base component classes to manage a DI scope

In ASP.NET Core apps, scoped services are typically scoped to the current request. After the request completes, any scoped or transient services are disposed by the DI system. In Blazor Server apps, the request scope lasts for the duration of the client connection, which can result in transient and scoped services living much longer than expected. In Blazor WebAssembly apps, services registered with a scoped lifetime are treated as singletons, so they live longer than scoped services in typical ASP.NET Core apps.

> **NOTE**
>
> To detect disposable transient services in an app, see the Detect transient disposables section.

An approach that limits a service lifetime in Blazor apps is use of the OwningComponentBase type. OwningComponentBase is an abstract type derived from ComponentBase that creates a DI scope corresponding to the lifetime of the component. Using this scope, it's possible to use DI services with a scoped lifetime and have them live as long as the component. When the component is destroyed, services from the component's scoped service provider are disposed as well. This can be useful for services that:

- Should be reused within a component, as the transient lifetime is inappropriate.
- Shouldn't be shared across components, as the singleton lifetime is inappropriate.

Two versions of the OwningComponentBase type are available:

- OwningComponentBase is an abstract, disposable child of the ComponentBase type with a protected ScopedServices property of type IServiceProvider. This provider can be used to resolve services that are scoped to the lifetime of the component.

  DI services injected into the component using `@inject` or the `[Inject]` attribute aren't created in the component's scope. To use the component's scope, services must be resolved using GetRequiredService or GetService. Any services resolved using the ScopedServices provider have their dependencies provided from that same scope.

```
@page "/preferences"
@using Microsoft.Extensions.DependencyInjection
@inherits OwningComponentBase

<h1>User (@UserService.Name)</h1>

<ul>
    @foreach (var setting in SettingService.GetSettings())
    {
        <li>@setting.SettingName: @setting.SettingValue</li>
    }
</ul>

@code {
    private IUserService UserService { get; set; }
    private ISettingService SettingService { get; set; }

    protected override void OnInitialized()
    {
        UserService = ScopedServices.GetRequiredService<IUserService>();
        SettingService = ScopedServices.GetRequiredService<ISettingService>();
    }
}
```

- OwningComponentBase<TService> derives from OwningComponentBase and adds a Service property that returns an instance of `T` from the scoped DI provider. This type is a convenient way to access scoped services without using an instance of IServiceProvider when there's one primary service the app requires from the DI container using the component's scope. The ScopedServices property is available, so the app can get services of other types, if necessary.

```
@page "/users"
@attribute [Authorize]
@inherits OwningComponentBase<AppDbContext>

<h1>Users (@Service.Users.Count())</h1>

<ul>
    @foreach (var user in Service.Users)
    {
        <li>@user.UserName</li>
    }
</ul>
```

Use of an Entity Framework Core (EF Core) DbContext from DI

For more information, see ASP.NET Core Blazor Server with Entity Framework Core (EFCore).

Detect transient disposables

The following examples show how to detect disposable transient services in an app that should use OwningComponentBase. For more information, see the Utility base component classes to manage a DI scope section.

Blazor WebAssembly

`DetectIncorrectUsagesOfTransientDisposables.cs`:

```
using System;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
```

```
namespace Microsoft.Extensions.DependencyInjection
{
    using BlazorServerTransientDisposable;
    using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

    public static class WebHostBuilderTransientDisposableExtensions
    {
        public static WebAssemblyHostBuilder DetectIncorrectUsageOfTransients(
            this WebAssemblyHostBuilder builder)
        {
            builder
                .ConfigureContainer(
                    new DetectIncorrectUsageOfTransientDisposablesServiceFactory());

            return builder;
        }

        public static WebAssemblyHost EnableTransientDisposableDetection(
            this WebAssemblyHost webAssemblyHost)
        {
            webAssemblyHost.Services
                .GetRequiredService<ThrowOnTransientDisposable>().ShouldThrow = true;
            return webAssemblyHost;
        }
    }
}

namespace BlazorServerTransientDisposable
{
    public class DetectIncorrectUsageOfTransientDisposablesServiceFactory
        : IServiceProviderFactory<IServiceCollection>
    {
        public IServiceCollection CreateBuilder(IServiceCollection services) =>
            services;

        public IServiceProvider CreateServiceProvider(
            IServiceCollection containerBuilder)
        {
            var collection = new ServiceCollection();
            foreach (var descriptor in containerBuilder)
            {
                if (descriptor.Lifetime == ServiceLifetime.Transient &&
                    descriptor.ImplementationType != null &&
                    typeof(IDisposable).IsAssignableFrom(
                        descriptor.ImplementationType))
                {
                    collection.Add(CreatePatchedDescriptor(descriptor));
                }
                else if (descriptor.Lifetime == ServiceLifetime.Transient &&
                         descriptor.ImplementationFactory != null)
                {
                    collection.Add(CreatePatchedFactoryDescriptor(descriptor));
                }
                else
                {
                    collection.Add(descriptor);
                }
            }

            collection.AddScoped<ThrowOnTransientDisposable>();

            return collection.BuildServiceProvider();
        }

        private ServiceDescriptor CreatePatchedFactoryDescriptor(
            ServiceDescriptor original)
        {
            var newDescriptor = new ServiceDescriptor(
```

```
                original.ServiceType,
                (sp) =>
                {
                    var originalFactory = original.ImplementationFactory;
                    var originalResult = originalFactory(sp);

                    var throwOnTransientDisposable =
                        sp.GetRequiredService<ThrowOnTransientDisposable>();
                    if (throwOnTransientDisposable.ShouldThrow &&
                        originalResult is IDisposable d)
                    {
                        throw new InvalidOperationException("Trying to resolve " +
                            $"transient disposable service {d.GetType().Name} in " +
                            "the wrong scope. Use an 'OwningComponentBase<T>' " +
                            "component base class for the service 'T' you are " +
                            "trying to resolve.");
                    }

                    return originalResult;
                },
                original.Lifetime);

            return newDescriptor;
        }

        private ServiceDescriptor CreatePatchedDescriptor(ServiceDescriptor original)
        {
            var newDescriptor = new ServiceDescriptor(
                original.ServiceType,
                (sp) => {
                    var throwOnTransientDisposable =
                        sp.GetRequiredService<ThrowOnTransientDisposable>();
                    if (throwOnTransientDisposable.ShouldThrow)
                    {
                        throw new InvalidOperationException("Trying to resolve " +
                        "transient disposable service " +
                        $"{original.ImplementationType.Name} in the wrong " +
                        "scope. Use an 'OwningComponentBase<T>' component base " +
                        "class for the service 'T' you are trying to resolve.");
                    }

                    return ActivatorUtilities.CreateInstance(sp,
                        original.ImplementationType);
                },
                ServiceLifetime.Transient);
            return newDescriptor;
        }
    }

    internal class ThrowOnTransientDisposable
    {
        public bool ShouldThrow { get; set; }
    }
}
```

The `TransientDisposable` in the following example is detected (`Program.cs`):

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.DetectIncorrectUsageOfTransients();
        builder.RootComponents.Add<App>("app");

        builder.Services.AddTransient<TransientDisposable>();
        builder.Services.AddScoped(sp =>
            new HttpClient
            {
                BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
            });

        var host = builder.Build();
        host.EnableTransientDisposableDetection();
        await host.RunAsync();
    }
}

public class TransientDisposable : IDisposable
{
    public void Dispose() => throw new NotImplementedException();
}
```

Blazor Server

`DetectIncorrectUsagesOfTransientDisposables.cs`:

```
using System;
using Microsoft.AspNetCore.Components.Server.Circuits;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace Microsoft.Extensions.DependencyInjection
{
    using BlazorServerTransientDisposable;

    public static class WebHostBuilderTransientDisposableExtensions
    {
        public static IHostBuilder DetectIncorrectUsageOfTransients(
            this IHostBuilder builder)
        {
            builder
                .UseServiceProviderFactory(
                    new DetectIncorrectUsageOfTransientDisposablesServiceFactory())
                .ConfigureServices(
                    s => s.TryAddEnumerable(ServiceDescriptor.Scoped<CircuitHandler,
                        ThrowOnTransientDisposableHandler>()));

            return builder;
        }
    }
}

namespace BlazorServerTransientDisposable
{
    internal class ThrowOnTransientDisposableHandler : CircuitHandler
    {
        public ThrowOnTransientDisposableHandler(
            ThrowOnTransientDisposable throwOnTransientDisposable)
        {
            throwOnTransientDisposable.ShouldThrow = true;
```

```
    }
}

public class DetectIncorrectUsageOfTransientDisposablesServiceFactory
    : IServiceProviderFactory<IServiceCollection>
{
    public IServiceCollection CreateBuilder(IServiceCollection services) =>
        services;

    public IServiceProvider CreateServiceProvider(
        IServiceCollection containerBuilder)
    {
        var collection = new ServiceCollection();
        foreach (var descriptor in containerBuilder)
        {
            if (descriptor.Lifetime == ServiceLifetime.Transient &&
                descriptor.ImplementationType != null &&
                typeof(IDisposable).IsAssignableFrom(
                    descriptor.ImplementationType))
            {
                collection.Add(CreatePatchedDescriptor(descriptor));
            }
            else if (descriptor.Lifetime == ServiceLifetime.Transient &&
                     descriptor.ImplementationFactory != null)
            {
                collection.Add(CreatePatchedFactoryDescriptor(descriptor));
            }
            else
            {
                collection.Add(descriptor);
            }
        }

        collection.AddScoped<ThrowOnTransientDisposable>();

        return collection.BuildServiceProvider();
    }

    private ServiceDescriptor CreatePatchedFactoryDescriptor(
        ServiceDescriptor original)
    {
        var newDescriptor = new ServiceDescriptor(
            original.ServiceType,
            (sp) =>
            {
                var originalFactory = original.ImplementationFactory;
                var originalResult = originalFactory(sp);

                var throwOnTransientDisposable =
                    sp.GetRequiredService<ThrowOnTransientDisposable>();
                if (throwOnTransientDisposable.ShouldThrow &&
                    originalResult is IDisposable d)
                {
                    throw new InvalidOperationException("Trying to resolve " +
                        $"transient disposable service {d.GetType().Name} in " +
                        "the wrong scope. Use an 'OwningComponentBase<T>' " +
                        "component base class for the service 'T' you are " +
                        "trying to resolve.");
                }

                return originalResult;
            },
            original.Lifetime);

        return newDescriptor;
    }

    private ServiceDescriptor CreatePatchedDescriptor(
        ServiceDescriptor original)
```

```
        {
            var newDescriptor = new ServiceDescriptor(
                original.ServiceType,
                (sp) => {
                    var throwOnTransientDisposable =
                        sp.GetRequiredService<ThrowOnTransientDisposable>();
                    if (throwOnTransientDisposable.ShouldThrow)
                    {
                        throw new InvalidOperationException("Trying to resolve " +
                            "transient disposable service " +
                            $"{original.ImplementationType.Name} in the wrong " +
                            "scope. Use an 'OwningComponentBase<T>' component " +
                            "base class for the service 'T' you are trying to " +
                            "resolve.");
                    }

                    return ActivatorUtilities.CreateInstance(sp,
                        original.ImplementationType);
                },
                ServiceLifetime.Transient);
            return newDescriptor;
        }
    }

    internal class ThrowOnTransientDisposable
    {
        public bool ShouldThrow { get; set; }
    }
}
```

`Program`:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .DetectIncorrectUsageOfTransients()
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

The `TransientDependency` in the following example is detected (`Startup.cs`):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddServerSideBlazor();
    services.AddSingleton<WeatherForecastService>();
    services.AddTransient<TransientDependency>();
    services.AddTransient<ITransitiveTransientDisposableDependency,
        TransitiveTransientDisposableDependency>();
}

public class TransitiveTransientDisposableDependency
    : ITransitiveTransientDisposableDependency, IDisposable
{
    public void Dispose() { }
}

public interface ITransitiveTransientDisposableDependency
{
}

public class TransientDependency
{
    private readonly ITransitiveTransientDisposableDependency
        _transitiveTransientDisposableDependency;

    public TransientDependency(ITransitiveTransientDisposableDependency
        transitiveTransientDisposableDependency)
    {
        _transitiveTransientDisposableDependency =
            transitiveTransientDisposableDependency;
    }
}
```

Additional resources

- Dependency injection in ASP.NET Core
- `IDisposable` guidance for Transient and shared instances
- Dependency injection into views in ASP.NET Core

ASP.NET Core Blazor environments

9/22/2020 • 2 minutes to read • Edit Online

> **NOTE**
>
> This topic applies to Blazor WebAssembly. For general guidance on ASP.NET Core app configuration, see Use multiple environments in ASP.NET Core.

When running an app locally, the environment defaults to Development. When the app is published, the environment defaults to Production.

A hosted Blazor WebAssembly app picks up the environment from the server via a middleware that communicates the environment to the browser by adding the `blazor-environment` header. The value of the header is the environment. The hosted Blazor app and the server app share the same environment. For more information, including how to configure the environment, see Use multiple environments in ASP.NET Core.

For a standalone app running locally, the development server adds the `blazor-environment` header to specify the Development environment. To specify the environment for other hosting environments, add the `blazor-environment` header.

In the following example for IIS, add the custom header to the published `web.config` file. The `web.config` file is located in the `bin/Release/{TARGET FRAMEWORK}/publish` folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>

    ...

    <httpProtocol>
      <customHeaders>
        <add name="blazor-environment" value="Staging" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

> **NOTE**
>
> To use a custom `web.config` file for IIS that isn't overwritten when the app is published to the `publish` folder, see Host and deploy ASP.NET Core Blazor WebAssembly.

Obtain the app's environment in a component by injecting IWebAssemblyHostEnvironment and reading the Environment property:

```
@page "/"
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
@inject IWebAssemblyHostEnvironment HostEnvironment

<h1>Environment example</h1>

<p>Environment: @HostEnvironment.Environment</p>
```

During startup, the WebAssemblyHostBuilder exposes the IWebAssemblyHostEnvironment through the HostEnvironment property, which enables developers to have environment-specific logic in their code:

```
if (builder.HostEnvironment.Environment == "Custom")
{
    ...
};
```

The following convenience extension methods permit checking the current environment for Development, Production, Staging, and custom environment names:

- `IsDevelopment()`
- `IsProduction()`
- `IsStaging()`
- `IsEnvironment("{ENVIRONMENT NAME}")`

```
if (builder.HostEnvironment.IsStaging())
{
    ...
};

if (builder.HostEnvironment.IsEnvironment("Custom"))
{
    ...
};
```

The IWebAssemblyHostEnvironment.BaseAddress property can be used during startup when the NavigationManager service isn't available.

Additional resources

- Use multiple environments in ASP.NET Core

ASP.NET Core Blazor logging

9/22/2020 • 2 minutes to read • Edit Online

Blazor WebAssembly

Configure logging in Blazor WebAssembly apps with the `WebAssemblyHostBuilder.Logging` property in `Program.Main`:

```
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

...

var builder = WebAssemblyHostBuilder.CreateDefault(args);

builder.Logging.SetMinimumLevel(LogLevel.Debug);
builder.Logging.AddProvider(new CustomLoggingProvider());
```

The `Logging` property is of type ILoggingBuilder, so all of the extension methods available on ILoggingBuilder are also available on `Logging`.

Logging configuration can be loaded from app settings files. For more information, see ASP.NET Core Blazor configuration.

Blazor Server

For general ASP.NET Core logging guidance, see Logging in .NET Core and ASP.NET Core.

Blazor WebAssembly SignalR .NET client logging

Inject an ILoggerProvider to add a `WebAssemblyConsoleLogger` to the logging providers passed to HubConnectionBuilder. Unlike a traditional ConsoleLogger, `WebAssemblyConsoleLogger` is a wrapper around browser-specific logging APIs (for example, `console.log`). Use of `WebAssemblyConsoleLogger` makes logging possible within Mono inside a browser context.

```
@using Microsoft.Extensions.Logging
@inject ILoggerProvider LoggerProvider

...

var connection = new HubConnectionBuilder()
    .WithUrl(NavigationManager.ToAbsoluteUri("/chathub"))
    .ConfigureLogging(logging => logging.AddProvider(LoggerProvider))
    .Build();
```

Log in Razor components

Loggers respect app startup configuration.

The `using` directive for Microsoft.Extensions.Logging is required to support Intellisense completions for APIs, such as LogWarning and LogError.

The following example demonstrates logging with an ILogger in Razor components:

```
@page "/counter"
@using Microsoft.Extensions.Logging;
@inject ILogger<Counter> logger;

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        logger.LogWarning("Someone has clicked me!");

        currentCount++;
    }
}
```

The following example demonstrates logging with an ILoggerFactory in Razor components:

```
@page "/counter"
@using Microsoft.Extensions.Logging;
@inject ILoggerFactory LoggerFactory

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        var logger = LoggerFactory.CreateLogger<Counter>();
        logger.LogWarning("Someone has clicked me!");

        currentCount++;
    }
}
```

Additional resources

- Logging in .NET Core and ASP.NET Core

Handle errors in ASP.NET Core Blazor apps

9/22/2020 • 11 minutes to read • Edit Online

By Steve Sanderson

This article describes how Blazor manages unhandled exceptions and how to develop apps that detect and handle errors.

Detailed errors during development

When a Blazor app isn't functioning properly during development, receiving detailed error information from the app assists in troubleshooting and fixing the issue. When an error occurs, Blazor apps display a gold bar at the bottom of the screen:

- During development, the gold bar directs you to the browser console, where you can see the exception.
- In production, the gold bar notifies the user that an error has occurred and recommends refreshing the browser.

The UI for this error handling experience is part of the Blazor project templates.

In a Blazor WebAssembly app, customize the experience in the `wwwroot/index.html` file:

```
<div id="blazor-error-ui">
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">🗙</a>
</div>
```

In a Blazor Server app, customize the experience in the `Pages/_Host.cshtml` file:

```
<div id="blazor-error-ui">
    <environment include="Staging,Production">
        An error has occurred. This application may no longer respond until reloaded.
    </environment>
    <environment include="Development">
        An unhandled exception has occurred. See browser dev tools for details.
    </environment>
    <a href="" class="reload">Reload</a>
    <a class="dismiss">🗙</a>
</div>
```

The `blazor-error-ui` element is hidden by the styles included in the Blazor templates (`wwwroot/css/app.css` or `wwwroot/css/site.css`) and then shown when an error occurs:

```
#blazor-error-ui {
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}

#blazor-error-ui .dismiss {
    cursor: pointer;
    position: absolute;
    right: 0.75rem;
    top: 0.5rem;
}
```

How a Blazor Server app reacts to unhandled exceptions

Blazor Server is a stateful framework. While users interact with an app, they maintain a connection to the server known as a *circuit*. The circuit holds active component instances, plus many other aspects of state, such as:

- The most recent rendered output of components.
- The current set of event-handling delegates that could be triggered by client-side events.

If a user opens the app in multiple browser tabs, they have multiple independent circuits.

Blazor treats most unhandled exceptions as fatal to the circuit where they occur. If a circuit is terminated due to an unhandled exception, the user can only continue to interact with the app by reloading the page to create a new circuit. Circuits outside of the one that's terminated, which are circuits for other users or other browser tabs, aren't affected. This scenario is similar to a desktop app that crashes. The crashed app must be restarted, but other apps aren't affected.

A circuit is terminated when an unhandled exception occurs for the following reasons:

- An unhandled exception often leaves the circuit in an undefined state.
- The app's normal operation can't be guaranteed after an unhandled exception.
- Security vulnerabilities may appear in the app if the circuit continues.

Manage unhandled exceptions in developer code

For an app to continue after an error, the app must have error handling logic. Later sections of this article describe potential sources of unhandled exceptions.

In production, don't render framework exception messages or stack traces in the UI. Rendering exception messages or stack traces could:

- Disclose sensitive information to end users.
- Help a malicious user discover weaknesses in an app that can compromise the security of the app, server, or network.

Log errors with a persistent provider

If an unhandled exception occurs, the exception is logged to ILogger instances configured in the service container. By default, Blazor apps log to console output with the Console Logging Provider. Consider logging to a more permanent location with a provider that manages log size and log rotation. For more information, see Logging in

.NET Core and ASP.NET Core.

During development, Blazor usually sends the full details of exceptions to the browser's console to aid in debugging. In production, detailed errors in the browser's console are disabled by default, which means that errors aren't sent to clients but the exception's full details are still logged server-side. For more information, see Handle errors in ASP.NET Core.

You must decide which incidents to log and the level of severity of logged incidents. Hostile users might be able to trigger errors deliberately. For example, don't log an incident from an error where an unknown `ProductId` is supplied in the URL of a component that displays product details. Not all errors should be treated as high-severity incidents for logging.

For more information, see ASP.NET Core Blazor logging.

Places where errors may occur

Framework and app code may trigger unhandled exceptions in any of the following locations:

- Component instantiation
- Lifecycle methods
- Rendering logic
- Event handlers
- Component disposal
- JavaScript interop
- Blazor Server rerendering

The preceding unhandled exceptions are described in the following sections of this article.

Component instantiation

When Blazor creates an instance of a component:

- The component's constructor is invoked.
- The constructors of any non-singleton DI services supplied to the component's constructor via the `@inject` directive or the `[Inject]` attribute are invoked.

A Blazor Server circuit fails when any executed constructor or a setter for any `[Inject]` property throws an unhandled exception. The exception is fatal because the framework can't instantiate the component. If constructor logic may throw exceptions, the app should trap the exceptions using a `try-catch` statement with error handling and logging.

Lifecycle methods

During the lifetime of a component, Blazor invokes the following lifecycle methods:

- OnInitialized / OnInitializedAsync
- OnParametersSet / OnParametersSetAsync
- ShouldRender
- OnAfterRender / OnAfterRenderAsync

If any lifecycle method throws an exception, synchronously or asynchronously, the exception is fatal to a Blazor Server circuit. For components to deal with errors in lifecycle methods, add error handling logic.

In the following example where OnParametersSetAsync calls a method to obtain a product:

- An exception thrown in the `ProductRepository.GetProductByIdAsync` method is handled by a `try-catch` statement.
- When the `catch` block is executed:

- `loadFailed` is set to `true`, which is used to display an error message to the user.
- The error is logged.

```
@page "/product-details/{ProductId:int}"
@using Microsoft.Extensions.Logging
@inject IProductRepository ProductRepository
@inject ILogger<ProductDetails> Logger

@if (details != null)
{
    <h1>@details.ProductName</h1>
    <p>@details.Description</p>
}
else if (loadFailed)
{
    <h1>Sorry, we could not load this product due to an error.</h1>
}
else
{
    <h1>Loading...</h1>
}

@code {
    private ProductDetails details;
    private bool loadFailed;

    [Parameter]
    public int ProductId { get; set; }

    protected override async Task OnParametersSetAsync()
    {
        try
        {
            loadFailed = false;
            details = await ProductRepository.GetProductByIdAsync(ProductId);
        }
        catch (Exception ex)
        {
            loadFailed = true;
            Logger.LogWarning(ex, "Failed to load product {ProductId}", ProductId);
        }
    }
}
```

Rendering logic

The declarative markup in a `.razor` component file is compiled into a C# method called BuildRenderTree. When a component renders, BuildRenderTree executes and builds up a data structure describing the elements, text, and child components of the rendered component.

Rendering logic can throw an exception. An example of this scenario occurs when `@someObject.PropertyName` is evaluated but `@someObject` is `null`. An unhandled exception thrown by rendering logic is fatal to a Blazor Server circuit.

To prevent a null reference exception in rendering logic, check for a `null` object before accessing its members. In the following example, `person.Address` properties aren't accessed if `person.Address` is `null`:

```
@if (person.Address != null)
{
    <div>@person.Address.Line1</div>
    <div>@person.Address.Line2</div>
    <div>@person.Address.City</div>
    <div>@person.Address.Country</div>
}
```

The preceding code assumes that `person` isn't `null`. Often, the structure of the code guarantees that an object exists at the time the component is rendered. In those cases, it isn't necessary to check for `null` in rendering logic. In the prior example, `person` might be guaranteed to exist because `person` is created when the component is instantiated.

Event handlers

Client-side code triggers invocations of C# code when event handlers are created using:

- `@onclick`
- `@onchange`
- Other `@on...` attributes
- `@bind`

Event handler code might throw an unhandled exception in these scenarios.

If an event handler throws an unhandled exception (for example, a database query fails), the exception is fatal to a Blazor Server circuit. If the app calls code that could fail for external reasons, trap exceptions using a `try-catch` statement with error handling and logging.

If user code doesn't trap and handle the exception, the framework logs the exception and terminates the circuit.

Component disposal

A component may be removed from the UI, for example, because the user has navigated to another page. When a component that implements System.IDisposable is removed from the UI, the framework calls the component's Dispose method.

If the component's `Dispose` method throws an unhandled exception, the exception is fatal to a Blazor Server circuit. If disposal logic may throw exceptions, the app should trap the exceptions using a `try-catch` statement with error handling and logging.

For more information on component disposal, see ASP.NET Core Blazor lifecycle.

JavaScript interop

IJSRuntime.InvokeAsync allows .NET code to make asynchronous calls to the JavaScript runtime in the user's browser.

The following conditions apply to error handling with InvokeAsync:

- If a call to InvokeAsync fails synchronously, a .NET exception occurs. A call to InvokeAsync may fail, for example, because the supplied arguments can't be serialized. Developer code must catch the exception. If app code in an event handler or component lifecycle method doesn't handle an exception, the resulting exception is fatal to a Blazor Server circuit.
- If a call to InvokeAsync fails asynchronously, the .NET Task fails. A call to InvokeAsync may fail, for example, because the JavaScript-side code throws an exception or returns a `Promise` that completed as `rejected`. Developer code must catch the exception. If using the `await` operator, consider wrapping the method call in a `try-catch` statement with error handling and logging. Otherwise, the failing code results in an unhandled exception that's fatal to a Blazor Server circuit.
- By default, calls to InvokeAsync must complete within a certain period or else the call times out. The default

timeout period is one minute. The timeout protects the code against a loss in network connectivity or JavaScript code that never sends back a completion message. If the call times out, the resulting System.Threading.Tasks fails with an OperationCanceledException. Trap and process the exception with logging.

Similarly, JavaScript code may initiate calls to .NET methods indicated by the `[JSInvokable]` attribute. If these .NET methods throw an unhandled exception:

- The exception isn't treated as fatal to a Blazor Server circuit.
- The JavaScript-side `Promise` is rejected.

You have the option of using error handling code on either the .NET side or the JavaScript side of the method call.

For more information, see the following articles:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Blazor Server prerendering

Blazor components can be prerendered using the Component Tag Helper so that their rendered HTML markup is returned as part of the user's initial HTTP request. This works by:

- Creating a new circuit for all of the prerendered components that are part of the same page.
- Generating the initial HTML.
- Treating the circuit as `disconnected` until the user's browser establishes a SignalR connection back to the same server. When the connection is established, interactivity on the circuit is resumed and the components' HTML markup is updated.

If any component throws an unhandled exception during prerendering, for example, during a lifecycle method or in rendering logic:

- The exception is fatal to the circuit.
- The exception is thrown up the call stack from the ComponentTagHelper Tag Helper. Therefore, the entire HTTP request fails unless the exception is explicitly caught by developer code.

Under normal circumstances when prerendering fails, continuing to build and render the component doesn't make sense because a working component can't be rendered.

To tolerate errors that may occur during prerendering, error handling logic must be placed inside a component that may throw exceptions. Use `try-catch` statements with error handling and logging. Instead of wrapping the ComponentTagHelper Tag Helper in a `try-catch` statement, place error handling logic in the component rendered by the ComponentTagHelper Tag Helper.

Advanced scenarios

Recursive rendering

Components can be nested recursively. This is useful for representing recursive data structures. For example, a `TreeNode` component can render more `TreeNode` components for each of the node's children.

When rendering recursively, avoid coding patterns that result in infinite recursion:

- Don't recursively render a data structure that contains a cycle. For example, don't render a tree node whose children includes itself.
- Don't create a chain of layouts that contain a cycle. For example, don't create a layout whose layout is itself.
- Don't allow an end user to violate recursion invariants (rules) through malicious data entry or JavaScript interop calls.

Infinite loops during rendering:

- Causes the rendering process to continue forever.
- Is equivalent to creating an unterminated loop.

In these scenarios, an affected Blazor Server circuit fails, and the thread usually attempts to:

- Consume as much CPU time as permitted by the operating system, indefinitely.
- Consume an unlimited amount of server memory. Consuming unlimited memory is equivalent to the scenario where an unterminated loop adds entries to a collection on every iteration.

To avoid infinite recursion patterns, ensure that recursive rendering code contains suitable stopping conditions.

Custom render tree logic

Most Blazor components are implemented as `.razor` files and are compiled to produce logic that operates on a RenderTreeBuilder to render their output. A developer may manually implement RenderTreeBuilder logic using procedural C# code. For more information, see ASP.NET Core Blazor advanced scenarios.

> **WARNING**
> Use of manual render tree builder logic is considered an advanced and unsafe scenario, not recommended for general component development.

If RenderTreeBuilder code is written, the developer must guarantee the correctness of the code. For example, the developer must ensure that:

- Calls to OpenElement and CloseElement are correctly balanced.
- Attributes are only added in the correct places.

Incorrect manual render tree builder logic can cause arbitrary undefined behavior, including crashes, server hangs, and security vulnerabilities.

Consider manual render tree builder logic on the same level of complexity and with the same level of *danger* as writing assembly code or MSIL instructions by hand.

ASP.NET Core Blazor hosting model configuration

9/22/2020 • 7 minutes to read • Edit Online

By Daniel Roth, Mackinnon Buck, and Luke Latham

This article covers hosting model configuration.

SignalR cross-origin negotiation for authentication

This section applies to Blazor WebAssembly.

To configure SignalR's underlying client to send credentials, such as cookies or HTTP authentication headers:

- Use SetBrowserRequestCredentials to set Include on cross-origin `fetch` requests:

```
public class IncludeRequestCredentialsMessageHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        return base.SendAsync(request, cancellationToken);
    }
}
```

- Assign the HttpMessageHandler to the HttpMessageHandlerFactory option:

```
var connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://signalr.example.com"), options =>
    {
        options.HttpMessageHandlerFactory = innerHandler =>
            new IncludeRequestCredentialsMessageHandler { InnerHandler = innerHandler };
    }).Build();
```

For more information, see ASP.NET Core SignalR configuration.

Reflect the connection state in the UI

This section applies to Blazor Server.

When the client detects that the connection has been lost, a default UI is displayed to the user while the client attempts to reconnect. If reconnection fails, the user is provided the option to retry.

To customize the UI, define an element with an `id` of `components-reconnect-modal` in the `<body>` of the `_Host.cshtml` Razor page:

```
<div id="components-reconnect-modal">
    ...
</div>
```

Add the following to the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`):

```
#components-reconnect-modal {
    display: none;
}

#components-reconnect-modal.components-reconnect-show {
    display: block;
}
```

The following table describes the CSS classes applied to the `components-reconnect-modal` element.

CSS CLASS	INDICATES...
`components-reconnect-show`	A lost connection. The client is attempting to reconnect. Show the modal.
`components-reconnect-hide`	An active connection is re-established to the server. Hide the modal.
`components-reconnect-failed`	Reconnection failed, probably due to a network failure. To attempt reconnection, call `window.Blazor.reconnect()`.
`components-reconnect-rejected`	Reconnection rejected. The server was reached but refused the connection, and the user's state on the server is lost. To reload the app, call `location.reload()`. This connection state may result when: • A crash in the server-side circuit occurs. • The client is disconnected long enough for the server to drop the user's state. Instances of the components that the user is interacting with are disposed. • The server is restarted, or the app's worker process is recycled.

Render mode

This section applies to Blazor Server.

Blazor Server apps are set up by default to prerender the UI on the server before the client connection to the server is established. This is set up in the `_Host.cshtml` Razor page:

```
<body>
    <app>
      <component type="typeof(App)" render-mode="ServerPrerendered" />
    </app>

    <script src="_framework/blazor.server.js"></script>
</body>
```

RenderMode configures whether the component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

RENDER MODE	DESCRIPTION
ServerPrerendered	Renders the component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a Blazor Server app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the component into static HTML.

Rendering server components from a static HTML page isn't supported.

Initialize the Blazor circuit

This section applies to Blazor Server.

Configure the manual start of a Blazor Server app's SignalR circuit in the `Pages/_Host.cshtml` file:

- Add an `autostart="false"` attribute to the `<script>` tag for the `blazor.server.js` script.
- Place a script that calls `Blazor.start` after the `blazor.server.js` script's tag and inside the closing `</body>` tag.

When `autostart` is disabled, any aspect of the app that doesn't depend on the circuit works normally. For example, client-side routing is operational. However, any aspect that depends on the circuit isn't operational until `Blazor.start` is called. App behavior is unpredictable without an established circuit. For example, component methods fail to execute while the circuit is disconnected.

Initialize Blazor when the document is ready

To initialize the Blazor app when the document is ready:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      document.addEventListener("DOMContentLoaded", function() {
        Blazor.start();
      });
    </script>
</body>
```

Chain to the `Promise` that results from a manual start

To perform additional tasks, such as JS interop initialization, use `then` to chain to the `Promise` that results from a manual Blazor app start:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start().then(function () {
        ...
      });
    </script>
</body>
```

Configure the SignalR client

Logging

To configure SignalR client logging, pass in a configuration object (`configureSignalR`) that calls `configureLogging` with the log level on the client builder:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        configureSignalR: function (builder) {
          builder.configureLogging("information");
        }
      });
    </script>
</body>
```

In the preceding example, `information` is equivalent to a log level of LogLevel.Information.

Modify the reconnection handler

The reconnection handler's circuit connection events can be modified for custom behaviors, such as:

- To notify the user if the connection is dropped.
- To perform logging (from the client) when a circuit is connected.

To modify the connection events, register callbacks for the following connection changes:

- Dropped connections use `onConnectionDown` .
- Established/re-established connections use `onConnectionUp` .

Both `onConnectionDown` and `onConnectionUp` must be specified:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionHandler: {
          onConnectionDown: (options, error) => console.error(error);
          onConnectionUp: () => console.log("Up, up, and away!");
        }
      });
    </script>
</body>
```

Adjust the reconnection retry count and interval

To adjust the reconnection retry count and interval, set the number of retries (`maxRetries`) and period in milliseconds permitted for each retry attempt (`retryIntervalMilliseconds`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionOptions: {
          maxRetries: 3,
          retryIntervalMilliseconds: 2000
        }
      });
    </script>
</body>
```

Hide or replace the reconnection display

To hide the reconnection display, set the reconnection handler's `_reconnectionDisplay` to an empty object (`{}` or `new Object()`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      window.addEventListener('beforeunload', function () {
        Blazor.defaultReconnectionHandler._reconnectionDisplay = {};
      });

      Blazor.start();
    </script>
</body>
```

To replace the reconnection display, set `_reconnectionDisplay` in the preceding example to the element for display:

```
Blazor.defaultReconnectionHandler._reconnectionDisplay =
  document.getElementById("{ELEMENT ID}");
```

The placeholder `{ELEMENT ID}` is the ID of the HTML element to display.

Customize the delay before the reconnection display appears by setting the `transition-delay` property in the app's CSS (`wwwroot/css/site.css`) for the modal element. The following example sets the transition delay from 500 ms (default) to 1,000 ms (1 second):

```
#components-reconnect-modal {
    transition: visibility 0s linear 1000ms;
}
```

Influence HTML `<head>` tag elements

This section applies to the upcoming ASP.NET Core 5.0 release of Blazor WebAssembly and Blazor Server.

When rendered, the `Title`, `Link`, and `Meta` components add or update data in the HTML `<head>` tag elements:

```
@using Microsoft.AspNetCore.Components.Web.Extensions.Head

<Title Value="{TITLE}" />
<Link href="{URL}" rel="stylesheet" />
<Meta content="{DESCRIPTION}" name="description" />
```

In the preceding example, placeholders for `{TITLE}`, `{URL}`, and `{DESCRIPTION}` are string values, Razor variables, or Razor expressions.

The following characteristics apply:

- Server-side prerendering is supported.
- The `Value` parameter is the only valid parameter for the `Title` component.
- HTML attributes provided to the `Meta` and `Link` components are captured in additional attributes and passed through to the rendered HTML tag.
- For multiple `Title` components, the title of the page reflects the `Value` of the last `Title` component rendered.
- If multiple `Meta` or `Link` components are included with identical attributes, there's exactly one HTML tag rendered per `Meta` or `Link` component. Two `Meta` or `Link` components can't refer to the same rendered HTML tag.
- Changes to the parameters of existing `Meta` or `Link` components are reflected in their rendered HTML tags.
- When the `Link` or `Meta` components are no longer rendered and thus disposed by the framework, their rendered HTML tags are removed.

When one of the framework components is used in a child component, the rendered HTML tag influences any other child component of the parent component as long as the child component containing the framework component is rendered. The distinction between using the one of these framework components in a child component and placing a an HTML tag in `wwwroot/index.html` or `Pages/_Host.cshtml` is that a framework component's rendered HTML tag:

- Can be modified by application state. A hard-coded HTML tag can't be modified by application state.
- Is removed from the HTML `<head>` when the parent component is no longer rendered.

Static files

This section applies to Blazor Server.

To create additional file mappings with a FileExtensionContentTypeProvider or configure other StaticFileOptions, use **one** of the following approaches. In the following examples, the `{EXTENSION}` placeholder is the file extension, and the `{CONTENT TYPE}` placeholder is the content type.

- Configure options through dependency injection (DI) in `Startup.ConfigureServices` (`Startup.cs`) using StaticFileOptions:

```
using Microsoft.AspNetCore.StaticFiles;

...

var provider = new FileExtensionContentTypeProvider();
provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

services.Configure<StaticFileOptions>(options =>
{
    options.ContentTypeProvider = provider;
});
```

  Because this approach configures the same file provider used to serve `blazor.server.js`, make sure that your custom configuration doesn't interfere with serving `blazor.server.js`. For example, don't remove the mapping for JavaScript files by configuring the provider with `provider.Mappings.Remove(".js")`.

- Use two calls to UseStaticFiles in `Startup.Configure` (`Startup.cs`):
  - Configure the custom file provider in the first call with StaticFileOptions.
  - The second middleware serves `blazor.server.js`, which uses the default static files configuration provided by the Blazor framework.

```
using Microsoft.AspNetCore.StaticFiles;

...

var provider = new FileExtensionContentTypeProvider();
provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

app.UseStaticFiles(new StaticFileOptions { ContentTypeProvider = provider });
app.UseStaticFiles();
```

Additional resources

- Logging in .NET Core and ASP.NET Core

Create and use ASP.NET Core Razor components

9/22/2020 • 21 minutes to read • Edit Online

By Luke Latham, Daniel Roth, and Tobias Bartsch

View or download sample code (how to download)

Blazor apps are built using *components*. A component is a self-contained chunk of user interface (UI), such as a page, dialog, or form. A component includes HTML markup and the processing logic required to inject data or respond to UI events. Components are flexible and lightweight. They can be nested, reused, and shared among projects.

Component classes

Components are implemented in Razor component files (`.razor`) using a combination of C# and HTML markup. A component in Blazor is formally referred to as a *Razor component*.

Razor syntax

Razor components in Blazor apps extensively use Razor syntax. If you aren't familiar with the Razor markup language, we recommend reading razor syntax reference for ASP.NET Core before proceeding.

When accessing the content on Razor syntax, pay special attention to the following sections:

- Directives: `@`-prefixed reserved keywords that typically change the way component markup is parsed or function.
- Directive attributes: `@`-prefixed reserved keywords that typically change the way component elements are parsed or function.

Names

A component's name must start with an uppercase character. For example, `MyCoolComponent.razor` is valid, and `myCoolComponent.razor` is invalid.

Routing

Routing in Blazor is achieved by providing a route template to each accessible component in the app. When a Razor file with an `@page` directive is compiled, the generated class is given a RouteAttribute specifying the route template. At runtime, the router looks for component classes with a RouteAttribute and renders whichever component has a route template that matches the requested URL. For more information, see ASP.NET Core Blazor routing.

```
@page "/ParentComponent"

...
```

Markup

The UI for a component is defined using HTML. Dynamic rendering logic (for example, loops, conditionals, expressions) is added using an embedded C# syntax called *Razor*. When an app is compiled, the HTML markup and C# rendering logic are converted into a component class. The name of the generated class matches the name of the file.

Members of the component class are defined in an `@code` block. In the `@code` block, component state (properties, fields) is specified with methods for event handling or for defining other component logic. More than one `@code`

block is permissible.

Component members can be used as part of the component's rendering logic using C# expressions that start with `@`. For example, a C# field is rendered by prefixing `@` to the field name. The following example evaluates and renders:

- `headingFontStyle` to the CSS property value for `font-style`.
- `headingText` to the content of the `<h1>` element.

```
<h1 style="font-style:@headingFontStyle">@headingText</h1>

@code {
    private string headingFontStyle = "italic";
    private string headingText = "Put on your new Blazor!";
}
```

After the component is initially rendered, the component regenerates its render tree in response to events. Blazor then compares the new render tree against the previous one and applies any modifications to the browser's Document Object Model (DOM).

Components are ordinary C# classes and can be placed anywhere within a project. Components that produce webpages usually reside in the `Pages` folder. Non-page components are frequently placed in the `Shared` folder or a custom folder added to the project.

Namespaces

Typically, a component's namespace is derived from the app's root namespace and the component's location (folder) within the app. If the app's root namespace is `BlazorSample` and the `Counter` component resides in the `Pages` folder:

- The `Counter` component's namespace is `BlazorSample.Pages`.
- The fully qualified type name of the component is `BlazorSample.Pages.Counter`.

For custom folders that hold components, add a `@using` directive to the parent component or to the app's `_Imports.razor` file. The following example makes components in the `Components` folder available:

```
@using BlazorSample.Components
```

Components can also be referenced using their fully qualified names, which doesn't require the `@using` directive:

```
<BlazorSample.Components.MyComponent />
```

The namespace of a component authored with Razor is based on (in priority order):

- `@namespace` designation in Razor file (`.razor`) markup (`@namespace BlazorSample.MyNamespace`).
- The project's `RootNamespace` in the project file (`<RootNamespace>BlazorSample</RootNamespace>`).
- The project name, taken from the project file's file name (`.csproj`), and the path from the project root to the component. For example, the framework resolves `{PROJECT ROOT}/Pages/Index.razor` (`BlazorSample.csproj`) to the namespace `BlazorSample.Pages`. Components follow C# name binding rules. For the `Index` component in this example, the components in scope are all of the components:
  - In the same folder, `Pages`.
  - The components in the project's root that don't explicitly specify a different namespace.

> **NOTE**
>
> The `global::` qualification isn't supported.
>
> Importing components with aliased `using` statements (for example, `@using Foo = Bar`) isn't supported.
>
> Partially qualified names aren't supported. For example, adding `@using BlazorSample` and referencing the `NavMenu` component (`NavMenu.razor`) with `<Shared.NavMenu></Shared.NavMenu>` isn't supported.

Partial class support

Razor components are generated as partial classes. Razor components are authored using either of the following approaches:

- C# code is defined in an `@code` block with HTML markup and Razor code in a single file. Blazor templates define their Razor components using this approach.
- C# code is placed in a code-behind file defined as a partial class.

The following example shows the default `Counter` component with an `@code` block in an app generated from a Blazor template. HTML markup, Razor code, and C# code are in the same file:

`Pages/Counter.razor`:

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    void IncrementCount()
    {
        currentCount++;
    }
}
```

The `Counter` component can also be created using a code-behind file with a partial class:

`Pages/Counter.razor`:

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

`Counter.razor.cs`:

```
namespace BlazorSample.Pages
{
    public partial class Counter
    {
        private int currentCount = 0;

        void IncrementCount()
        {
            currentCount++;
        }
    }
}
```

Add any required namespaces to the partial class file as needed. Typical namespaces used by Razor components include:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Components.Forms;
using Microsoft.AspNetCore.Components.Routing;
using Microsoft.AspNetCore.Components.Web;
```

IMPORTANT

`@using` directives in the `_Imports.razor` file are only applied to Razor files (`.razor`), not C# files (`.cs`).

Specify a base class

The `@inherits` directive can be used to specify a base class for a component. The following example shows how a component can inherit a base class, `BlazorRocksBase`, to provide the component's properties and methods. The base class should derive from ComponentBase.

`Pages/BlazorRocks.razor`:

```
@page "/BlazorRocks"
@inherits BlazorRocksBase

<h1>@BlazorRocksText</h1>
```

`BlazorRocksBase.cs`:

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample
{
    public class BlazorRocksBase : ComponentBase
    {
        public string BlazorRocksText { get; set; } =
            "Blazor rocks the browser!";
    }
}
```

Use components

Components can include other components by declaring them using HTML element syntax. The markup for using a component looks like an HTML tag where the name of the tag is the component type.

The following markup in `Pages/Index.razor` renders a `HeadingComponent` instance:

```
<HeadingComponent />
```

`Components/HeadingComponent.razor`:

```
@using System.Globalization

<h1 style="font-style:@headingFontStyle">@headingText</h1>

<form>
    <div>
        <label class="form-check-label">
            <input type="checkbox" id="italicsCheck"
               @bind="italicsCheck" />
            Use italics
        </label>
    </div>

    <button type="button" class="btn btn-primary" @onclick="UpdateHeading">
        Update heading
    </button>
</form>

@code {
    private static TextInfo tinfo = CultureInfo.CurrentCulture.TextInfo;
    private string headingText =
        tinfo.ToTitleCase("welcome to blazor!");
    private string headingFontStyle = "normal";
    private bool italicsCheck = false;

    public void UpdateHeading()
    {
        headingFontStyle = italicsCheck ? "italic" : "normal";
    }
}
```

If a component contains an HTML element with an uppercase first letter that doesn't match a component name, a warning is emitted indicating that the element has an unexpected name. Adding an `@using` directive for the component's namespace makes the component available, which resolves the warning.

Parameters

Route parameters

Components can receive route parameters from the route template provided in the `@page` directive. The router uses route parameters to populate the corresponding component parameters.

`Pages/RouteParameter.razor`:

```
@page "/RouteParameter"
@page "/RouteParameter/{text}"

<h1>Blazor is @Text!</h1>

@code {
    [Parameter]
    public string Text { get; set; }

    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

Optional parameters aren't supported, so two `@page` directives are applied in the preceding example. The first permits navigation to the component without a parameter. The second `@page` directive receives the `{text}` route parameter and assigns the value to the `Text` property.

For information on catch-all route parameters (`{*pageRoute}`), which capture paths across multiple folder boundaries, see ASP.NET Core Blazor routing.

Component parameters

Components can have *component parameters*, which are defined using public properties on the component class with the `[Parameter]` attribute. Use attributes to specify arguments for a component in markup.

`Components/ChildComponent.razor` :

```
<div class="panel panel-default">
    <div class="panel-heading">@Title</div>
    <div class="panel-body">@ChildContent</div>

    <button class="btn btn-primary" @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

In the following example from the sample app, the `ParentComponent` sets the value of the `Title` property of the `ChildComponent` .

`Pages/ParentComponent.razor` :

```
@page "/ParentComponent"

<h1>Parent-child example</h1>

<ChildComponent Title="Panel Title from Parent"
                OnClickCallback="@ShowMessage">
    Content of the child component is supplied
    by the parent component.
</ChildComponent>
```

WARNING

Don't create components that write to their own *component parameters*, use a private field instead. For more information, see the Overwritten parameters section.

Child content

Components can set the content of another component. The assigning component provides the content between the tags that specify the receiving component.

In the following example, the `ChildComponent` has a `ChildContent` property that represents a RenderFragment, which represents a segment of UI to render. The value of `ChildContent` is positioned in the component's markup where the content should be rendered. The value of `ChildContent` is received from the parent component and rendered inside the Bootstrap panel's `panel-body`.

`Components/ChildComponent.razor`:

```
<div class="panel panel-default">
    <div class="panel-heading">@Title</div>
    <div class="panel-body">@ChildContent</div>

    <button class="btn btn-primary" @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

NOTE

The property receiving the RenderFragment content must be named `ChildContent` by convention.

The `ParentComponent` in the sample app can provide content for rendering the `ChildComponent` by placing the content inside the `<ChildComponent>` tags.

`Pages/ParentComponent.razor`:

```
@page "/ParentComponent"

<h1>Parent-child example</h1>

<ChildComponent Title="Panel Title from Parent"
                OnClickCallback="@ShowMessage">
    Content of the child component is supplied
    by the parent component.
</ChildComponent>
```

Due to the way that Blazor renders child content, rendering components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the child component's content:

```
@for (int c = 0; c < 10; c++)
{
    var current = c;
    <ChildComponent Param1="@c">
        Child Content: Count: @current
    </ChildComponent>
}
```

Alternatively, use a `foreach` loop with Enumerable.Range:

```
@foreach(var c in Enumerable.Range(0,10))
{
    <ChildComponent Param1="@c">
        Child Content: Count: @c
    </ChildComponent>
}
```

Attribute splatting and arbitrary parameters

Components can capture and render additional attributes in addition to the component's declared parameters. Additional attributes can be captured in a dictionary and then *splatted* onto an element when the component is rendered using the `@attributes` Razor directive. This scenario is useful when defining a component that produces a markup element that supports a variety of customizations. For example, it can be tedious to define attributes separately for an `<input>` that supports many parameters.

In the following example, the first `<input>` element (`id="useIndividualParams"`) uses individual component parameters, while the second `<input>` element (`id="useAttributesDict"`) uses attribute splatting:

```
<input id="useIndividualParams"
       maxlength="@maxlength"
       placeholder="@placeholder"
       required="@required"
       size="@size" />

<input id="useAttributesDict"
       @attributes="InputAttributes" />

@code {
    public string maxlength = "10";
    public string placeholder = "Input placeholder text";
    public string required = "required";
    public string size = "50";

    public Dictionary<string, object> InputAttributes { get; set; } =
        new Dictionary<string, object>()
        {
            { "maxlength", "10" },
            { "placeholder", "Input placeholder text" },
            { "required", "required" },
            { "size", "50" }
        };
}
```

The type of the parameter must implement `IEnumerable<KeyValuePair<string, object>>` or `IReadOnlyDictionary<string, object>` with string keys.

The rendered `<input>` elements using both approaches is identical:

```
<input id="useIndividualParams"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">

<input id="useAttributesDict"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">
```

To accept arbitrary attributes, define a component parameter using the `[Parameter]` attribute with the CaptureUnmatchedValues property set to `true`:

```
@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> InputAttributes { get; set; }
}
```

The CaptureUnmatchedValues property on `[Parameter]` allows the parameter to match all attributes that don't match any other parameter. A component can only define a single parameter with CaptureUnmatchedValues. The property type used with CaptureUnmatchedValues must be assignable from `Dictionary<string, object>` with string keys. `IEnumerable<KeyValuePair<string, object>>` or `IReadOnlyDictionary<string, object>` are also options in this scenario.

The position of `@attributes` relative to the position of element attributes is important. When `@attributes` are splatted on the element, the attributes are processed from right to left (last to first). Consider the following example of a component that consumes a `Child` component:

`ParentComponent.razor`:

```
<ChildComponent extra="10" />
```

`ChildComponent.razor`:

```
<div @attributes="AdditionalAttributes" extra="5" />

[Parameter(CaptureUnmatchedValues = true)]
public IDictionary<string, object> AdditionalAttributes { get; set; }
```

The `Child` component's `extra` attribute is set to the right of `@attributes`. The `Parent` component's rendered `<div>` contains `extra="5"` when passed through the additional attribute because the attributes are processed right to left (last to first):

```
<div extra="5" />
```

In the following example, the order of `extra` and `@attributes` is reversed in the `Child` component's `<div>`:

`ParentComponent.razor`:

```
<ChildComponent extra="10" />
```

`ChildComponent.razor`:

```
<div extra="5" @attributes="AdditionalAttributes" />

[Parameter(CaptureUnmatchedValues = true)]
public IDictionary<string, object> AdditionalAttributes { get; set; }
```

The rendered `<div>` in the `Parent` component contains `extra="10"` when passed through the additional attribute:

```
<div extra="10" />
```

Capture references to components

Component references provide a way to reference a component instance so that you can issue commands to that instance, such as `Show` or `Reset`. To capture a component reference:

- Add an `@ref` attribute to the child component.
- Define a field with the same type as the child component.

```
<CustomLoginDialog @ref="loginDialog" ... />

@code {
    private CustomLoginDialog loginDialog;

    private void OnSomething()
    {
        loginDialog.Show();
    }
}
```

When the component is rendered, the `loginDialog` field is populated with the `MyLoginDialog` child component instance. You can then invoke .NET methods on the component instance.

IMPORTANT

The `loginDialog` variable is only populated after the component is rendered and its output includes the `MyLoginDialog` element. Until the component is rendered, there's nothing to reference.

To manipulate components references after the component has finished rendering, use the `OnAfterRenderAsync` or `OnAfterRender` methods.

To use a reference variable with an event handler, use a lambda expression or assign the event handler delegate in the `OnAfterRenderAsync` or `OnAfterRender` methods. This ensures that the reference variable is assigned before the event handler is assigned.

```
<button type="button"
    @onclick="@(() => loginDialog.DoSomething())">Do Something</button>

<MyLoginDialog @ref="loginDialog" ... />

@code {
    private MyLoginDialog loginDialog;
}
```

To reference components in a loop, see Capture references to multiple similar child-components (dotnet/aspnetcore #13358).

While capturing component references use a similar syntax to capturing element references, it isn't a JavaScript interop feature. Component references aren't passed to JavaScript code. Component references are only used in .NET code.

NOTE

Do **not** use component references to mutate the state of child components. Instead, use normal declarative parameters to pass data to child components. Use of normal declarative parameters result in child components that rerender at the correct times automatically.

Synchronization context

Blazor uses a synchronization context (SynchronizationContext) to enforce a single logical thread of execution. A component's lifecycle methods and any event callbacks that are raised by Blazor are executed on the synchronization context.

Blazor Server's synchronization context attempts to emulate a single-threaded environment so that it closely matches the WebAssembly model in the browser, which is single threaded. At any given point in time, work is performed on exactly one thread, giving the impression of a single logical thread. No two operations execute

concurrently.

Avoid thread-blocking calls

Generally, don't call the following methods. The following methods block the thread and thus block the app from resuming work until the underlying Task is complete:

- Result
- Wait
- WaitAny
- WaitAll
- Sleep
- GetResult

Invoke component methods externally to update state

In the event a component must be updated based on an external event, such as a timer or other notifications, use the `InvokeAsync` method, which dispatches to Blazor's synchronization context. For example, consider a *notifier service* that can notify any listening component of the updated state:

```
public class NotifierService
{
    // Can be called from anywhere
    public async Task Update(string key, int value)
    {
        if (Notify != null)
        {
            await Notify.Invoke(key, value);
        }
    }

    public event Func<string, int, Task> Notify;
}
```

Register the `NotifierService`:

- In Blazor WebAssembly, register the service as singleton in `Program.Main`:

  ```
  builder.Services.AddSingleton<NotifierService>();
  ```

- In Blazor Server, register the service as scoped in `Startup.ConfigureServices`:

  ```
  services.AddScoped<NotifierService>();
  ```

Use the `NotifierService` to update a component:

```
@page "/"
@inject NotifierService Notifier
@implements IDisposable

<p>Last update: @lastNotification.key = @lastNotification.value</p>

@code {
    private (string key, int value) lastNotification;

    protected override void OnInitialized()
    {
        Notifier.Notify += OnNotify;
    }

    public async Task OnNotify(string key, int value)
    {
        await InvokeAsync(() =>
        {
            lastNotification = (key, value);
            StateHasChanged();
        });
    }

    public void Dispose()
    {
        Notifier.Notify -= OnNotify;
    }
}
```

In the preceding example, `NotifierService` invokes the component's `OnNotify` method outside of Blazor's synchronization context. `InvokeAsync` is used to switch to the correct context and queue a render.

Use @key to control the preservation of elements and components

When rendering a list of elements or components and the elements or components subsequently change, Blazor's diffing algorithm must decide which of the previous elements or components can be retained and how model objects should map to them. Normally, this process is automatic and can be ignored, but there are cases where you may want to control the process.

Consider the following example:

```
@foreach (var person in People)
{
    <DetailsEditor Details="@person.Details" />
}

@code {
    [Parameter]
    public IEnumerable<Person> People { get; set; }
}
```

The contents of the `People` collection may change with inserted, deleted, or re-ordered entries. When the component rerenders, the `<DetailsEditor>` component may change to receive different `Details` parameter values. This may cause more complex rerendering than expected. In some cases, rerendering can lead to visible behavior differences, such as lost element focus.

The mapping process can be controlled with the `@key` directive attribute. `@key` causes the diffing algorithm to guarantee preservation of elements or components based on the key's value:

```
@foreach (var person in People)
{
    <DetailsEditor @key="person" Details="@person.Details" />
}

@code {
    [Parameter]
    public IEnumerable<Person> People { get; set; }
}
```

When the `People` collection changes, the diffing algorithm retains the association between `<DetailsEditor>` instances and `person` instances:

- If a `Person` is deleted from the `People` list, only the corresponding `<DetailsEditor>` instance is removed from the UI. Other instances are left unchanged.
- If a `Person` is inserted at some position in the list, one new `<DetailsEditor>` instance is inserted at that corresponding position. Other instances are left unchanged.
- If `Person` entries are re-ordered, the corresponding `<DetailsEditor>` instances are preserved and re-ordered in the UI.

In some scenarios, use of `@key` minimizes the complexity of rerendering and avoids potential issues with stateful parts of the DOM changing, such as focus position.

> **IMPORTANT**
> Keys are local to each container element or component. Keys aren't compared globally across the document.

When to use @key

Typically, it makes sense to use `@key` whenever a list is rendered (for example, in a foreach block) and a suitable value exists to define the `@key`.

You can also use `@key` to prevent Blazor from preserving an element or component subtree when an object changes:

```
<div @key="currentPerson">
    ... content that depends on currentPerson ...
</div>
```

If `@currentPerson` changes, the `@key` attribute directive forces Blazor to discard the entire `<div>` and its descendants and rebuild the subtree within the UI with new elements and components. This can be useful if you need to guarantee that no UI state is preserved when `@currentPerson` changes.

When not to use @key

There's a performance cost when diffing with `@key`. The performance cost isn't large, but only specify `@key` if controlling the element or component preservation rules benefit the app.

Even if `@key` isn't used, Blazor preserves child element and component instances as much as possible. The only advantage to using `@key` is control over *how* model instances are mapped to the preserved component instances, instead of the diffing algorithm selecting the mapping.

What values to use for @key

Generally, it makes sense to supply one of the following kinds of value for `@key`:

- Model object instances (for example, a `Person` instance as in the earlier example). This ensures preservation based on object reference equality.

- Unique identifiers (for example, primary key values of type `int`, `string`, or `Guid`).

Ensure that values used for `@key` don't clash. If clashing values are detected within the same parent element, Blazor throws an exception because it can't deterministically map old elements or components to new elements or components. Only use distinct values, such as object instances or primary key values.

Overwritten parameters

New parameter values are supplied, typically overwriting existing ones, when the parent component rerenders.

Consider the following `Expander` component that:

- Renders child content.
- Toggles showing child content with a component parameter.

```
<div @onclick="@Toggle" class="card bg-light mb-3" style="width:30rem">
    <div class="card-body">
        <h2 class="card-title">Toggle (<code>Expanded</code> = @Expanded)</h2>

        @if (Expanded)
        {
            <p class="card-text">@ChildContent</p>
        }
    </div>
</div>

@code {
    [Parameter]
    public bool Expanded { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    private void Toggle()
    {
        Expanded = !Expanded;
    }
}
```

The `Expander` component is added to a parent component that may call StateHasChanged:

```
@page "/expander"

<Expander Expanded="true">
    Expander 1 content
</Expander>

<Expander Expanded="true" />

<button @onclick="StateHasChanged">
    Call StateHasChanged
</button>
```

Initially, the `Expander` components behave independently when their `Expanded` properties are toggled. The child components maintain their states as expected. When StateHasChanged is called in the parent, the `Expanded` parameter of the first child component is reset back to its initial value (`true`). The second `Expander` component's `Expanded` value isn't reset because no child content is rendered in the second component.

To maintain state in the preceding scenario, use a *private field* in the `Expander` component to maintain its toggled state.

The following revised `Expander` component:

- Accepts the `Expanded` component parameter value from the parent.
- Assigns the component parameter value to a *private field* (`expanded`) in the OnInitialized event.
- Uses the private field to maintain its internal toggle state.

```
<div @onclick="@Toggle" class="card bg-light mb-3" style="width:30rem">
    <div class="card-body">
        <h2 class="card-title">Toggle (<code>expanded</code> = @expanded)</h2>

        @if (expanded)
        {
            <p class="card-text">@ChildContent</p>
        }
    </div>
</div>

@code {
    private bool expanded;

    [Parameter]
    public bool Expanded { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    protected override void OnInitialized()
    {
        expanded = Expanded;
    }

    private void Toggle()
    {
        expanded = !expanded;
    }
}
```

Apply an attribute

Attributes can be applied to Razor components with the `@attribute` directive. The following example applies the `[Authorize]` attribute to the component class:

```
@page "/"
@attribute [Authorize]
```

Conditional HTML element attributes

HTML element attributes are conditionally rendered based on the .NET value. If the value is `false` or `null`, the attribute isn't rendered. If the value is `true`, the attribute is rendered minimized.

In the following example, `IsCompleted` determines if `checked` is rendered in the element's markup:

```
<input type="checkbox" checked="@IsCompleted" />

@code {
    [Parameter]
    public bool IsCompleted { get; set; }
}
```

If `IsCompleted` is `true`, the check box is rendered as:

```
<input type="checkbox" checked />
```

If `IsCompleted` is `false`, the check box is rendered as:

```
<input type="checkbox" />
```

For more information, see razor syntax reference for ASP.NET Core.

> **WARNING**
> Some HTML attributes, such as `aria-pressed`, don't function properly when the .NET type is a `bool`. In those cases, use a `string` type instead of a `bool`.

Raw HTML

Strings are normally rendered using DOM text nodes, which means that any markup they may contain is ignored and treated as literal text. To render raw HTML, wrap the HTML content in a `MarkupString` value. The value is parsed as HTML or SVG and inserted into the DOM.

> **WARNING**
> Rendering raw HTML constructed from any untrusted source is a **security risk** and should be avoided!

The following example shows using the `MarkupString` type to add a block of static HTML content to the rendered output of a component:

```
@((MarkupString)myMarkup)

@code {
    private string myMarkup =
        "<p class='markup'>This is a <em>markup string</em>.</p>";
}
```

Razor templates

Render fragments can be defined using Razor template syntax. Razor templates are a way to define a UI snippet and assume the following format:

```
@<{HTML tag}>...</{HTML tag}>
```

The following example illustrates how to specify RenderFragment and RenderFragment<TValue> values and render templates directly in a component. Render fragments can also be passed as arguments to templated components.

```
@timeTemplate

@petTemplate(new Pet { Name = "Rex" })

@code {
    private RenderFragment timeTemplate = @<p>The time is @DateTime.Now.</p>;
    private RenderFragment<Pet> petTemplate = (pet) => @<p>Pet: @pet.Name</p>;

    private class Pet
    {
        public string Name { get; set; }
    }
}
```

Rendered output of the preceding code:

```
<p>The time is 10/04/2018 01:26:52.</p>

<p>Pet: Rex</p>
```

Static assets

Blazor follows the convention of ASP.NET Core apps placing static assets under the project's `web root (wwwroot)` folder.

Use a base-relative path (`/`) to refer to the web root for a static asset. In the following example, `logo.png` is physically located in the `{PROJECT ROOT}/wwwroot/images` folder:

```
<img alt="Company logo" src="/images/logo.png" />
```

Razor components do **not** support tilde-slash notation (`~/`).

For information on setting an app's base path, see Host and deploy ASP.NET Core Blazor.

Tag Helpers aren't supported in components

`Tag Helpers` aren't supported in Razor components (`.razor` files). To provide Tag Helper-like functionality in Blazor, create a component with the same functionality as the Tag Helper and use the component instead.

Scalable Vector Graphics (SVG) images

Since Blazor renders HTML, browser-supported images, including Scalable Vector Graphics (SVG) images (`.svg`), are supported via the `<img>` tag:

```
<img alt="Example image" src="some-image.svg" />
```

Similarly, SVG images are supported in the CSS rules of a stylesheet file (`.css`):

```
.my-element {
    background-image: url("some-image.svg");
}
```

However, inline SVG markup isn't supported in all scenarios. If you place an `<svg>` tag directly into a component file (`.razor`), basic image rendering is supported but many advanced scenarios aren't yet supported. For example,

`<use>` tags aren't currently respected, and `@bind` can't be used with some SVG tags. For more information, see SVG support in Blazor (dotnet/aspnetcore #18271).

Additional resources

- Threat mitigation guidance for ASP.NET Core Blazor Server: Includes guidance on building Blazor Server apps that must contend with resource exhaustion.

ASP.NET Core Blazor templates

9/22/2020 • 3 minutes to read • Edit Online

By Daniel Roth and Luke Latham

The Blazor framework provides templates to develop apps for each of the Blazor hosting models:

- Blazor WebAssembly (`blazorwasm`)
- Blazor Server (`blazorserver`)

For more information on Blazor's hosting models, see ASP.NET Core Blazor hosting models.

Template options are available by passing the `--help` option to the `dotnet new` CLI command:

```
dotnet new blazorwasm --help
dotnet new blazorserver --help
```

Blazor project structure

The following files and folders make up a Blazor app generated from a Blazor project template:

- `Program.cs` : The app's entry point that sets up the:
  - ASP.NET Core host (Blazor Server)
  - WebAssembly host (Blazor WebAssembly): The code in this file is unique to apps created from the Blazor WebAssembly template (`blazorwasm`).
    - The `App` component is the root component of the app. The `App` component is specified as the `app` DOM element (`<app>...</app>`) to the root component collection (`builder.RootComponents.Add<App>("app")`).
    - Services are added and configured (for example, `builder.Services.AddSingleton<IMyDependency, MyDependency>()`).
- `Startup.cs` (Blazor Server): Contains the app's startup logic. The `Startup` class defines two methods:
  - `ConfigureServices` : Configures the app's dependency injection (DI) services. In Blazor Server apps, services are added by calling AddServerSideBlazor, and the `WeatherForecastService` is added to the service container for use by the example `FetchData` component.
  - `Configure` : Configures the app's request handling pipeline:
    - MapBlazorHub is called to set up an endpoint for the real-time connection with the browser. The connection is created with SignalR, which is a framework for adding real-time web functionality to apps.
    - `MapFallbackToPage("/_Host")` is called to set up the root page of the app (`Pages/_Host.cshtml`) and enable navigation.
- `wwwroot/index.html` (Blazor WebAssembly): The root page of the app implemented as an HTML page:
  - When any page of the app is initially requested, this page is rendered and returned in the response.
  - The page specifies where the root `App` component is rendered. The component is rendered at the location of the `app` DOM element (`<app>...</app>`).
  - The `_framework/blazor.webassembly.js` JavaScript file is loaded, which:
    - Downloads the .NET runtime, the app, and the app's dependencies.

    - Initializes the runtime to run the app.
- `App.razor`: The root component of the app that sets up client-side routing using the Router component. The Router component intercepts browser navigation and renders the page that matches the requested address.
- `Pages` folder: Contains the routable components/pages (`.razor`) that make up the Blazor app and the root Razor page of a Blazor Server app. The route for each page is specified using the `@page` directive. The template includes the following:
    - `_Host.cshtml` (Blazor Server): The root page of the app implemented as a Razor Page:
        - When any page of the app is initially requested, this page is rendered and returned in the response.
        - The `_framework/blazor.server.js` JavaScript file is loaded, which sets up the real-time SignalR connection between the browser and the server.
        - The Host page specifies where the root `App` component (`App.razor`) is rendered.
    - `Counter` (`Pages/Counter.razor`): Implements the Counter page.
    - `Error` (`Error.razor`, Blazor Server app only): Rendered when an unhandled exception occurs in the app.
    - `FetchData` (`Pages/FetchData.razor`): Implements the Fetch data page.
    - `Index` (`Pages/Index.razor`): Implements the Home page.
- `Properties/launchSettings.json`: Holds development environment configuration.
- `Shared` folder: Contains other UI components (`.razor`) used by the app:
    - `MainLayout` (`MainLayout.razor`): The app's layout component.
    - `NavMenu` (`NavMenu.razor`): Implements sidebar navigation. Includes the `NavLink` component (NavLink), which renders navigation links to other Razor components. The NavLink component automatically indicates a selected state when its component is loaded, which helps the user understand which component is currently displayed.
- `_Imports.razor`: Includes common Razor directives to include in the app's components (`.razor`), such as `@using` directives for namespaces.
- `Data` folder (Blazor Server): Contains the `WeatherForecast` class and implementation of the `WeatherForecastService` that provide example weather data to the app's `FetchData` component.
- `wwwroot`: The Web Root folder for the app containing the app's public static assets.
- `appsettings.json`: Holds configuration settings for the app. In a Blazor WebAssembly app, the app settings file is located in the `wwwroot` folder. In a Blazor Server app, the app settings file is located at the project root.

Secure ASP.NET Core Blazor WebAssembly

9/22/2020 • 5 minutes to read • Edit Online

By Javier Calvarro Nelson

Blazor WebAssembly apps are secured in the same manner as Single Page Applications (SPAs). There are several approaches for authenticating users to SPAs, but the most common and comprehensive approach is to use an implementation based on the OAuth 2.0 protocol, such as OpenID Connect (OIDC).

Authentication library

Blazor WebAssembly supports authenticating and authorizing apps using OIDC via the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` library. The library provides a set of primitives for seamlessly authenticating against ASP.NET Core backends. The library integrates ASP.NET Core Identity with API authorization support built on top of Identity Server. The library can authenticate against any third-party Identity Provider (IP) that supports OIDC, which are called OpenID Providers (OP).

The authentication support in Blazor WebAssembly is built on top of the `oidc-client.js` library, which is used to handle the underlying authentication protocol details.

Other options for authenticating SPAs exist, such as the use of SameSite cookies. However, the engineering design of Blazor WebAssembly is settled on OAuth and OIDC as the best option for authentication in Blazor WebAssembly apps. Token-based authentication based on JSON Web Tokens (JWTs) was chosen over cookie-based authentication for functional and security reasons:

- Using a token-based protocol offers a smaller attack surface area, as the tokens aren't sent in all requests.
- Server endpoints don't require protection against Cross-Site Request Forgery (CSRF) because the tokens are sent explicitly. This allows you to host Blazor WebAssembly apps alongside MVC or Razor pages apps.
- Tokens have narrower permissions than cookies. For example, tokens can't be used to manage the user account or change a user's password unless such functionality is explicitly implemented.
- Tokens have a short lifetime, one hour by default, which limits the attack window. Tokens can also be revoked at any time.
- Self-contained JWTs offer guarantees to the client and server about the authentication process. For example, a client has the means to detect and validate that the tokens it receives are legitimate and were emitted as part of a given authentication process. If a third party attempts to switch a token in the middle of the authentication process, the client can detect the switched token and avoid using it.
- Tokens with OAuth and OIDC don't rely on the user agent behaving correctly to ensure that the app is secure.
- Token-based protocols, such as OAuth and OIDC, allow for authenticating and authorizing hosted and standalone apps with the same set of security characteristics.

Authentication process with OIDC

The `Microsoft.AspNetCore.Components.WebAssembly.Authentication` library offers several primitives to implement authentication and authorization using OIDC. In broad terms, authentication works as follows:

- When an anonymous user selects the login button or requests a page with the `[Authorize]` attribute applied, the user is redirected to the app's login page (`/authentication/login`).
- In the login page, the authentication library prepares for a redirect to the authorization endpoint. The authorization endpoint is outside of the Blazor WebAssembly app and can be hosted at a separate origin. The endpoint is responsible for determining whether the user is authenticated and for issuing one or more tokens in

response. The authentication library provides a login callback to receive the authentication response.
    - If the user isn't authenticated, the user is redirected to the underlying authentication system, which is usually ASP.NET Core Identity.
    - If the user was already authenticated, the authorization endpoint generates the appropriate tokens and redirects the browser back to the login callback endpoint (`/authentication/login-callback`).
- When the Blazor WebAssembly app loads the login callback endpoint (`/authentication/login-callback`), the authentication response is processed.
    - If the authentication process completes successfully, the user is authenticated and optionally sent back to the original protected URL that the user requested.
    - If the authentication process fails for any reason, the user is sent to the login failed page (`/authentication/login-failed`), and an error is displayed.

`Authentication` component

The `Authentication` component (`Pages/Authentication.razor`) handles remote authentication operations and permits the app to:

- Configure app routes for authentication states.
- Set UI content for authentication states.
- Manage authentication state.

Authentication actions, such as registering or signing in a user, are passed to the Blazor framework's RemoteAuthenticatorViewCore<TAuthenticationState> component, which persists and controls state across authentication operations.

For more information and examples, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Authorization

In Blazor WebAssembly apps, authorization checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks or native apps for any operating system.

Always perform authorization checks on the server within any API endpoints accessed by your client-side app.

Require authorization for the entire app

Apply the `[Authorize]` attribute (API documentation) to each Razor component of the app using one of the following approaches:

- Use the `@attribute` directive in the `_Imports.razor` file:

```
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
```

- Add the attribute to each Razor component in the `Pages` folder.

> **NOTE**
>
> Setting an AuthorizationOptions.FallbackPolicy to a policy with RequireAuthenticatedUser is **not** supported.

Refresh tokens

Refresh tokens can't be secured client-side in Blazor WebAssembly apps. Therefore, refresh tokens shouldn't be sent to the app for direct use.

Refresh tokens can be maintained and used by the server-side app in a Hosted Blazor WebAssembly solution to access third-party APIs. For more information, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Establish claims for users

Apps often require claims for users based on a web API call to a server. For example, claims are frequently used to establish authorization in an app. In these scenarios, the app requests an access token to access the service and uses the token to obtain the user data for the claims. For examples, see the following resources:

- Additional scenarios: Customize the user
- ASP.NET Core Blazor WebAssembly with Azure Active Directory groups and roles

Implementation guidance

Articles under this *Overview* provide information on authenticating users in Blazor WebAssembly apps against specific providers.

Standalone Blazor WebAssembly apps:

- General guidance for OIDC providers and the WebAssembly Authentication Library
- Microsoft Accounts
- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C

Hosted Blazor WebAssembly apps:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

For further guidance on configuration, see ASP.NET Core Blazor WebAssembly additional security scenarios.

ASP.NET Core Blazor authentication and authorization

9/22/2020 • 10 minutes to read • Edit Online

By Steve Sanderson and Luke Latham

ASP.NET Core supports the configuration and management of security in Blazor apps.

Security scenarios differ between Blazor Server and Blazor WebAssembly apps. Because Blazor Server apps run on the server, authorization checks are able to determine:

- The UI options presented to a user (for example, which menu entries are available to a user).
- Access rules for areas of the app and components.

Blazor WebAssembly apps run on the client. Authorization is *only* used to determine which UI options to show. Since client-side checks can be modified or bypassed by a user, a Blazor WebAssembly app can't enforce authorization access rules.

Razor Pages authorization conventions don't apply to routable Razor components. If a non-routable Razor component is embedded in a page, the page's authorization conventions indirectly affect the Razor component along with the rest of the page's content.

> **NOTE**
> SignInManager<TUser> and UserManager<TUser> aren't supported in Razor components.

Authentication

Blazor uses the existing ASP.NET Core authentication mechanisms to establish the user's identity. The exact mechanism depends on how the Blazor app is hosted, Blazor WebAssembly or Blazor Server.

Blazor WebAssembly authentication

In Blazor WebAssembly apps, authentication checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks or native apps for any operating system.

Add the following:

- A package reference for `Microsoft.AspNetCore.Components.Authorization` to the app's project file.
- The `Microsoft.AspNetCore.Components.Authorization` namespace to the app's `_Imports.razor` file.

To handle authentication, use of a built-in or custom AuthenticationStateProvider service is covered in the following sections.

For more information on creating apps and configuration, see Secure ASP.NET Core Blazor WebAssembly.

Blazor Server authentication

Blazor Server apps operate over a real-time connection that's created using SignalR. Authentication in SignalR-based apps is handled when the connection is established. Authentication can be based on a cookie or some other bearer token.

The built-in AuthenticationStateProvider service for Blazor Server apps obtains authentication state data from

ASP.NET Core's `HttpContext.User`. This is how authentication state integrates with existing ASP.NET Core authentication mechanisms.

For more information on creating apps and configuration, see Secure ASP.NET Core Blazor Server apps.

AuthenticationStateProvider service

AuthenticationStateProvider is the underlying service used by the AuthorizeView component and CascadingAuthenticationState component to get the authentication state.

You don't typically use AuthenticationStateProvider directly. Use the `AuthorizeView` component or `Task<AuthenticationState>` approaches described later in this article. The main drawback to using AuthenticationStateProvider directly is that the component isn't notified automatically if the underlying authentication state data changes.

The AuthenticationStateProvider service can provide the current user's ClaimsPrincipal data, as shown in the following example:

```
@page "/"
@using System.Security.Claims
@using Microsoft.AspNetCore.Components.Authorization
@inject AuthenticationStateProvider AuthenticationStateProvider

<h3>ClaimsPrincipal Data</h3>

<button @onclick="GetClaimsPrincipalData">Get ClaimsPrincipal Data</button>

<p>@_authMessage</p>

@if (_claims.Count() > 0)
{
    <ul>
        @foreach (var claim in _claims)
        {
            <li>@claim.Type: @claim.Value</li>
        }
    </ul>
}

<p>@_surnameMessage</p>

@code {
    private string _authMessage;
    private string _surnameMessage;
    private IEnumerable<Claim> _claims = Enumerable.Empty<Claim>();

    private async Task GetClaimsPrincipalData()
    {
        var authState = await AuthenticationStateProvider.GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity.IsAuthenticated)
        {
            _authMessage = $"{user.Identity.Name} is authenticated.";
            _claims = user.Claims;
            _surnameMessage =
                $"Surname: {user.FindFirst(c => c.Type == ClaimTypes.Surname)?.Value}";
        }
        else
        {
            _authMessage = "The user is NOT authenticated.";
        }
    }
}
```

If `user.Identity.IsAuthenticated` is `true` and because the user is a ClaimsPrincipal, claims can be enumerated and membership in roles evaluated.

For more information on dependency injection (DI) and services, see ASP.NET Core Blazor dependency injection and Dependency injection in ASP.NET Core.

Implement a custom AuthenticationStateProvider

If the app requires a custom provider, implement AuthenticationStateProvider and override `GetAuthenticationStateAsync`:

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
    {
        var identity = new ClaimsIdentity(new[]
        {
            new Claim(ClaimTypes.Name, "mrfibuli"),
        }, "Fake authentication type");

        var user = new ClaimsPrincipal(identity);

        return Task.FromResult(new AuthenticationState(user));
    }
}
```

In a Blazor WebAssembly app, the `CustomAuthStateProvider` service is registered in `Main` of `Program.cs`:

```
using Microsoft.AspNetCore.Components.Authorization;

...

builder.Services.AddScoped<AuthenticationStateProvider, CustomAuthStateProvider>();
```

In a Blazor Server app, the `CustomAuthStateProvider` service is registered in `Startup.ConfigureServices`:

```
using Microsoft.AspNetCore.Components.Authorization;

...

services.AddScoped<AuthenticationStateProvider, CustomAuthStateProvider>();
```

Using the `CustomAuthStateProvider` in the preceding example, all users are authenticated with the username `mrfibuli`.

Expose the authentication state as a cascading parameter

If authentication state data is required for procedural logic, such as when performing an action triggered by the user, obtain the authentication state data by defining a cascading parameter of type `Task<` AuthenticationState `>`:

```
@page "/"

<button @onclick="LogUsername">Log username</button>

<p>@_authMessage</p>

@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private string _authMessage;

    private async Task LogUsername()
    {
        var authState = await authenticationStateTask;
        var user = authState.User;

        if (user.Identity.IsAuthenticated)
        {
            _authMessage = $"{user.Identity.Name} is authenticated.";
        }
        else
        {
            _authMessage = "The user is NOT authenticated.";
        }
    }
}
```

If `user.Identity.IsAuthenticated` is `true`, claims can be enumerated and membership in roles evaluated.

Set up the `Task<` AuthenticationState `>` cascading parameter using the AuthorizeRouteView and CascadingAuthenticationState components in the `App` component (`App.razor`):

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)" />
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

In a Blazor WebAssembly App, add services for options and authorization to `Program.Main`:

```
builder.Services.AddOptions();
builder.Services.AddAuthorizationCore();
```

In a Blazor Server app, services for options and authorization are already present, so no further action is required.

Authorization

After a user is authenticated, *authorization* rules are applied to control what the user can do.

Access is typically granted or denied based on whether:

- A user is authenticated (signed in).

- A user is in a *role*.
- A user has a *claim*.
- A *policy* is satisfied.

Each of these concepts is the same as in an ASP.NET Core MVC or Razor Pages app. For more information on ASP.NET Core security, see the articles under ASP.NET Core Security and Identity.

AuthorizeView component

The AuthorizeView component selectively displays UI depending on whether the user is authorized to see it. This approach is useful when you only need to *display* data for the user and don't need to use the user's identity in procedural logic.

The component exposes a `context` variable of type AuthenticationState, which you can use to access information about the signed-in user:

```
<AuthorizeView>
    <h1>Hello, @context.User.Identity.Name!</h1>
    <p>You can only see this content if you're authenticated.</p>
</AuthorizeView>
```

You can also supply different content for display if the user isn't authenticated:

```
<AuthorizeView>
    <Authorized>
        <h1>Hello, @context.User.Identity.Name!</h1>
        <p>You can only see this content if you're authenticated.</p>
    </Authorized>
    <NotAuthorized>
        <h1>Authentication Failure!</h1>
        <p>You're not signed in.</p>
    </NotAuthorized>
</AuthorizeView>
```

The AuthorizeView component can be used in the `NavMenu` component (`Shared/NavMenu.razor`) to display a list item (`<li>...</li>`) for a `NavLink` component (NavLink), but note that this approach only removes the list item from the rendered output. It doesn't prevent the user from navigating to the component.

The content of `<Authorized>` and `<NotAuthorized>` tags can include arbitrary items, such as other interactive components.

Authorization conditions, such as roles or policies that control UI options or access, are covered in the Authorization section.

If authorization conditions aren't specified, AuthorizeView uses a default policy and treats:

- Authenticated (signed-in) users as authorized.
- Unauthenticated (signed-out) users as unauthorized.

Role-based and policy-based authorization

The AuthorizeView component supports *role-based* or *policy-based* authorization.

For role-based authorization, use the Roles parameter:

```
<AuthorizeView Roles="admin, superuser">
    <p>You can only see this if you're an admin or superuser.</p>
</AuthorizeView>
```

For more information, see Role-based authorization in ASP.NET Core.

For policy-based authorization, use the Policy parameter:

```
<AuthorizeView Policy="content-editor">
    <p>You can only see this if you satisfy the "content-editor" policy.</p>
</AuthorizeView>
```

Claims-based authorization is a special case of policy-based authorization. For example, you can define a policy that requires users to have a certain claim. For more information, see Policy-based authorization in ASP.NET Core.

These APIs can be used in either Blazor Server or Blazor WebAssembly apps.

If neither Roles nor Policy is specified, AuthorizeView uses the default policy.

Content displayed during asynchronous authentication

Blazor allows for authentication state to be determined *asynchronously*. The primary scenario for this approach is in Blazor WebAssembly apps that make a request to an external endpoint for authentication.

While authentication is in progress, AuthorizeView displays no content by default. To display content while authentication occurs, use the `<Authorizing>` tag:

```
<AuthorizeView>
    <Authorized>
        <h1>Hello, @context.User.Identity.Name!</h1>
        <p>You can only see this content if you're authenticated.</p>
    </Authorized>
    <Authorizing>
        <h1>Authentication in progress</h1>
        <p>You can only see this content while authentication is in progress.</p>
    </Authorizing>
</AuthorizeView>
```

This approach isn't normally applicable to Blazor Server apps. Blazor Server apps know the authentication state as soon as the state is established. Authorizing content can be provided in a Blazor Server app's AuthorizeView component, but the content is never displayed.

[Authorize] attribute

The `[Authorize]` attribute can be used in Razor components:

```
@page "/"
@attribute [Authorize]

You can only see this if you're signed in.
```

> **IMPORTANT**
>
> Only use `[Authorize]` on `@page` components reached via the Blazor Router. Authorization is only performed as an aspect of routing and *not* for child components rendered within a page. To authorize the display of specific parts within a page, use AuthorizeView instead.

The `[Authorize]` attribute also supports role-based or policy-based authorization. For role-based authorization, use the Roles parameter:

```
@page "/"
@attribute [Authorize(Roles = "admin, superuser")]

<p>You can only see this if you're in the 'admin' or 'superuser' role.</p>
```

For policy-based authorization, use the Policy parameter:

```
@page "/"
@attribute [Authorize(Policy = "content-editor")]

<p>You can only see this if you satisfy the 'content-editor' policy.</p>
```

If neither Roles nor Policy is specified, `[Authorize]` uses the default policy, which by default is to treat:

- Authenticated (signed-in) users as authorized.
- Unauthenticated (signed-out) users as unauthorized.

Customize unauthorized content with the Router component

The Router component, in conjunction with the AuthorizeRouteView component, allows the app to specify custom content if:

- Content isn't found.
- The user fails an `[Authorize]` condition applied to the component. The `[Authorize]` attribute is covered in the `[Authorize]` attribute section.
- Asynchronous authentication is in progress.

In the default Blazor Server project template, the `App` component (`App.razor`) demonstrates how to set custom content:

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    <h1>Sorry</h1>
                    <p>You're not authorized to reach this page.</p>
                    <p>You may need to log in as a different user.</p>
                </NotAuthorized>
                <Authorizing>
                    <h1>Authentication in progress</h1>
                    <p>Only visible while authentication is in progress.</p>
                </Authorizing>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <h1>Sorry</h1>
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

The content of `<NotFound>`, `<NotAuthorized>`, and `<Authorizing>` tags can include arbitrary items, such as other interactive components.

If the `<NotAuthorized>` tag isn't specified, the AuthorizeRouteView uses the following fallback message:

```
Not authorized.
```

Notification about authentication state changes

If the app determines that the underlying authentication state data has changed (for example, because the user signed out or another user has changed their roles), a custom `AuthenticationStateProvider` can optionally invoke the method NotifyAuthenticationStateChanged on the AuthenticationStateProvider base class. This notifies consumers of the authentication state data (for example, AuthorizeView) to rerender using the new data.

Procedural logic

If the app is required to check authorization rules as part of procedural logic, use a cascaded parameter of type `Task<` AuthenticationState `>` to obtain the user's ClaimsPrincipal. `Task<` AuthenticationState `>` can be combined with other services, such as `IAuthorizationService`, to evaluate policies.

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<button @onclick="@DoSomething">Do something important</button>

@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private async Task DoSomething()
    {
        var user = (await authenticationStateTask).User;

        if (user.Identity.IsAuthenticated)
        {
            // Perform an action only available to authenticated (signed-in) users.
        }

        if (user.IsInRole("admin"))
        {
            // Perform an action only available to users in the 'admin' role.
        }

        if ((await AuthorizationService.AuthorizeAsync(user, "content-editor"))
            .Succeeded)
        {
            // Perform an action only available to users satisfying the
            // 'content-editor' policy.
        }
    }
}
```

NOTE

In a Blazor WebAssembly app component, add the Microsoft.AspNetCore.Authorization and Microsoft.AspNetCore.Components.Authorization namespaces:

```
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
```

These namespaces can be provided globally by adding them to the app's `_Imports.razor` file.

Troubleshoot errors

Common errors:

- **Authorization requires a cascading parameter of type `Task\<AuthenticationState>`. Consider using `CascadingAuthenticationState` to supply this.**
- **`null` value is received for `authenticationStateTask`**

It's likely that the project wasn't created using a Blazor Server template with authentication enabled. Wrap a `<CascadingAuthenticationState>` around some part of the UI tree, for example in the `App` component (`App.razor`) as follows:

```
<CascadingAuthenticationState>
    <Router AppAssembly="typeof(Startup).Assembly">
        ...
    </Router>
</CascadingAuthenticationState>
```

The CascadingAuthenticationState supplies the `Task<` AuthenticationState `>` cascading parameter, which in turn it receives from the underlying AuthenticationStateProvider DI service.

Additional resources

- Overview of ASP.NET Core Security
- Configure Windows Authentication in ASP.NET Core
- Awesome Blazor: Authentication community sample links

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor file uploads

9/22/2020 • 2 minutes to read • Edit Online

By Daniel Roth

Use the `InputFile` component to read browser file data into .NET code, including for file uploads. The `InputFile` component renders as an HTML input of type `file`.

By default, the user selects single files. Add the `multiple` attribute to permit the user to upload multiple files at once. When one or more files is selected by the user, the `InputFile` component fires an `OnChange` event and passes in an `InputFileChangeEventArgs` that provides access to the selected file list and details about each file.

A component that receives an image file can call the `RequestImageFileAsync` convenience method on the file to resize the image data within the browser's JavaScript runtime before the image is streamed into the app.

The following example demonstrates multiple image file upload in a component:

```
<h3>Upload PNG images</h3>

<p>
    <InputFile OnChange="@OnInputFileChange" multiple />
</p>

@if (imageDataUrls.Count > 0)
{
    <h3>Images</h3>

    <div class="card" style="width:30rem;">
        <div class="card-body">
            @foreach (var imageDataUrl in imageDataUrls)
            {
                <img class="rounded m-1" src="@imageDataUrl" />
            }
        </div>
    </div>
}

@code {
    IList<string> imageDataUrls = new List<string>();

    private async Task OnInputFileChange(InputFileChangeEventArgs e)
    {
        var imageFiles = e.GetMultipleFiles();
        var format = "image/png";

        foreach (var imageFile in imageFiles)
        {
            var resizedImageFile = await imageFile.RequestImageFileAsync(format,
                100, 100);
            var buffer = new byte[resizedImageFile.Size];
            await resizedImageFile.OpenReadStream().ReadAsync(buffer);
            var imageDataUrl =
                $"data:{format};base64,{Convert.ToBase64String(buffer)}";
            imageDataUrls.Add(imageDataUrl);
        }
    }
}
```

To read data from a user-selected file, call `OpenReadStream` on the file and read from the returned stream. In a Blazor

WebAssembly app, the data is streamed directly into the .NET code within the browser. In a Blazor Server app, the file data is streamed into .NET code on the server as the file is read from the stream.

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

NOTE

If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor hosting model configuration

9/22/2020 • 7 minutes to read • Edit Online

By Daniel Roth, Mackinnon Buck, and Luke Latham

This article covers hosting model configuration.

SignalR cross-origin negotiation for authentication

This section applies to Blazor WebAssembly.

To configure SignalR's underlying client to send credentials, such as cookies or HTTP authentication headers:

- Use SetBrowserRequestCredentials to set Include on cross-origin `fetch` requests:

```
public class IncludeRequestCredentialsMessageHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        return base.SendAsync(request, cancellationToken);
    }
}
```

- Assign the HttpMessageHandler to the HttpMessageHandlerFactory option:

```
var connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://signalr.example.com"), options =>
    {
        options.HttpMessageHandlerFactory = innerHandler =>
            new IncludeRequestCredentialsMessageHandler { InnerHandler = innerHandler };
    }).Build();
```

For more information, see ASP.NET Core SignalR configuration.

Reflect the connection state in the UI

This section applies to Blazor Server.

When the client detects that the connection has been lost, a default UI is displayed to the user while the client attempts to reconnect. If reconnection fails, the user is provided the option to retry.

To customize the UI, define an element with an `id` of `components-reconnect-modal` in the `<body>` of the `_Host.cshtml` Razor page:

```
<div id="components-reconnect-modal">
    ...
</div>
```

Add the following to the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`):

```
#components-reconnect-modal {
    display: none;
}

#components-reconnect-modal.components-reconnect-show {
    display: block;
}
```

The following table describes the CSS classes applied to the `components-reconnect-modal` element.

CSS CLASS	INDICATES...
`components-reconnect-show`	A lost connection. The client is attempting to reconnect. Show the modal.
`components-reconnect-hide`	An active connection is re-established to the server. Hide the modal.
`components-reconnect-failed`	Reconnection failed, probably due to a network failure. To attempt reconnection, call `window.Blazor.reconnect()`.
`components-reconnect-rejected`	Reconnection rejected. The server was reached but refused the connection, and the user's state on the server is lost. To reload the app, call `location.reload()`. This connection state may result when: • A crash in the server-side circuit occurs. • The client is disconnected long enough for the server to drop the user's state. Instances of the components that the user is interacting with are disposed. • The server is restarted, or the app's worker process is recycled.

Render mode

This section applies to Blazor Server.

Blazor Server apps are set up by default to prerender the UI on the server before the client connection to the server is established. This is set up in the `_Host.cshtml` Razor page:

```
<body>
    <app>
      <component type="typeof(App)" render-mode="ServerPrerendered" />
    </app>

    <script src="_framework/blazor.server.js"></script>
</body>
```

RenderMode configures whether the component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

RENDER MODE	DESCRIPTION
ServerPrerendered	Renders the component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a Blazor Server app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the component into static HTML.

Rendering server components from a static HTML page isn't supported.

Initialize the Blazor circuit

This section applies to Blazor Server.

Configure the manual start of a Blazor Server app's SignalR circuit in the `Pages/_Host.cshtml` file:

- Add an `autostart="false"` attribute to the `<script>` tag for the `blazor.server.js` script.
- Place a script that calls `Blazor.start` after the `blazor.server.js` script's tag and inside the closing `</body>` tag.

When `autostart` is disabled, any aspect of the app that doesn't depend on the circuit works normally. For example, client-side routing is operational. However, any aspect that depends on the circuit isn't operational until `Blazor.start` is called. App behavior is unpredictable without an established circuit. For example, component methods fail to execute while the circuit is disconnected.

Initialize Blazor when the document is ready

To initialize the Blazor app when the document is ready:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      document.addEventListener("DOMContentLoaded", function() {
        Blazor.start();
      });
    </script>
</body>
```

Chain to the `Promise` that results from a manual start

To perform additional tasks, such as JS interop initialization, use `then` to chain to the `Promise` that results from a manual Blazor app start:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start().then(function () {
        ...
      });
    </script>
</body>
```

Configure the SignalR client

Logging

To configure SignalR client logging, pass in a configuration object (`configureSignalR`) that calls `configureLogging` with the log level on the client builder:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        configureSignalR: function (builder) {
          builder.configureLogging("information");
        }
      });
    </script>
</body>
```

In the preceding example, `information` is equivalent to a log level of LogLevel.Information.

Modify the reconnection handler

The reconnection handler's circuit connection events can be modified for custom behaviors, such as:

- To notify the user if the connection is dropped.
- To perform logging (from the client) when a circuit is connected.

To modify the connection events, register callbacks for the following connection changes:

- Dropped connections use `onConnectionDown` .
- Established/re-established connections use `onConnectionUp` .

Both `onConnectionDown` and `onConnectionUp` must be specified:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionHandler: {
          onConnectionDown: (options, error) => console.error(error);
          onConnectionUp: () => console.log("Up, up, and away!");
        }
      });
    </script>
</body>
```

Adjust the reconnection retry count and interval

To adjust the reconnection retry count and interval, set the number of retries (`maxRetries`) and period in milliseconds permitted for each retry attempt (`retryIntervalMilliseconds`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionOptions: {
          maxRetries: 3,
          retryIntervalMilliseconds: 2000
        }
      });
    </script>
</body>
```

Hide or replace the reconnection display

To hide the reconnection display, set the reconnection handler's `_reconnectionDisplay` to an empty object (`{}` or `new Object()`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      window.addEventListener('beforeunload', function () {
        Blazor.defaultReconnectionHandler._reconnectionDisplay = {};
      });

      Blazor.start();
    </script>
</body>
```

To replace the reconnection display, set `_reconnectionDisplay` in the preceding example to the element for display:

```
Blazor.defaultReconnectionHandler._reconnectionDisplay =
  document.getElementById("{ELEMENT ID}");
```

The placeholder `{ELEMENT ID}` is the ID of the HTML element to display.

Customize the delay before the reconnection display appears by setting the `transition-delay` property in the app's CSS (`wwwroot/css/site.css`) for the modal element. The following example sets the transition delay from 500 ms (default) to 1,000 ms (1 second):

```
#components-reconnect-modal {
    transition: visibility 0s linear 1000ms;
}
```

Influence HTML `<head>` tag elements

This section applies to the upcoming ASP.NET Core 5.0 release of Blazor WebAssembly and Blazor Server.

When rendered, the `Title`, `Link`, and `Meta` components add or update data in the HTML `<head>` tag elements:

```
@using Microsoft.AspNetCore.Components.Web.Extensions.Head

<Title Value="{TITLE}" />
<Link href="{URL}" rel="stylesheet" />
<Meta content="{DESCRIPTION}" name="description" />
```

In the preceding example, placeholders for `{TITLE}`, `{URL}`, and `{DESCRIPTION}` are string values, Razor variables, or Razor expressions.

The following characteristics apply:

- Server-side prerendering is supported.
- The `Value` parameter is the only valid parameter for the `Title` component.
- HTML attributes provided to the `Meta` and `Link` components are captured in additional attributes and passed through to the rendered HTML tag.
- For multiple `Title` components, the title of the page reflects the `Value` of the last `Title` component rendered.
- If multiple `Meta` or `Link` components are included with identical attributes, there's exactly one HTML tag rendered per `Meta` or `Link` component. Two `Meta` or `Link` components can't refer to the same rendered HTML tag.
- Changes to the parameters of existing `Meta` or `Link` components are reflected in their rendered HTML tags.
- When the `Link` or `Meta` components are no longer rendered and thus disposed by the framework, their rendered HTML tags are removed.

When one of the framework components is used in a child component, the rendered HTML tag influences any other child component of the parent component as long as the child component containing the framework component is rendered. The distinction between using the one of these framework components in a child component and placing a an HTML tag in `wwwroot/index.html` or `Pages/_Host.cshtml` is that a framework component's rendered HTML tag:

- Can be modified by application state. A hard-coded HTML tag can't be modified by application state.
- Is removed from the HTML `<head>` when the parent component is no longer rendered.

Static files

This section applies to Blazor Server.

To create additional file mappings with a FileExtensionContentTypeProvider or configure other StaticFileOptions, use **one** of the following approaches. In the following examples, the `{EXTENSION}` placeholder is the file extension, and the `{CONTENT TYPE}` placeholder is the content type.

- Configure options through dependency injection (DI) in `Startup.ConfigureServices` (`Startup.cs`) using StaticFileOptions:

```
using Microsoft.AspNetCore.StaticFiles;

...

var provider = new FileExtensionContentTypeProvider();
provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

services.Configure<StaticFileOptions>(options =>
{
    options.ContentTypeProvider = provider;
});
```

  Because this approach configures the same file provider used to serve `blazor.server.js`, make sure that your custom configuration doesn't interfere with serving `blazor.server.js`. For example, don't remove the mapping for JavaScript files by configuring the provider with `provider.Mappings.Remove(".js")`.

- Use two calls to UseStaticFiles in `Startup.Configure` (`Startup.cs`):
  - Configure the custom file provider in the first call with StaticFileOptions.
  - The second middleware serves `blazor.server.js`, which uses the default static files configuration provided by the Blazor framework.

```
using Microsoft.AspNetCore.StaticFiles;

...

var provider = new FileExtensionContentTypeProvider();
provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

app.UseStaticFiles(new StaticFileOptions { ContentTypeProvider = provider });
app.UseStaticFiles();
```

Additional resources

- Logging in .NET Core and ASP.NET Core

ASP.NET Core Blazor layouts

9/22/2020 • 4 minutes to read • Edit Online

By Rainer Stropek and Luke Latham

Some app elements, such as menus, copyright messages, and company logos, are usually part of app's overall layout and used by every component in the app. Copying the code of these elements into all of the components of an app isn't an efficient approach. Every time one of the elements requires an update, every component must be updated. Such duplication is difficult to maintain and can lead to inconsistent content over time. *Layouts* solve this problem.

Technically, a layout is just another component. A layout is defined in a Razor template or in C# code and can use data binding, dependency injection, and other component scenarios.

To turn a *component* into a *layout*, the component:

- Inherits from LayoutComponentBase, which defines a Body property for the rendered content inside the layout.
- Uses the Razor syntax `@Body` to specify the location in the layout markup where the content is rendered.

The following code sample shows the Razor template of a layout component, `MainLayout.razor`. The layout inherits LayoutComponentBase and sets the `@Body` between the navigation bar and the footer:

```
@inherits LayoutComponentBase

<header>
    <h1>Doctor Who&trade; Episode Database</h1>
</header>

<nav>
    <a href="masterlist">Master Episode List</a>
    <a href="search">Search</a>
    <a href="new">Add Episode</a>
</nav>

@Body

<footer>
    @TrademarkMessage
</footer>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/";
}
```

`MainLayout` component

In an app based on one of the Blazor project templates, the `MainLayout` component (`MainLayout.razor`) is in the app's `Shared` folder:

```
@inherits LayoutComponentBase

<div class="sidebar">
    <NavMenu />
</div>

<div class="main">
    <div class="content px-4">
        @Body
    </div>
</div>
```

Default layout

Specify the default app layout in the Router component in the app's `App.razor` file. The following Router component, which is provided by the default Blazor templates, sets the default layout to the `MainLayout` component:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

To supply a default layout for NotFound content, specify a LayoutView for NotFound content:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <h1>Page not found</h1>
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

For more information on the Router component, see ASP.NET Core Blazor routing.

Specifying the layout as a default layout in the router is a useful practice because it can be overridden on a per-component or per-folder basis. Prefer using the router to set the app's default layout because it's the most general technique.

Specify a layout in a component

Use the Razor directive `@layout` to apply a layout to a component. The compiler converts `@layout` into a LayoutAttribute, which is applied to the component class.

The content of the following `MasterList` component is inserted into the `MasterLayout` at the position of `@Body`:

```
@layout MasterLayout
@page "/masterlist"

<h1>Master Episode List</h1>
```

Specifying the layout directly in a component overrides a *default layout* set in the router or an `@layout` directive imported from `_Imports.razor`.

Centralized layout selection

Every folder of an app can optionally contain a template file named `_Imports.razor`. The compiler includes the directives specified in the imports file in all of the Razor templates in the same folder and recursively in all of its subfolders. Therefore, an `_Imports.razor` file containing `@layout MyCoolLayout` ensures that all of the components in a folder use `MyCoolLayout`. There's no need to repeatedly add `@layout MyCoolLayout` to all of the `.razor` files within the folder and subfolders. `@using` directives are also applied to components in the same way.

The following `_Imports.razor` file imports:

- `MyCoolLayout`.
- All Razor components in the same folder and any subfolders.
- The `BlazorApp1.Data` namespace.

```
@layout MyCoolLayout
@using Microsoft.AspNetCore.Components
@using BlazorApp1.Data
```

The `_Imports.razor` file is similar to the _ViewImports.cshtml file for Razor views and pages but applied specifically to Razor component files.

Specifying a layout in `_Imports.razor` overrides a layout specified as the router's *default layout*.

WARNING

Do **not** add a Razor `@layout` directive to the root `_Imports.razor` file, which results in an infinite loop of layouts in the app. To control the default app layout, specify the layout in the `Router` component. For more information, see the Default layout section.

Nested layouts

Apps can consist of nested layouts. A component can reference a layout which in turn references another layout. For example, nesting layouts are used to create a multi-level menu structure.

The following example shows how to use nested layouts. The `EpisodesComponent.razor` file is the component to display. The component references the `MasterListLayout`:

```
@layout MasterListLayout
@page "/masterlist/episodes"

<h1>Episodes</h1>
```

The `MasterListLayout.razor` file provides the `MasterListLayout`. The layout references another layout, `MasterLayout`, where it's rendered. `EpisodesComponent` is rendered where `@Body` appears:

```
@layout MasterLayout
@inherits LayoutComponentBase

<nav>
    <!-- Menu structure of master list -->
    ...
</nav>

@Body
```

Finally, `MasterLayout` in `MasterLayout.razor` contains the top-level layout elements, such as the header, main menu, and footer. `MasterListLayout` with the `EpisodesComponent` is rendered where `@Body` appears:

```
@inherits LayoutComponentBase

<header>...</header>
<nav>...</nav>

@Body

<footer>
    @TrademarkMessage
</footer>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/";
}
```

Share a Razor Pages layout with integrated components

When routable components are integrated into a Razor Pages app, the app's shared layout can be used with the components. For more information, see Integrate ASP.NET Core Razor components into Razor Pages and MVC apps.

Additional resources

- Layout in ASP.NET Core

ASP.NET Core Blazor hosting model configuration

9/22/2020 • 7 minutes to read • Edit Online

By Daniel Roth, Mackinnon Buck, and Luke Latham

This article covers hosting model configuration.

SignalR cross-origin negotiation for authentication

This section applies to Blazor WebAssembly.

To configure SignalR's underlying client to send credentials, such as cookies or HTTP authentication headers:

- Use SetBrowserRequestCredentials to set Include on cross-origin `fetch` requests:

```
public class IncludeRequestCredentialsMessageHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        return base.SendAsync(request, cancellationToken);
    }
}
```

- Assign the HttpMessageHandler to the HttpMessageHandlerFactory option:

```
var connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://signalr.example.com"), options =>
    {
        options.HttpMessageHandlerFactory = innerHandler =>
            new IncludeRequestCredentialsMessageHandler { InnerHandler = innerHandler };
    }).Build();
```

For more information, see ASP.NET Core SignalR configuration.

Reflect the connection state in the UI

This section applies to Blazor Server.

When the client detects that the connection has been lost, a default UI is displayed to the user while the client attempts to reconnect. If reconnection fails, the user is provided the option to retry.

To customize the UI, define an element with an `id` of `components-reconnect-modal` in the `<body>` of the `_Host.cshtml` Razor page:

```
<div id="components-reconnect-modal">
    ...
</div>
```

Add the following to the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`):

```
#components-reconnect-modal {
    display: none;
}

#components-reconnect-modal.components-reconnect-show {
    display: block;
}
```

The following table describes the CSS classes applied to the `components-reconnect-modal` element.

CSS CLASS	INDICATES...
`components-reconnect-show`	A lost connection. The client is attempting to reconnect. Show the modal.
`components-reconnect-hide`	An active connection is re-established to the server. Hide the modal.
`components-reconnect-failed`	Reconnection failed, probably due to a network failure. To attempt reconnection, call `window.Blazor.reconnect()`.
`components-reconnect-rejected`	Reconnection rejected. The server was reached but refused the connection, and the user's state on the server is lost. To reload the app, call `location.reload()`. This connection state may result when: • A crash in the server-side circuit occurs. • The client is disconnected long enough for the server to drop the user's state. Instances of the components that the user is interacting with are disposed. • The server is restarted, or the app's worker process is recycled.

Render mode

This section applies to Blazor Server.

Blazor Server apps are set up by default to prerender the UI on the server before the client connection to the server is established. This is set up in the `_Host.cshtml` Razor page:

```
<body>
    <app>
      <component type="typeof(App)" render-mode="ServerPrerendered" />
    </app>

    <script src="_framework/blazor.server.js"></script>
</body>
```

RenderMode configures whether the component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

RENDER MODE	DESCRIPTION
ServerPrerendered	Renders the component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a Blazor Server app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the component into static HTML.

Rendering server components from a static HTML page isn't supported.

Initialize the Blazor circuit

This section applies to Blazor Server.

Configure the manual start of a Blazor Server app's SignalR circuit in the `Pages/_Host.cshtml` file:

- Add an `autostart="false"` attribute to the `<script>` tag for the `blazor.server.js` script.
- Place a script that calls `Blazor.start` after the `blazor.server.js` script's tag and inside the closing `</body>` tag.

When `autostart` is disabled, any aspect of the app that doesn't depend on the circuit works normally. For example, client-side routing is operational. However, any aspect that depends on the circuit isn't operational until `Blazor.start` is called. App behavior is unpredictable without an established circuit. For example, component methods fail to execute while the circuit is disconnected.

Initialize Blazor when the document is ready

To initialize the Blazor app when the document is ready:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      document.addEventListener("DOMContentLoaded", function() {
        Blazor.start();
      });
    </script>
</body>
```

Chain to the `Promise` that results from a manual start

To perform additional tasks, such as JS interop initialization, use `then` to chain to the `Promise` that results from a manual Blazor app start:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start().then(function () {
        ...
      });
    </script>
</body>
```

Configure the SignalR client

Logging

To configure SignalR client logging, pass in a configuration object (`configureSignalR`) that calls `configureLogging` with the log level on the client builder:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        configureSignalR: function (builder) {
          builder.configureLogging("information");
        }
      });
    </script>
</body>
```

In the preceding example, `information` is equivalent to a log level of LogLevel.Information.

Modify the reconnection handler

The reconnection handler's circuit connection events can be modified for custom behaviors, such as:

- To notify the user if the connection is dropped.
- To perform logging (from the client) when a circuit is connected.

To modify the connection events, register callbacks for the following connection changes:

- Dropped connections use `onConnectionDown` .
- Established/re-established connections use `onConnectionUp` .

Both `onConnectionDown` and `onConnectionUp` must be specified:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionHandler: {
          onConnectionDown: (options, error) => console.error(error);
          onConnectionUp: () => console.log("Up, up, and away!");
        }
      });
    </script>
</body>
```

Adjust the reconnection retry count and interval

To adjust the reconnection retry count and interval, set the number of retries (`maxRetries`) and period in milliseconds permitted for each retry attempt (`retryIntervalMilliseconds`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionOptions: {
          maxRetries: 3,
          retryIntervalMilliseconds: 2000
        }
      });
    </script>
</body>
```

Hide or replace the reconnection display

To hide the reconnection display, set the reconnection handler's `_reconnectionDisplay` to an empty object (`{}` or `new Object()`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      window.addEventListener('beforeunload', function () {
        Blazor.defaultReconnectionHandler._reconnectionDisplay = {};
      });

      Blazor.start();
    </script>
</body>
```

To replace the reconnection display, set `_reconnectionDisplay` in the preceding example to the element for display:

```
Blazor.defaultReconnectionHandler._reconnectionDisplay =
  document.getElementById("{ELEMENT ID}");
```

The placeholder `{ELEMENT ID}` is the ID of the HTML element to display.

Customize the delay before the reconnection display appears by setting the `transition-delay` property in the app's CSS (`wwwroot/css/site.css`) for the modal element. The following example sets the transition delay from 500 ms (default) to 1,000 ms (1 second):

```
#components-reconnect-modal {
    transition: visibility 0s linear 1000ms;
}
```

Influence HTML `<head>` tag elements

This section applies to the upcoming ASP.NET Core 5.0 release of Blazor WebAssembly and Blazor Server.

When rendered, the `Title`, `Link`, and `Meta` components add or update data in the HTML `<head>` tag elements:

```
@using Microsoft.AspNetCore.Components.Web.Extensions.Head

<Title Value="{TITLE}" />
<Link href="{URL}" rel="stylesheet" />
<Meta content="{DESCRIPTION}" name="description" />
```

In the preceding example, placeholders for `{TITLE}`, `{URL}`, and `{DESCRIPTION}` are string values, Razor variables, or Razor expressions.

The following characteristics apply:

- Server-side prerendering is supported.
- The `Value` parameter is the only valid parameter for the `Title` component.
- HTML attributes provided to the `Meta` and `Link` components are captured in additional attributes and passed through to the rendered HTML tag.
- For multiple `Title` components, the title of the page reflects the `Value` of the last `Title` component rendered.
- If multiple `Meta` or `Link` components are included with identical attributes, there's exactly one HTML tag rendered per `Meta` or `Link` component. Two `Meta` or `Link` components can't refer to the same rendered HTML tag.
- Changes to the parameters of existing `Meta` or `Link` components are reflected in their rendered HTML tags.
- When the `Link` or `Meta` components are no longer rendered and thus disposed by the framework, their rendered HTML tags are removed.

When one of the framework components is used in a child component, the rendered HTML tag influences any other child component of the parent component as long as the child component containing the framework component is rendered. The distinction between using the one of these framework components in a child component and placing a an HTML tag in `wwwroot/index.html` or `Pages/_Host.cshtml` is that a framework component's rendered HTML tag:

- Can be modified by application state. A hard-coded HTML tag can't be modified by application state.
- Is removed from the HTML `<head>` when the parent component is no longer rendered.

Static files

This section applies to Blazor Server.

To create additional file mappings with a FileExtensionContentTypeProvider or configure other StaticFileOptions, use **one** of the following approaches. In the following examples, the `{EXTENSION}` placeholder is the file extension, and the `{CONTENT TYPE}` placeholder is the content type.

- Configure options through dependency injection (DI) in `Startup.ConfigureServices` (`Startup.cs`) using StaticFileOptions:

```
using Microsoft.AspNetCore.StaticFiles;

...

var provider = new FileExtensionContentTypeProvider();
provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

services.Configure<StaticFileOptions>(options =>
{
    options.ContentTypeProvider = provider;
});
```

  Because this approach configures the same file provider used to serve `blazor.server.js`, make sure that your custom configuration doesn't interfere with serving `blazor.server.js`. For example, don't remove the mapping for JavaScript files by configuring the provider with `provider.Mappings.Remove(".js")`.

- Use two calls to UseStaticFiles in `Startup.Configure` (`Startup.cs`):
  - Configure the custom file provider in the first call with StaticFileOptions.
  - The second middleware serves `blazor.server.js`, which uses the default static files configuration provided by the Blazor framework.

```
using Microsoft.AspNetCore.StaticFiles;

...

var provider = new FileExtensionContentTypeProvider();
provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

app.UseStaticFiles(new StaticFileOptions { ContentTypeProvider = provider });
app.UseStaticFiles();
```

Additional resources

- Logging in .NET Core and ASP.NET Core

ASP.NET Core Blazor routing

9/22/2020 • 9 minutes to read • Edit Online

By Luke Latham

Learn how to route requests and how to use the NavLink component to create navigation links in Blazor apps.

ASP.NET Core endpoint routing integration

Blazor Server is integrated into ASP.NET Core Endpoint Routing. An ASP.NET Core app is configured to accept incoming connections for interactive components with MapBlazorHub in `Startup.Configure`:

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

The most typical configuration is to route all requests to a Razor page, which acts as the host for the server-side part of the Blazor Server app. By convention, the *host* page is usually named `_Host.cshtml`. The route specified in the host file is called a *fallback route* because it operates with a low priority in route matching. The fallback route is considered when other routes don't match. This allows the app to use others controllers and pages without interfering with the Blazor Server app.

For information on configuring MapFallbackToPage for non-root URL server hosting, see Host and deploy ASP.NET Core Blazor.

Route templates

The Router component enables routing to each component with a specified route. The Router component appears in the `App.razor` file:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

When a `.razor` file with an `@page` directive is compiled, the generated class is provided a RouteAttribute specifying the route template.

At runtime, the RouteView component:

- Receives the RouteData from the Router along with any desired parameters.
- Renders the specified component with its layout (or an optional default layout) using the specified parameters.

You can optionally specify a DefaultLayout parameter with a layout class to use for components that don't specify a layout. The default Blazor templates specify the `MainLayout` component. `MainLayout.razor` is in the template

project's `Shared` folder. For more information on layouts, see ASP.NET Core Blazor layouts.

Multiple route templates can be applied to a component. The following component responds to requests for `/BlazorRoute` and `/DifferentBlazorRoute`:

```
@page "/BlazorRoute"
@page "/DifferentBlazorRoute"

<h1>Blazor routing</h1>
```

IMPORTANT

For URLs to resolve correctly, the app must include a `<base>` tag in its `wwwroot/index.html` file (Blazor WebAssembly) or `Pages/_Host.cshtml` file (Blazor Server) with the app base path specified in the `href` attribute (`<base href="/">`). For more information, see Host and deploy ASP.NET Core Blazor.

Provide custom content when content isn't found

The Router component allows the app to specify custom content if content isn't found for the requested route.

In the `App.razor` file, set custom content in the NotFound template parameter of the Router component:

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <h1>Sorry</h1>
        <p>Sorry, there's nothing at this address.</p> b
    </NotFound>
</Router>
```

The content of `<NotFound>` tags can include arbitrary items, such as other interactive components. To apply a default layout to NotFound content, see ASP.NET Core Blazor layouts.

Route to components from multiple assemblies

Use the AdditionalAssemblies parameter to specify additional assemblies for the Router component to consider when searching for routable components. Specified assemblies are considered in addition to the `AppAssembly`-specified assembly. In the following example, `Component1` is a routable component defined in a referenced class library. The following AdditionalAssemblies example results in routing support for `Component1`:

```
<Router
    AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="new[] { typeof(Component1).Assembly }">
    ...
</Router>
```

Route parameters

The router uses route parameters to populate the corresponding component parameters with the same name (case insensitive):

```
@page "/RouteParameter"
@page "/RouteParameter/{text}"

<h1>Blazor is @Text!</h1>

@code {
    [Parameter]
    public string Text { get; set; }

    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

Optional parameters aren't supported. Two `@page` directives are applied in the previous example. The first permits navigation to the component without a parameter. The second `@page` directive takes the `{text}` route parameter and assigns the value to the `Text` property.

Route constraints

A route constraint enforces type matching on a route segment to a component.

In the following example, the route to the `Users` component only matches if:

- An `Id` route segment is present on the request URL.
- The `Id` segment is an integer (`int`).

```
@page "/Users/{Id:int}"

<h1>The user Id is @Id!</h1>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

The route constraints shown in the following table are available. For the route constraints that match with the invariant culture, see the warning below the table for more information.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`bool`	`{active:bool}`	`true`, `FALSE`	No
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Yes
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Yes
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Yes
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Yes

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	No
`int`	`{id:int}`	`123456789`, `-123456789`	Yes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Yes

WARNING

Route constraints that verify the URL and are converted to a CLR type (such as `int` or DateTime) always use the invariant culture. These constraints assume that the URL is non-localizable.

Routing with URLs that contain dots

For hosted Blazor WebAssembly and Blazor Server apps, the server-side default route template assumes that if the last segment of a request URL contains a dot (`.`) that a file is requested (for example, `https://localhost.com:5001/example/some.thing`). Without additional configuration, an app returns a *404 - Not Found* response if this was meant to route to a component. To use a route with one or more parameters that contains a dot, the app must configure the route with a custom template.

Consider the following `Example` component that can receive a route parameter from the last segment of the URL:

```
@page "/example"
@page "/example/{param}"

<p>
    Param: @Param
</p>

@code {
    [Parameter]
    public string Param { get; set; }
}
```

To permit the *Server* app of a hosted Blazor WebAssembly solution to route the request with a dot in the `param` parameter, add a fallback file route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToFile("/example/{param?}", "index.html");
```

To configure a Blazor Server app to route the request with a dot in the `param` parameter, add a fallback page route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToPage("/example/{param?}", "/_Host");
```

For more information, see Routing in ASP.NET Core.

Catch-all route parameters

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components. The catch-all route parameter must be:

- Named to match the route segment name. Naming isn't case sensitive.
- A `string` type. The framework doesn't provide automatic casting.
- At the end of the URL.

```
@page "/page/{*pageRoute}"

@code {
    [Parameter]
    public string PageRoute { get; set; }
}
```

For the URL `/page/this/is/a/test` with a route template of `/page/{*pageRoute}`, the value of `PageRoute` is set to `this/is/a/test`.

Slashes and segments of the captured path are decoded. For a route template of `/page/{*pageRoute}`, the URL `/page/this/is/a%2Ftest%2A` yields `this/is/a/test*`.

Catch-all route parameters are supported in ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.*

NavLink component

Use a NavLink component in place of HTML hyperlink elements (`<a>`) when creating navigation links. A NavLink component behaves like an `<a>` element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed. Optionally, assign a CSS class name to NavLink.ActiveClass to apply a custom CSS class to the rendered link when the current route matches the `href`.

The following `NavMenu` component creates a `Bootstrap` navigation bar that demonstrates how to use NavLink components:

```
<div class="@NavMenuCssClass" @onclick="@ToggleNavMenu">
    <ul class="nav flex-column">
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="MyComponent" Match="NavLinkMatch.Prefix">
                <span class="oi oi-plus" aria-hidden="true"></span> My Component
            </NavLink>
        </li>
    </ul>
</div>
```

There are two NavLinkMatch options that you can assign to the `Match` attribute of the `<NavLink>` element:

- NavLinkMatch.All: The NavLink is active when it matches the entire current URL.
- NavLinkMatch.Prefix (*default*): The NavLink is active when it matches any prefix of the current URL.

In the preceding example, the Home NavLink `href=""` matches the home URL and only receives the `active` CSS class at the app's default base path URL (for example, `https://localhost:5001/`). The second NavLink receives the `active` class when the user visits any URL with a `MyComponent` prefix (for example,

`https://localhost:5001/MyComponent` and `https://localhost:5001/MyComponent/AnotherSegment`).

Additional NavLink component attributes are passed through to the rendered anchor tag. In the following example, the NavLink component includes the `target` attribute:

```
<NavLink href="my-page" target="_blank">My page</NavLink>
```

The following HTML markup is rendered:

```
<a href="my-page" target="_blank">My page</a>
```

WARNING

Due to the way that Blazor renders child content, rendering `NavLink` components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the `NavLink` (child) component's content:

```
@for (int c = 0; c < 10; c++)
{
    var current = c;
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @current
        </NavLink>
    </li>
}
```

Using an index variable in this scenario is a requirement for **any** child component that uses a loop variable in its child content, not just the `NavLink` component.

Alternatively, use a `foreach` loop with Enumerable.Range:

```
@foreach(var c in Enumerable.Range(0,10))
{
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @c
        </NavLink>
    </li>
}
```

URI and navigation state helpers

Use NavigationManager to work with URIs and navigation in C# code. NavigationManager provides the event and methods shown in the following table.

MEMBER	DESCRIPTION
Uri	Gets the current absolute URI.
BaseUri	Gets the base URI (with a trailing slash) that can be prepended to relative URI paths to produce an absolute URI. Typically, BaseUri corresponds to the `href` attribute on the document's `<base>` element in `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server).

MEMBER	DESCRIPTION
NavigateTo	Navigates to the specified URI. If `forceLoad` is `true`: • Client-side routing is bypassed. • The browser is forced to load the new page from the server, whether or not the URI is normally handled by the client-side router.
LocationChanged	An event that fires when the navigation location has changed.
ToAbsoluteUri	Converts a relative URI into an absolute URI.
ToBaseRelativePath	Given a base URI (for example, a URI previously returned by BaseUri), converts an absolute URI into a URI relative to the base URI prefix.

The following component navigates to the app's `Counter` component when the button is selected:

```
@page "/navigate"
@inject NavigationManager NavigationManager

<h1>Navigate in Code Example</h1>

<button class="btn btn-primary" @onclick="NavigateToCounterComponent">
    Navigate to the Counter component
</button>

@code {
    private void NavigateToCounterComponent()
    {
        NavigationManager.NavigateTo("counter");
    }
}
```

The following component handles a location changed event by subscribing to NavigationManager.LocationChanged. The `HandleLocationChanged` method is unhooked when `Dispose` is called by the framework. Unhooking the method permits garbage collection of the component.

```
@implements IDisposable
@inject NavigationManager NavigationManager

...

protected override void OnInitialized()
{
    NavigationManager.LocationChanged += HandleLocationChanged;
}

private void HandleLocationChanged(object sender, LocationChangedEventArgs e)
{
    ...
}

public void Dispose()
{
    NavigationManager.LocationChanged -= HandleLocationChanged;
}
```

LocationChangedEventArgs provides the following information about the event:

- Location: The URL of the new location.
- IsNavigationIntercepted: If `true`, Blazor intercepted the navigation from the browser. If `false`, NavigationManager.NavigateTo caused the navigation to occur.

For more information on component disposal, see ASP.NET Core Blazor lifecycle.

Query string and parse parameters

The query string of a request can be obtained from the NavigationManager's Uri property:

```
@inject NavigationManager Navigation

...

var query = new Uri(Navigation.Uri).Query;
```

To parse a query string's parameters:

- Add a package reference for Microsoft.AspNetCore.WebUtilities.
- Obtain the value after parsing the query string with QueryHelpers.ParseQuery.

```
@page "/"
@using Microsoft.AspNetCore.WebUtilities
@inject NavigationManager NavigationManager

<h1>Query string parse example</h1>

<p>Value: @queryValue</p>

@code {
    private string queryValue = "Not set";

    protected override void OnInitialized()
    {
        var query = new Uri(NavigationManager.Uri).Query;

        if (QueryHelpers.ParseQuery(query).TryGetValue("{KEY}", out var value))
        {
            queryValue = value;
        }
    }
}
```

The placeholder `{KEY}` in the preceding example is the query string parameter key. For example, the URL key-value pair `?ship=Tardis` uses a key of `ship`.

ASP.NET Core Blazor routing

9/22/2020 • 9 minutes to read • Edit Online

By Luke Latham

Learn how to route requests and how to use the NavLink component to create navigation links in Blazor apps.

ASP.NET Core endpoint routing integration

Blazor Server is integrated into ASP.NET Core Endpoint Routing. An ASP.NET Core app is configured to accept incoming connections for interactive components with MapBlazorHub in `Startup.Configure`:

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

The most typical configuration is to route all requests to a Razor page, which acts as the host for the server-side part of the Blazor Server app. By convention, the *host* page is usually named `_Host.cshtml`. The route specified in the host file is called a *fallback route* because it operates with a low priority in route matching. The fallback route is considered when other routes don't match. This allows the app to use others controllers and pages without interfering with the Blazor Server app.

For information on configuring MapFallbackToPage for non-root URL server hosting, see Host and deploy ASP.NET Core Blazor.

Route templates

The Router component enables routing to each component with a specified route. The Router component appears in the `App.razor` file:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

When a `.razor` file with an `@page` directive is compiled, the generated class is provided a RouteAttribute specifying the route template.

At runtime, the RouteView component:

- Receives the RouteData from the Router along with any desired parameters.
- Renders the specified component with its layout (or an optional default layout) using the specified parameters.

You can optionally specify a DefaultLayout parameter with a layout class to use for components that don't specify a layout. The default Blazor templates specify the `MainLayout` component. `MainLayout.razor` is in the template

project's `Shared` folder. For more information on layouts, see ASP.NET Core Blazor layouts.

Multiple route templates can be applied to a component. The following component responds to requests for `/BlazorRoute` and `/DifferentBlazorRoute`:

```
@page "/BlazorRoute"
@page "/DifferentBlazorRoute"

<h1>Blazor routing</h1>
```

IMPORTANT

For URLs to resolve correctly, the app must include a `<base>` tag in its `wwwroot/index.html` file (Blazor WebAssembly) or `Pages/_Host.cshtml` file (Blazor Server) with the app base path specified in the `href` attribute (`<base href="/">`). For more information, see Host and deploy ASP.NET Core Blazor.

Provide custom content when content isn't found

The Router component allows the app to specify custom content if content isn't found for the requested route.

In the `App.razor` file, set custom content in the NotFound template parameter of the Router component:

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <h1>Sorry</h1>
        <p>Sorry, there's nothing at this address.</p> b
    </NotFound>
</Router>
```

The content of `<NotFound>` tags can include arbitrary items, such as other interactive components. To apply a default layout to NotFound content, see ASP.NET Core Blazor layouts.

Route to components from multiple assemblies

Use the AdditionalAssemblies parameter to specify additional assemblies for the Router component to consider when searching for routable components. Specified assemblies are considered in addition to the `AppAssembly`-specified assembly. In the following example, `Component1` is a routable component defined in a referenced class library. The following AdditionalAssemblies example results in routing support for `Component1`:

```
<Router
    AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="new[] { typeof(Component1).Assembly }">
    ...
</Router>
```

Route parameters

The router uses route parameters to populate the corresponding component parameters with the same name (case insensitive):

```
@page "/RouteParameter"
@page "/RouteParameter/{text}"

<h1>Blazor is @Text!</h1>

@code {
    [Parameter]
    public string Text { get; set; }

    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

Optional parameters aren't supported. Two `@page` directives are applied in the previous example. The first permits navigation to the component without a parameter. The second `@page` directive takes the `{text}` route parameter and assigns the value to the `Text` property.

Route constraints

A route constraint enforces type matching on a route segment to a component.

In the following example, the route to the `Users` component only matches if:

- An `Id` route segment is present on the request URL.
- The `Id` segment is an integer (`int`).

```
@page "/Users/{Id:int}"

<h1>The user Id is @Id!</h1>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

The route constraints shown in the following table are available. For the route constraints that match with the invariant culture, see the warning below the table for more information.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`bool`	`{active:bool}`	`true`, `FALSE`	No
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Yes
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Yes
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Yes
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Yes

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	No
`int`	`{id:int}`	`123456789`, `-123456789`	Yes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Yes

WARNING

Route constraints that verify the URL and are converted to a CLR type (such as `int` or DateTime) always use the invariant culture. These constraints assume that the URL is non-localizable.

Routing with URLs that contain dots

For hosted Blazor WebAssembly and Blazor Server apps, the server-side default route template assumes that if the last segment of a request URL contains a dot (`.`) that a file is requested (for example, `https://localhost.com:5001/example/some.thing`). Without additional configuration, an app returns a *404 - Not Found* response if this was meant to route to a component. To use a route with one or more parameters that contains a dot, the app must configure the route with a custom template.

Consider the following `Example` component that can receive a route parameter from the last segment of the URL:

```
@page "/example"
@page "/example/{param}"

<p>
    Param: @Param
</p>

@code {
    [Parameter]
    public string Param { get; set; }
}
```

To permit the *Server* app of a hosted Blazor WebAssembly solution to route the request with a dot in the `param` parameter, add a fallback file route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToFile("/example/{param?}", "index.html");
```

To configure a Blazor Server app to route the request with a dot in the `param` parameter, add a fallback page route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToPage("/example/{param?}", "/_Host");
```

For more information, see Routing in ASP.NET Core.

Catch-all route parameters

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components. The catch-all route parameter must be:

- Named to match the route segment name. Naming isn't case sensitive.
- A `string` type. The framework doesn't provide automatic casting.
- At the end of the URL.

```
@page "/page/{*pageRoute}"

@code {
    [Parameter]
    public string PageRoute { get; set; }
}
```

For the URL `/page/this/is/a/test` with a route template of `/page/{*pageRoute}`, the value of `PageRoute` is set to `this/is/a/test`.

Slashes and segments of the captured path are decoded. For a route template of `/page/{*pageRoute}`, the URL `/page/this/is/a%2Ftest%2A` yields `this/is/a/test*`.

Catch-all route parameters are supported in ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.*

NavLink component

Use a NavLink component in place of HTML hyperlink elements (`<a>`) when creating navigation links. A NavLink component behaves like an `<a>` element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed. Optionally, assign a CSS class name to NavLink.ActiveClass to apply a custom CSS class to the rendered link when the current route matches the `href`.

The following `NavMenu` component creates a `Bootstrap` navigation bar that demonstrates how to use NavLink components:

```
<div class="@NavMenuCssClass" @onclick="@ToggleNavMenu">
    <ul class="nav flex-column">
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="MyComponent" Match="NavLinkMatch.Prefix">
                <span class="oi oi-plus" aria-hidden="true"></span> My Component
            </NavLink>
        </li>
    </ul>
</div>
```

There are two NavLinkMatch options that you can assign to the `Match` attribute of the `<NavLink>` element:

- NavLinkMatch.All: The NavLink is active when it matches the entire current URL.
- NavLinkMatch.Prefix (*default*): The NavLink is active when it matches any prefix of the current URL.

In the preceding example, the Home NavLink `href=""` matches the home URL and only receives the `active` CSS class at the app's default base path URL (for example, `https://localhost:5001/`). The second NavLink receives the `active` class when the user visits any URL with a `MyComponent` prefix (for example,

`https://localhost:5001/MyComponent` and `https://localhost:5001/MyComponent/AnotherSegment`).

Additional NavLink component attributes are passed through to the rendered anchor tag. In the following example, the NavLink component includes the `target` attribute:

```
<NavLink href="my-page" target="_blank">My page</NavLink>
```

The following HTML markup is rendered:

```
<a href="my-page" target="_blank">My page</a>
```

WARNING

Due to the way that Blazor renders child content, rendering `NavLink` components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the `NavLink` (child) component's content:

```
@for (int c = 0; c < 10; c++)
{
    var current = c;
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @current
        </NavLink>
    </li>
}
```

Using an index variable in this scenario is a requirement for **any** child component that uses a loop variable in its child content, not just the `NavLink` component.

Alternatively, use a `foreach` loop with Enumerable.Range:

```
@foreach(var c in Enumerable.Range(0,10))
{
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @c
        </NavLink>
    </li>
}
```

URI and navigation state helpers

Use NavigationManager to work with URIs and navigation in C# code. NavigationManager provides the event and methods shown in the following table.

MEMBER	DESCRIPTION
Uri	Gets the current absolute URI.
BaseUri	Gets the base URI (with a trailing slash) that can be prepended to relative URI paths to produce an absolute URI. Typically, BaseUri corresponds to the `href` attribute on the document's `<base>` element in `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server).

MEMBER	DESCRIPTION
NavigateTo	Navigates to the specified URI. If `forceLoad` is `true`: • Client-side routing is bypassed. • The browser is forced to load the new page from the server, whether or not the URI is normally handled by the client-side router.
LocationChanged	An event that fires when the navigation location has changed.
ToAbsoluteUri	Converts a relative URI into an absolute URI.
ToBaseRelativePath	Given a base URI (for example, a URI previously returned by BaseUri), converts an absolute URI into a URI relative to the base URI prefix.

The following component navigates to the app's `Counter` component when the button is selected:

```
@page "/navigate"
@inject NavigationManager NavigationManager

<h1>Navigate in Code Example</h1>

<button class="btn btn-primary" @onclick="NavigateToCounterComponent">
    Navigate to the Counter component
</button>

@code {
    private void NavigateToCounterComponent()
    {
        NavigationManager.NavigateTo("counter");
    }
}
```

The following component handles a location changed event by subscribing to NavigationManager.LocationChanged. The `HandleLocationChanged` method is unhooked when `Dispose` is called by the framework. Unhooking the method permits garbage collection of the component.

```
@implements IDisposable
@inject NavigationManager NavigationManager

...

protected override void OnInitialized()
{
    NavigationManager.LocationChanged += HandleLocationChanged;
}

private void HandleLocationChanged(object sender, LocationChangedEventArgs e)
{
    ...
}

public void Dispose()
{
    NavigationManager.LocationChanged -= HandleLocationChanged;
}
```

LocationChangedEventArgs provides the following information about the event:

- Location: The URL of the new location.
- IsNavigationIntercepted: If `true`, Blazor intercepted the navigation from the browser. If `false`, NavigationManager.NavigateTo caused the navigation to occur.

For more information on component disposal, see ASP.NET Core Blazor lifecycle.

Query string and parse parameters

The query string of a request can be obtained from the NavigationManager's Uri property:

```
@inject NavigationManager Navigation

...

var query = new Uri(Navigation.Uri).Query;
```

To parse a query string's parameters:

- Add a package reference for Microsoft.AspNetCore.WebUtilities.
- Obtain the value after parsing the query string with QueryHelpers.ParseQuery.

```
@page "/"
@using Microsoft.AspNetCore.WebUtilities
@inject NavigationManager NavigationManager

<h1>Query string parse example</h1>

<p>Value: @queryValue</p>

@code {
    private string queryValue = "Not set";

    protected override void OnInitialized()
    {
        var query = new Uri(NavigationManager.Uri).Query;

        if (QueryHelpers.ParseQuery(query).TryGetValue("{KEY}", out var value))
        {
            queryValue = value;
        }
    }
}
```

The placeholder `{KEY}` in the preceding example is the query string parameter key. For example, the URL key-value pair `?ship=Tardis` uses a key of `ship`.

ASP.NET Core Blazor routing

9/22/2020 • 9 minutes to read • Edit Online

By Luke Latham

Learn how to route requests and how to use the NavLink component to create navigation links in Blazor apps.

ASP.NET Core endpoint routing integration

Blazor Server is integrated into ASP.NET Core Endpoint Routing. An ASP.NET Core app is configured to accept incoming connections for interactive components with MapBlazorHub in `Startup.Configure`:

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

The most typical configuration is to route all requests to a Razor page, which acts as the host for the server-side part of the Blazor Server app. By convention, the *host* page is usually named `_Host.cshtml`. The route specified in the host file is called a *fallback route* because it operates with a low priority in route matching. The fallback route is considered when other routes don't match. This allows the app to use others controllers and pages without interfering with the Blazor Server app.

For information on configuring MapFallbackToPage for non-root URL server hosting, see Host and deploy ASP.NET Core Blazor.

Route templates

The Router component enables routing to each component with a specified route. The Router component appears in the `App.razor` file:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

When a `.razor` file with an `@page` directive is compiled, the generated class is provided a RouteAttribute specifying the route template.

At runtime, the RouteView component:

- Receives the RouteData from the Router along with any desired parameters.
- Renders the specified component with its layout (or an optional default layout) using the specified parameters.

You can optionally specify a DefaultLayout parameter with a layout class to use for components that don't specify a layout. The default Blazor templates specify the `MainLayout` component. `MainLayout.razor` is in the template

project's `Shared` folder. For more information on layouts, see ASP.NET Core Blazor layouts.

Multiple route templates can be applied to a component. The following component responds to requests for `/BlazorRoute` and `/DifferentBlazorRoute`:

```
@page "/BlazorRoute"
@page "/DifferentBlazorRoute"

<h1>Blazor routing</h1>
```

IMPORTANT

For URLs to resolve correctly, the app must include a `<base>` tag in its `wwwroot/index.html` file (Blazor WebAssembly) or `Pages/_Host.cshtml` file (Blazor Server) with the app base path specified in the `href` attribute (`<base href="/">`). For more information, see Host and deploy ASP.NET Core Blazor.

Provide custom content when content isn't found

The Router component allows the app to specify custom content if content isn't found for the requested route.

In the `App.razor` file, set custom content in the NotFound template parameter of the Router component:

```
<Router AppAssembly="typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <h1>Sorry</h1>
        <p>Sorry, there's nothing at this address.</p> b
    </NotFound>
</Router>
```

The content of `<NotFound>` tags can include arbitrary items, such as other interactive components. To apply a default layout to NotFound content, see ASP.NET Core Blazor layouts.

Route to components from multiple assemblies

Use the AdditionalAssemblies parameter to specify additional assemblies for the Router component to consider when searching for routable components. Specified assemblies are considered in addition to the `AppAssembly`-specified assembly. In the following example, `Component1` is a routable component defined in a referenced class library. The following AdditionalAssemblies example results in routing support for `Component1`:

```
<Router
    AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="new[] { typeof(Component1).Assembly }">
    ...
</Router>
```

Route parameters

The router uses route parameters to populate the corresponding component parameters with the same name (case insensitive):

```
@page "/RouteParameter"
@page "/RouteParameter/{text}"

<h1>Blazor is @Text!</h1>

@code {
    [Parameter]
    public string Text { get; set; }

    protected override void OnInitialized()
    {
        Text = Text ?? "fantastic";
    }
}
```

Optional parameters aren't supported. Two `@page` directives are applied in the previous example. The first permits navigation to the component without a parameter. The second `@page` directive takes the `{text}` route parameter and assigns the value to the `Text` property.

Route constraints

A route constraint enforces type matching on a route segment to a component.

In the following example, the route to the `Users` component only matches if:

- An `Id` route segment is present on the request URL.
- The `Id` segment is an integer (`int`).

```
@page "/Users/{Id:int}"

<h1>The user Id is @Id!</h1>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

The route constraints shown in the following table are available. For the route constraints that match with the invariant culture, see the warning below the table for more information.

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`bool`	`{active:bool}`	`true`, `FALSE`	No
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Yes
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Yes
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Yes
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Yes

CONSTRAINT	EXAMPLE	EXAMPLE MATCHES	INVARIANT CULTURE MATCHING
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`, `{CD2C1638-1638-72D5-1638-DEADBEEF1638}`	No
`int`	`{id:int}`	`123456789`, `-123456789`	Yes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Yes

> **WARNING**
> Route constraints that verify the URL and are converted to a CLR type (such as `int` or DateTime) always use the invariant culture. These constraints assume that the URL is non-localizable.

Routing with URLs that contain dots

For hosted Blazor WebAssembly and Blazor Server apps, the server-side default route template assumes that if the last segment of a request URL contains a dot (`.`) that a file is requested (for example, `https://localhost.com:5001/example/some.thing`). Without additional configuration, an app returns a *404 - Not Found* response if this was meant to route to a component. To use a route with one or more parameters that contains a dot, the app must configure the route with a custom template.

Consider the following `Example` component that can receive a route parameter from the last segment of the URL:

```
@page "/example"
@page "/example/{param}"

<p>
    Param: @Param
</p>

@code {
    [Parameter]
    public string Param { get; set; }
}
```

To permit the *Server* app of a hosted Blazor WebAssembly solution to route the request with a dot in the `param` parameter, add a fallback file route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToFile("/example/{param?}", "index.html");
```

To configure a Blazor Server app to route the request with a dot in the `param` parameter, add a fallback page route template with the optional parameter in `Startup.Configure` (`Startup.cs`):

```
endpoints.MapFallbackToPage("/example/{param?}", "/_Host");
```

For more information, see Routing in ASP.NET Core.

Catch-all route parameters

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components. The catch-all route parameter must be:

- Named to match the route segment name. Naming isn't case sensitive.
- A `string` type. The framework doesn't provide automatic casting.
- At the end of the URL.

```
@page "/page/{*pageRoute}"

@code {
    [Parameter]
    public string PageRoute { get; set; }
}
```

For the URL `/page/this/is/a/test` with a route template of `/page/{*pageRoute}`, the value of `PageRoute` is set to `this/is/a/test`.

Slashes and segments of the captured path are decoded. For a route template of `/page/{*pageRoute}`, the URL `/page/this/is/a%2Ftest%2A` yields `this/is/a/test*`.

Catch-all route parameters are supported in ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.*

NavLink component

Use a NavLink component in place of HTML hyperlink elements (`<a>`) when creating navigation links. A NavLink component behaves like an `<a>` element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed. Optionally, assign a CSS class name to NavLink.ActiveClass to apply a custom CSS class to the rendered link when the current route matches the `href`.

The following `NavMenu` component creates a `Bootstrap` navigation bar that demonstrates how to use NavLink components:

```
<div class="@NavMenuCssClass" @onclick="@ToggleNavMenu">
    <ul class="nav flex-column">
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="oi oi-home" aria-hidden="true"></span> Home
            </NavLink>
        </li>
        <li class="nav-item px-3">
            <NavLink class="nav-link" href="MyComponent" Match="NavLinkMatch.Prefix">
                <span class="oi oi-plus" aria-hidden="true"></span> My Component
            </NavLink>
        </li>
    </ul>
</div>
```

There are two NavLinkMatch options that you can assign to the `Match` attribute of the `<NavLink>` element:

- NavLinkMatch.All: The NavLink is active when it matches the entire current URL.
- NavLinkMatch.Prefix (*default*): The NavLink is active when it matches any prefix of the current URL.

In the preceding example, the Home NavLink `href=""` matches the home URL and only receives the `active` CSS class at the app's default base path URL (for example, `https://localhost:5001/`). The second NavLink receives the `active` class when the user visits any URL with a `MyComponent` prefix (for example,

`https://localhost:5001/MyComponent` and `https://localhost:5001/MyComponent/AnotherSegment`).

Additional NavLink component attributes are passed through to the rendered anchor tag. In the following example, the NavLink component includes the `target` attribute:

```
<NavLink href="my-page" target="_blank">My page</NavLink>
```

The following HTML markup is rendered:

```
<a href="my-page" target="_blank">My page</a>
```

WARNING

Due to the way that Blazor renders child content, rendering `NavLink` components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the `NavLink` (child) component's content:

```
@for (int c = 0; c < 10; c++)
{
    var current = c;
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @current
        </NavLink>
    </li>
}
```

Using an index variable in this scenario is a requirement for **any** child component that uses a loop variable in its child content, not just the `NavLink` component.

Alternatively, use a `foreach` loop with Enumerable.Range:

```
@foreach(var c in Enumerable.Range(0,10))
{
    <li ...>
        <NavLink ... href="@c">
            <span ...></span> @c
        </NavLink>
    </li>
}
```

URI and navigation state helpers

Use NavigationManager to work with URIs and navigation in C# code. NavigationManager provides the event and methods shown in the following table.

MEMBER	DESCRIPTION
Uri	Gets the current absolute URI.
BaseUri	Gets the base URI (with a trailing slash) that can be prepended to relative URI paths to produce an absolute URI. Typically, BaseUri corresponds to the `href` attribute on the document's `<base>` element in `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server).

MEMBER	DESCRIPTION
NavigateTo	Navigates to the specified URI. If `forceLoad` is `true`: • Client-side routing is bypassed. • The browser is forced to load the new page from the server, whether or not the URI is normally handled by the client-side router.
LocationChanged	An event that fires when the navigation location has changed.
ToAbsoluteUri	Converts a relative URI into an absolute URI.
ToBaseRelativePath	Given a base URI (for example, a URI previously returned by BaseUri), converts an absolute URI into a URI relative to the base URI prefix.

The following component navigates to the app's `Counter` component when the button is selected:

```
@page "/navigate"
@inject NavigationManager NavigationManager

<h1>Navigate in Code Example</h1>

<button class="btn btn-primary" @onclick="NavigateToCounterComponent">
    Navigate to the Counter component
</button>

@code {
    private void NavigateToCounterComponent()
    {
        NavigationManager.NavigateTo("counter");
    }
}
```

The following component handles a location changed event by subscribing to NavigationManager.LocationChanged. The `HandleLocationChanged` method is unhooked when `Dispose` is called by the framework. Unhooking the method permits garbage collection of the component.

```
@implements IDisposable
@inject NavigationManager NavigationManager

...

protected override void OnInitialized()
{
    NavigationManager.LocationChanged += HandleLocationChanged;
}

private void HandleLocationChanged(object sender, LocationChangedEventArgs e)
{
    ...
}

public void Dispose()
{
    NavigationManager.LocationChanged -= HandleLocationChanged;
}
```

LocationChangedEventArgs provides the following information about the event:

- Location: The URL of the new location.
- IsNavigationIntercepted: If `true`, Blazor intercepted the navigation from the browser. If `false`, NavigationManager.NavigateTo caused the navigation to occur.

For more information on component disposal, see ASP.NET Core Blazor lifecycle.

Query string and parse parameters

The query string of a request can be obtained from the NavigationManager's Uri property:

```
@inject NavigationManager Navigation

...

var query = new Uri(Navigation.Uri).Query;
```

To parse a query string's parameters:

- Add a package reference for Microsoft.AspNetCore.WebUtilities.
- Obtain the value after parsing the query string with QueryHelpers.ParseQuery.

```
@page "/"
@using Microsoft.AspNetCore.WebUtilities
@inject NavigationManager NavigationManager

<h1>Query string parse example</h1>

<p>Value: @queryValue</p>

@code {
    private string queryValue = "Not set";

    protected override void OnInitialized()
    {
        var query = new Uri(NavigationManager.Uri).Query;

        if (QueryHelpers.ParseQuery(query).TryGetValue("{KEY}", out var value))
        {
            queryValue = value;
        }
    }
}
```

The placeholder `{KEY}` in the preceding example is the query string parameter key. For example, the URL key-value pair `?ship=Tardis` uses a key of `ship`.

ASP.NET Core Blazor hosting model configuration

9/22/2020 • 7 minutes to read • Edit Online

By Daniel Roth, Mackinnon Buck, and Luke Latham

This article covers hosting model configuration.

SignalR cross-origin negotiation for authentication

This section applies to Blazor WebAssembly.

To configure SignalR's underlying client to send credentials, such as cookies or HTTP authentication headers:

- Use SetBrowserRequestCredentials to set Include on cross-origin `fetch` requests:

```
public class IncludeRequestCredentialsMessageHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        return base.SendAsync(request, cancellationToken);
    }
}
```

- Assign the HttpMessageHandler to the HttpMessageHandlerFactory option:

```
var connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://signalr.example.com"), options =>
    {
        options.HttpMessageHandlerFactory = innerHandler =>
            new IncludeRequestCredentialsMessageHandler { InnerHandler = innerHandler };
    }).Build();
```

For more information, see ASP.NET Core SignalR configuration.

Reflect the connection state in the UI

This section applies to Blazor Server.

When the client detects that the connection has been lost, a default UI is displayed to the user while the client attempts to reconnect. If reconnection fails, the user is provided the option to retry.

To customize the UI, define an element with an `id` of `components-reconnect-modal` in the `<body>` of the `_Host.cshtml` Razor page:

```
<div id="components-reconnect-modal">
    ...
</div>
```

Add the following to the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`):

```
#components-reconnect-modal {
    display: none;
}

#components-reconnect-modal.components-reconnect-show {
    display: block;
}
```

The following table describes the CSS classes applied to the `components-reconnect-modal` element.

CSS CLASS	INDICATES...
`components-reconnect-show`	A lost connection. The client is attempting to reconnect. Show the modal.
`components-reconnect-hide`	An active connection is re-established to the server. Hide the modal.
`components-reconnect-failed`	Reconnection failed, probably due to a network failure. To attempt reconnection, call `window.Blazor.reconnect()`.
`components-reconnect-rejected`	Reconnection rejected. The server was reached but refused the connection, and the user's state on the server is lost. To reload the app, call `location.reload()`. This connection state may result when: • A crash in the server-side circuit occurs. • The client is disconnected long enough for the server to drop the user's state. Instances of the components that the user is interacting with are disposed. • The server is restarted, or the app's worker process is recycled.

Render mode

This section applies to Blazor Server.

Blazor Server apps are set up by default to prerender the UI on the server before the client connection to the server is established. This is set up in the `_Host.cshtml` Razor page:

```
<body>
    <app>
      <component type="typeof(App)" render-mode="ServerPrerendered" />
    </app>

    <script src="_framework/blazor.server.js"></script>
</body>
```

RenderMode configures whether the component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

RENDER MODE	DESCRIPTION
ServerPrerendered	Renders the component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a Blazor Server app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the component into static HTML.

Rendering server components from a static HTML page isn't supported.

Initialize the Blazor circuit

This section applies to Blazor Server.

Configure the manual start of a Blazor Server app's SignalR circuit in the `Pages/_Host.cshtml` file:

- Add an `autostart="false"` attribute to the `<script>` tag for the `blazor.server.js` script.
- Place a script that calls `Blazor.start` after the `blazor.server.js` script's tag and inside the closing `</body>` tag.

When `autostart` is disabled, any aspect of the app that doesn't depend on the circuit works normally. For example, client-side routing is operational. However, any aspect that depends on the circuit isn't operational until `Blazor.start` is called. App behavior is unpredictable without an established circuit. For example, component methods fail to execute while the circuit is disconnected.

Initialize Blazor when the document is ready

To initialize the Blazor app when the document is ready:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      document.addEventListener("DOMContentLoaded", function() {
        Blazor.start();
      });
    </script>
</body>
```

Chain to the `Promise` that results from a manual start

To perform additional tasks, such as JS interop initialization, use `then` to chain to the `Promise` that results from a manual Blazor app start:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start().then(function () {
        ...
      });
    </script>
</body>
```

Configure the SignalR client

Logging

To configure SignalR client logging, pass in a configuration object (`configureSignalR`) that calls `configureLogging` with the log level on the client builder:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        configureSignalR: function (builder) {
          builder.configureLogging("information");
        }
      });
    </script>
</body>
```

In the preceding example, `information` is equivalent to a log level of LogLevel.Information.

Modify the reconnection handler

The reconnection handler's circuit connection events can be modified for custom behaviors, such as:

- To notify the user if the connection is dropped.
- To perform logging (from the client) when a circuit is connected.

To modify the connection events, register callbacks for the following connection changes:

- Dropped connections use `onConnectionDown` .
- Established/re-established connections use `onConnectionUp` .

Both `onConnectionDown` and `onConnectionUp` must be specified:

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionHandler: {
          onConnectionDown: (options, error) => console.error(error);
          onConnectionUp: () => console.log("Up, up, and away!");
        }
      });
    </script>
</body>
```

Adjust the reconnection retry count and interval

To adjust the reconnection retry count and interval, set the number of retries (`maxRetries`) and period in milliseconds permitted for each retry attempt (`retryIntervalMilliseconds`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      Blazor.start({
        reconnectionOptions: {
          maxRetries: 3,
          retryIntervalMilliseconds: 2000
        }
      });
    </script>
</body>
```

Hide or replace the reconnection display

To hide the reconnection display, set the reconnection handler's `_reconnectionDisplay` to an empty object (`{}` or `new Object()`):

```
<body>

    ...

    <script autostart="false" src="_framework/blazor.server.js"></script>
    <script>
      window.addEventListener('beforeunload', function () {
        Blazor.defaultReconnectionHandler._reconnectionDisplay = {};
      });

      Blazor.start();
    </script>
</body>
```

To replace the reconnection display, set `_reconnectionDisplay` in the preceding example to the element for display:

```
Blazor.defaultReconnectionHandler._reconnectionDisplay =
  document.getElementById("{ELEMENT ID}");
```

The placeholder `{ELEMENT ID}` is the ID of the HTML element to display.

Customize the delay before the reconnection display appears by setting the `transition-delay` property in the app's CSS (`wwwroot/css/site.css`) for the modal element. The following example sets the transition delay from 500 ms (default) to 1,000 ms (1 second):

```
#components-reconnect-modal {
    transition: visibility 0s linear 1000ms;
}
```

Influence HTML `<head>` tag elements

This section applies to the upcoming ASP.NET Core 5.0 release of Blazor WebAssembly and Blazor Server.

When rendered, the `Title` , `Link` , and `Meta` components add or update data in the HTML `<head>` tag elements:

```
@using Microsoft.AspNetCore.Components.Web.Extensions.Head

<Title Value="{TITLE}" />
<Link href="{URL}" rel="stylesheet" />
<Meta content="{DESCRIPTION}" name="description" />
```

In the preceding example, placeholders for `{TITLE}` , `{URL}` , and `{DESCRIPTION}` are string values, Razor variables, or Razor expressions.

The following characteristics apply:

- Server-side prerendering is supported.
- The `Value` parameter is the only valid parameter for the `Title` component.
- HTML attributes provided to the `Meta` and `Link` components are captured in additional attributes and passed through to the rendered HTML tag.
- For multiple `Title` components, the title of the page reflects the `Value` of the last `Title` component rendered.
- If multiple `Meta` or `Link` components are included with identical attributes, there's exactly one HTML tag rendered per `Meta` or `Link` component. Two `Meta` or `Link` components can't refer to the same rendered HTML tag.
- Changes to the parameters of existing `Meta` or `Link` components are reflected in their rendered HTML tags.
- When the `Link` or `Meta` components are no longer rendered and thus disposed by the framework, their rendered HTML tags are removed.

When one of the framework components is used in a child component, the rendered HTML tag influences any other child component of the parent component as long as the child component containing the framework component is rendered. The distinction between using the one of these framework components in a child component and placing a an HTML tag in `wwwroot/index.html` or `Pages/_Host.cshtml` is that a framework component's rendered HTML tag:

- Can be modified by application state. A hard-coded HTML tag can't be modified by application state.
- Is removed from the HTML `<head>` when the parent component is no longer rendered.

Static files

This section applies to Blazor Server.

To create additional file mappings with a FileExtensionContentTypeProvider or configure other StaticFileOptions, use **one** of the following approaches. In the following examples, the `{EXTENSION}` placeholder is the file extension,

and the `{CONTENT TYPE}` placeholder is the content type.

- Configure options through dependency injection (DI) in `Startup.ConfigureServices` (`Startup.cs`) using StaticFileOptions:

  ```
  using Microsoft.AspNetCore.StaticFiles;

  ...

  var provider = new FileExtensionContentTypeProvider();
  provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

  services.Configure<StaticFileOptions>(options =>
  {
      options.ContentTypeProvider = provider;
  });
  ```

  Because this approach configures the same file provider used to serve `blazor.server.js`, make sure that your custom configuration doesn't interfere with serving `blazor.server.js`. For example, don't remove the mapping for JavaScript files by configuring the provider with `provider.Mappings.Remove(".js")`.

- Use two calls to UseStaticFiles in `Startup.Configure` (`Startup.cs`):

  - Configure the custom file provider in the first call with StaticFileOptions.
  - The second middleware serves `blazor.server.js`, which uses the default static files configuration provided by the Blazor framework.

  ```
  using Microsoft.AspNetCore.StaticFiles;

  ...

  var provider = new FileExtensionContentTypeProvider();
  provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

  app.UseStaticFiles(new StaticFileOptions { ContentTypeProvider = provider });
  app.UseStaticFiles();
  ```

Additional resources

- Logging in .NET Core and ASP.NET Core

ASP.NET Core Blazor component virtualization

9/22/2020 • 3 minutes to read • Edit Online

By Daniel Roth

Improve the perceived performance of component rendering using the Blazor framework's built-in virtualization support. Virtualization is a technique for limiting UI rendering to just the parts that are currently visible. For example, virtualization is helpful when the app must render a long list or a table with many rows and only a subset of items is required to be visible at any given time. Blazor provides the `Virtualize` component that can be used to add virtualization to an app's components.

Without virtualization, a typical list or table-based component might use a C# `foreach` loop to render each item in the list or each row in the table:

```
<table>
    @foreach (var employee in employees)
    {
        <tr>
            <td>@employee.FirstName</td>
            <td>@employee.LastName</td>
            <td>@employee.JobTitle</td>
        </tr>
    }
</table>
```

If the list contains thousands of items, then rendering the list may take a long time. The user may experience a noticeable UI lag.

Instead of rendering each item in the list all at one time, replace the `foreach` loop with the `Virtualize` component and specify a fixed item source with `Items`. Only the items that are currently visible are rendered:

```
<table>
    <Virtualize Context="employee" Items="@employees">
        <tr>
            <td>@employee.FirstName</td>
            <td>@employee.LastName</td>
            <td>@employee.JobTitle</td>
        </tr>
    </Virtualize>
</table>
```

If not specifying a context to the component with `Context`, use the `context` value (`@context.{PROPERTY}`) in the item content template:

```
<table>
    <Virtualize Items="@employees">
        <tr>
            <td>@context.FirstName</td>
            <td>@context.LastName</td>
            <td>@context.JobTitle</td>
        </tr>
    </Virtualize>
</table>
```

The `Virtualize` component calculates how many items to render based on the height of the container and the size of the rendered items.

Item provider delegate

If you don't want to load all of the items into memory, you can specify an items provider delegate method to the component's `ItemsProvider` parameter that asynchronously retrieves the requested items on demand:

```
<table>
    <Virtualize Context="employee" ItemsProvider="@LoadEmployees">
         <tr>
            <td>@employee.FirstName</td>
            <td>@employee.LastName</td>
            <td>@employee.JobTitle</td>
        </tr>
    </Virtualize>
</table>
```

The items provider receives an `ItemsProviderRequest`, which specifies the required number of items starting at a specific start index. The items provider then retrieves the requested items from a database or other service and returns them as an `ItemsProviderResult<TItem>` along with a count of the total items. The items provider can choose to retrieve the items with each request or cache them so that they're readily available. Don't attempt to use an items provider and assign a collection to `Items` for the same `Virtualize` component.

The following example loads employees from an `EmployeeService`:

```
private async ValueTask<ItemsProviderResult<Employee>> LoadEmployees(
    ItemsProviderRequest request)
{
    var numEmployees = Math.Min(request.Count, totalEmployees - request.StartIndex);
    var employees = await EmployeesService.GetEmployeesAsync(request.StartIndex,
        numEmployees, request.CancellationToken);

    return new ItemsProviderResult<Employee>(employees, totalEmployees);
}
```

Placeholder

Because requesting items from a remote data source might take some time, you have the option to render a placeholder (`<Placeholder>...</Placeholder>`) until the item data is available:

```
<table>
    <Virtualize Context="employee" ItemsProvider="@LoadEmployees">
        <ItemContent>
            <tr>
                <td>@employee.FirstName</td>
                <td>@employee.LastName</td>
                <td>@employee.JobTitle</td>
            </tr>
        </ItemContent>
        <Placeholder>
            <tr>
                <td>Loading...</td>
            </tr>
        </Placeholder>
    </Virtualize>
</table>
```

Item size

The size of each item in pixels can be set with `ItemSize` (default: 50px):

```
<table>
    <Virtualize Context="employee" Items="@employees" ItemSize="25">
        ...
    </Virtualize>
</table>
```

Overscan count

`OverscanCount` determines how many additional items are rendered before and after the visible region. This setting helps to reduce the frequency of rendering during scrolling. However, higher values result in more elements rendered in the page (default: 3):

```
<table>
    <Virtualize Context="employee" Items="@employees" OverscanCount="4">
        ...
    </Virtualize>
</table>
```

For example, a grid or list that renders hundreds of rows containing components is processor intensive to render. Consider virtualizing a grid or list layout so that only a subset of the components is rendered at any given time. For an example of component subset rendering, see the following components in the `Virtualization` sample app (aspnet/samples GitHub repository):

- `Virtualize` component (`Shared/Virtualize.razor`): A component written in C# that implements ComponentBase to render a set of weather data rows based on user scrolling.
- `FetchData` component (`Pages/FetchData.razor`): Uses the `Virtualize` component to display 25 rows of weather data at a time.

State changes

When making changes to items rendered by the `Virtualize` component, call StateHasChanged to force re-evaluation and rerendering of the component.

ASP.NET Core Blazor cascading values and parameters

9/22/2020 • 4 minutes to read • Edit Online

By Luke Latham and Daniel Roth

View or download sample code (how to download)

In some scenarios, it's inconvenient to flow data from an ancestor component to a descendent component using component parameters, especially when there are several component layers. Cascading values and parameters solve this problem by providing a convenient way for an ancestor component to provide a value to all of its descendent components. Cascading values and parameters also provide an approach for components to coordinate.

Theme example

In the following example from the sample app, the `ThemeInfo` class specifies the theme information to flow down the component hierarchy so that all of the buttons within a given part of the app share the same style.

`UIThemeClasses/ThemeInfo.cs`:

```
public class ThemeInfo
{
    public string ButtonClass { get; set; }
}
```

An ancestor component can provide a cascading value using the Cascading Value component. The CascadingValue<TValue> component wraps a subtree of the component hierarchy and supplies a single value to all components within that subtree.

For example, the sample app specifies theme information (`ThemeInfo`) in one of the app's layouts as a cascading parameter for all components that make up the layout body of the `@Body` property. `ButtonClass` is assigned a value of `btn-success` in the layout component. Any descendent component can consume this property through the `ThemeInfo` cascading object.

`CascadingValuesParametersLayout` component:

```
@inherits LayoutComponentBase
@using BlazorSample.UIThemeClasses

<div class="container-fluid">
    <div class="row">
        <div class="col-sm-3">
            <NavMenu />
        </div>
        <div class="col-sm-9">
            <CascadingValue Value="theme">
                <div class="content px-4">
                    @Body
                </div>
            </CascadingValue>
        </div>
    </div>
</div>

@code {
    private ThemeInfo theme = new ThemeInfo { ButtonClass = "btn-success" };
}
```

To make use of cascading values, components declare cascading parameters using the `[CascadingParameter]` attribute. Cascading values are bound to cascading parameters by type.

In the sample app, the `CascadingValuesParametersTheme` component binds the `ThemeInfo` cascading value to a cascading parameter. The parameter is used to set the CSS class for one of the buttons displayed by the component.

`CascadingValuesParametersTheme` component:

```
@page "/cascadingvaluesparameterstheme"
@layout CascadingValuesParametersLayout
@using BlazorSample.UIThemeClasses

<h1>Cascading Values & Parameters</h1>

<p>Current count: @currentCount</p>

<p>
    <button class="btn" @onclick="IncrementCount">
        Increment Counter (Unthemed)
    </button>
</p>

<p>
    <button class="btn @ThemeInfo.ButtonClass" @onclick="IncrementCount">
        Increment Counter (Themed)
    </button>
</p>

@code {
    private int currentCount = 0;

    [CascadingParameter]
    protected ThemeInfo ThemeInfo { get; set; }

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

To cascade multiple values of the same type within the same subtree, provide a unique Name string to each CascadingValue<TValue> component and its corresponding `[CascadingParameter]` attribute. In the following example, two CascadingValue<TValue> components cascade different instances of `MyCascadingType` by name:

```
<CascadingValue Value="@parentCascadeParameter1" Name="CascadeParam1">
    <CascadingValue Value="@ParentCascadeParameter2" Name="CascadeParam2">
        ...
    </CascadingValue>
</CascadingValue>

@code {
    private MyCascadingType parentCascadeParameter1;

    [Parameter]
    public MyCascadingType ParentCascadeParameter2 { get; set; }

    ...
}
```

In a descendant component, the cascaded parameters receive their values from the corresponding cascaded values in the ancestor component by name:

```
...

@code {
    [CascadingParameter(Name = "CascadeParam1")]
    protected MyCascadingType ChildCascadeParameter1 { get; set; }

    [CascadingParameter(Name = "CascadeParam2")]
    protected MyCascadingType ChildCascadeParameter2 { get; set; }
}
```

TabSet example

Cascading parameters also enable components to collaborate across the component hierarchy. For example, consider the following `TabSet` example in the sample app.

The sample app has an `ITab` interface that tabs implement:

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.UIInterfaces
{
    public interface ITab
    {
        RenderFragment ChildContent { get; }
    }
}
```

The `CascadingValuesParametersTabSet` component uses the `TabSet` component, which contains several `Tab` components:

```
@page "/CascadingValuesParametersTabSet"

<TabSet>
    <Tab Title="First tab">
        <h4>Greetings from the first tab!</h4>

        <label>
            <input type="checkbox" @bind="showThirdTab" />
            Toggle third tab
        </label>
    </Tab>
    <Tab Title="Second tab">
        <h4>The second tab says Hello World!</h4>
    </Tab>

    @if (showThirdTab)
    {
        <Tab Title="Third tab">
            <h4>Welcome to the disappearing third tab!</h4>
            <p>Toggle this tab from the first tab.</p>
        </Tab>
    }
</TabSet>

@code {
    private bool showThirdTab;
}
```

The child `Tab` components aren't explicitly passed as parameters to the `TabSet`. Instead, the child `Tab` components are part of the child content of the `TabSet`. However, the `TabSet` still needs to know about each `Tab` component so that it can render the headers and the active tab. To enable this coordination without requiring additional code, the `TabSet` component *can provide itself as a cascading value* that is then picked up by the descendent `Tab` components.

`TabSet` component:

```
@using BlazorSample.UIInterfaces

<!-- Display the tab headers -->
<CascadingValue Value=this>
    <ul class="nav nav-tabs">
        @ChildContent
    </ul>
</CascadingValue>

<!-- Display body for only the active tab -->
<div class="nav-tabs-body p-4">
    @ActiveTab?.ChildContent
</div>

@code {
    [Parameter]
    public RenderFragment ChildContent { get; set; }

    public ITab ActiveTab { get; private set; }

    public void AddTab(ITab tab)
    {
        if (ActiveTab == null)
        {
            SetActiveTab(tab);
        }
    }

    public void SetActiveTab(ITab tab)
    {
        if (ActiveTab != tab)
        {
            ActiveTab = tab;
            StateHasChanged();
        }
    }
}
```

The descendent `Tab` components capture the containing `TabSet` as a cascading parameter, so the `Tab` components add themselves to the `TabSet` and coordinate on which tab is active.

`Tab` component:

```
@using BlazorSample.UIInterfaces
@implements ITab

<li>
    <a @onclick="ActivateTab" class="nav-link @TitleCssClass" role="button">
        @Title
    </a>
</li>

@code {
    [CascadingParameter]
    public TabSet ContainerTabSet { get; set; }

    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    private string TitleCssClass => ContainerTabSet.ActiveTab == this ? "active" : null;

    protected override void OnInitialized()
    {
        ContainerTabSet.AddTab(this);
    }

    private void ActivateTab()
    {
        ContainerTabSet.SetActiveTab(this);
    }
}
```

ASP.NET Core Blazor data binding

9/22/2020 • 6 minutes to read • Edit Online

By Luke Latham and Daniel Roth

Razor components provide data binding features via an HTML element attribute named `@bind` with a field, property, or Razor expression value.

The following example binds an `<input>` element to the `currentValue` field and an `<input>` element to the `CurrentValue` property:

```
<p>
    <input @bind="currentValue" /> Current value: @currentValue
</p>

<p>
    <input @bind="CurrentValue" /> Current value: @CurrentValue
</p>

@code {
    private string currentValue;

    private string CurrentValue { get; set; }
}
```

When one of the elements looses focus, its bound field or property is updated.

The text box is updated in the UI only when the component is rendered, not in response to changing the field's or property's value. Since components render themselves after event handler code executes, field and property updates are *usually* reflected in the UI immediately after an event handler is triggered.

Using `@bind` with the `CurrentValue` property (`<input @bind="CurrentValue" />`) is essentially equivalent to the following:

```
<input value="@CurrentValue"
    @onchange="@((ChangeEventArgs __e) => CurrentValue =
        __e.Value.ToString())" />

@code {
    private string CurrentValue { get; set; }
}
```

When the component is rendered, the `value` of the input element comes from the `CurrentValue` property. When the user types in the text box and changes element focus, the `onchange` event is fired and the `CurrentValue` property is set to the changed value. In reality, the code generation is more complex than that because `@bind` handles cases where type conversions are performed. In principle, `@bind` associates the current value of an expression with a `value` attribute and handles changes using the registered handler.

Bind a property or field on other events by also including an `@bind:event` attribute with an `event` parameter. The following example binds the `CurrentValue` property on the `oninput` event:

```
<input @bind="CurrentValue" @bind:event="oninput" />

@code {
    private string CurrentValue { get; set; }
}
```

Unlike `onchange`, which fires when the element loses focus, `oninput` fires when the value of the text box changes.

Attribute binding is case sensitive:

- `@bind` is valid.
- `@Bind` and `@BIND` are invalid.

Unparsable values

When a user provides an unparsable value to a databound element, the unparsable value is automatically reverted to its previous value when the bind event is triggered.

Consider the following scenario:

- An `<input>` element is bound to an `int` type with an initial value of `123`:

  ```
  <input @bind="inputValue" />

  @code {
      private int inputValue = 123;
  }
  ```

- The user updates the value of the element to `123.45` in the page and changes the element focus.

In the preceding scenario, the element's value is reverted to `123`. When the value `123.45` is rejected in favor of the original value of `123`, the user understands that their value wasn't accepted.

By default, binding applies to the element's `onchange` event (`@bind="{PROPERTY OR FIELD}"`). Use `@bind="{PROPERTY OR FIELD}" @bind:event={EVENT}` to trigger binding on a different event. For the `oninput` event (`@bind:event="oninput"`), the reversion occurs after any keystroke that introduces an unparsable value. When targeting the `oninput` event with an `int`-bound type, a user is prevented from typing a `.` character. A `.` character is immediately removed, so the user receives immediate feedback that only whole numbers are permitted. There are scenarios where reverting the value on the `oninput` event isn't ideal, such as when the user should be allowed to clear an unparsable `<input>` value. Alternatives include:

- Don't use the `oninput` event. Use the default `onchange` event (only specify `@bind="{PROPERTY OR FIELD}"`), where an invalid value isn't reverted until the element loses focus.
- Bind to a nullable type, such as `int?` or `string` and provide custom logic to handle invalid entries.
- Use a form validation component, such as InputNumber<TValue> or InputDate<TValue>. Form validation components have built-in support to manage invalid inputs. For more information, see ASP.NET Core Blazor forms and validation. Form validation components:
  - Permit the user to provide invalid input and receive validation errors on the associated EditContext.
  - Display validation errors in the UI without interfering with the user entering additional webform data.

Format strings

Data binding works with DateTime format strings using `@bind:format`. Other format expressions, such as currency or number formats, aren't available at this time.

```
<input @bind="startDate" @bind:format="yyyy-MM-dd" />

@code {
    private DateTime startDate = new DateTime(2020, 1, 1);
}
```

In the preceding code, the `<input>` element's field type (`type`) defaults to `text`. `@bind:format` is supported for binding the following .NET types:

- System.DateTime
- System.DateTime?
- System.DateTimeOffset
- System.DateTimeOffset?

The `@bind:format` attribute specifies the date format to apply to the `value` of the `<input>` element. The format is also used to parse the value when an `onchange` event occurs.

Specifying a format for the `date` field type isn't recommended because Blazor has built-in support to format dates. In spite of the recommendation, only use the `yyyy-MM-dd` date format for binding to function correctly if a format is supplied with the `date` field type:

```
<input type="date" @bind="startDate" @bind:format="yyyy-MM-dd">
```

Parent-to-child binding with component parameters

Component parameters permit binding properties and fields of a parent component with `@bind-{PROPERTY OR FIELD}` syntax.

The following `Child` component (`Shared/Child.razor`) has a `Year` component parameter and `YearChanged` callback:

```
<div class="card bg-light mt-3" style="width:18rem ">
    <div class="card-body">
        <h3 class="card-title">Child Component</h3>
        <p class="card-text">Child <code>Year</code>: @Year</p>
    </div>
</div>

@code {
    [Parameter]
    public int Year { get; set; }

    [Parameter]
    public EventCallback<int> YearChanged { get; set; }
}
```

The callback (EventCallback<TValue>) must be named as the component parameter name followed by the "`Changed`" suffix (`{PARAMETER NAME}Changed`). In the preceding example, the callback is named `YearChanged`. For more information on EventCallback<TValue>, see ASP.NET Core Blazor event handling.

In the following `Parent` component (`Parent.razor`), the `year` field is bound to the `Year` parameter of the child component:

```
@page "/Parent"

<h1>Parent Component</h1>

<p>Parent <code>year</code>: @year</p>

<button @onclick="UpdateYear">Update Parent <code>year</code></button>

<Child @bind-Year="year" />

@code {
    private Random r = new Random();
    private int year = 1979;

    private void UpdateYear()
    {
        year = r.Next(1950, 2021);
    }
}
```

The `Year` parameter is bindable because it has a companion `YearChanged` event that matches the type of the `Year` parameter.

By convention, a property can be bound to a corresponding event handler by including an `@bind-{PROPERTY}:event` attribute assigned to the handler. `<Child @bind-Year="year" />` is equivalent to writing:

```
<Child @bind-Year="year" @bind-Year:event="YearChanged" />
```

Child-to-parent binding with chained bind

A common scenario is chaining a data-bound parameter to a page element in the component's output. This scenario is called a *chained bind* because multiple levels of binding occur simultaneously.

A chained bind can't be implemented with `@bind` syntax in the child component. The event handler and value must be specified separately. A parent component, however, can use `@bind` syntax with the child component's parameter.

The following `PasswordField` component (`PasswordField.razor`):

- Sets an `<input>` element's value to a `password` field.
- Exposes changes of a `Password` property to a parent component with an `EventCallback` that passes in the current value of the child's `password` field as its argument.
- Uses the `onclick` event to trigger the `ToggleShowPassword` method. For more information, see ASP.NET Core Blazor event handling.

```
<h1>Provide your password</h1>

Password:

<input @oninput="OnPasswordChanged"
       required
       type="@(showPassword ? "text" : "password")"
       value="@password" />

<button class="btn btn-primary" @onclick="ToggleShowPassword">
    Show password
</button>

@code {
    private bool showPassword;
    private string password;

    [Parameter]
    public string Password { get; set; }

    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private Task OnPasswordChanged(ChangeEventArgs e)
    {
        password = e.Value.ToString();

        return PasswordChanged.InvokeAsync(password);
    }

    private void ToggleShowPassword()
    {
        showPassword = !showPassword;
    }
}
```

The `PasswordField` component is used in another component:

```
@page "/Parent"

<h1>Parent Component</h1>

<PasswordField @bind-Password="password" />

@code {
    private string password;
}
```

Perform checks or trap errors in the method that invokes the binding's delegate. The following example provides immediate feedback to the user if a space is used in the password's value:

```
<h1>Child Component</h1>

Password:

<input @oninput="OnPasswordChanged"
       required
       type="@(showPassword ? "text" : "password")"
       value="@password" />

<button class="btn btn-primary" @onclick="ToggleShowPassword">
    Show password
</button>

<span class="text-danger">@validationMessage</span>

@code {
    private bool showPassword;
    private string password;
    private string validationMessage;

    [Parameter]
    public string Password { get; set; }

    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private Task OnPasswordChanged(ChangeEventArgs e)
    {
        if (password.Contains(' '))
        {
            validationMessage = "Spaces not allowed!";

            return Task.CompletedTask;
        }
        else
        {
            validationMessage = string.Empty;

            return PasswordChanged.InvokeAsync(password);
        }
    }

    private void ToggleShowPassword()
    {
        showPassword = !showPassword;
    }
}
```

Additional resources

- Binding to radio buttons in a form
- Binding `<select>` element options to C# object `null` values in a form

ASP.NET Core Blazor event handling

9/22/2020 • 6 minutes to read • Edit Online

By Luke Latham and Daniel Roth

Razor components provide event handling features. For an HTML element attribute named `@on{EVENT}` (for example, `@onclick`) with a delegate-typed value, a Razor component treats the attribute's value as an event handler.

The following code calls the `UpdateHeading` method when the button is selected in the UI:

```
<button class="btn btn-primary" @onclick="UpdateHeading">
    Update heading
</button>

@code {
    private void UpdateHeading(MouseEventArgs e)
    {
        ...
    }
}
```

The following code calls the `CheckChanged` method when the check box is changed in the UI:

```
<input type="checkbox" class="form-check-input" @onchange="CheckChanged" />

@code {
    private void CheckChanged()
    {
        ...
    }
}
```

Event handlers can also be asynchronous and return a Task. There's no need to manually call StateHasChanged. Exceptions are logged when they occur.

In the following example, `UpdateHeading` is called asynchronously when the button is selected:

```
<button class="btn btn-primary" @onclick="UpdateHeading">
    Update heading
</button>

@code {
    private async Task UpdateHeading(MouseEventArgs e)
    {
        ...
    }
}
```

Event argument types

For some events, event argument types are permitted. Specifying an event parameter in an event method definition is optional and only necessary if the event type is used in the method. In the following example, the `MouseEventArgs` event argument is used in the `ShowMessage` method to set message text:

```
private void ShowMessage(MouseEventArgs e)
{
    messageText = $"The mouse is at coordinates: {e.ScreenX}:{e.ScreenY}";
}
```

Supported EventArgs are shown in the following table.

EVENT	CLASS	DOM EVENTS AND NOTES
Clipboard	ClipboardEventArgs	`oncut`, `oncopy`, `onpaste`
Drag	DragEventArgs	`ondrag`, `ondragstart`, `ondragenter`, `ondragleave`, `ondragover`, `ondrop`, `ondragend` DataTransfer and DataTransferItem hold dragged item data.
Error	ErrorEventArgs	`onerror`
Event	EventArgs	*General* `onactivate`, `onbeforeactivate`, `onbeforedeactivate`, `ondeactivate`, `onfullscreenchange`, `onfullscreenerror`, `onloadeddata`, `onloadedmetadata`, `onpointerlockchange`, `onpointerlockerror`, `onreadystatechange`, `onscroll` *Clipboard* `onbeforecut`, `onbeforecopy`, `onbeforepaste` *Input* `oninvalid`, `onreset`, `onselect`, `onselectionchange`, `onselectstart`, `onsubmit` *Media* `oncanplay`, `oncanplaythrough`, `oncuechange`, `ondurationchange`, `onemptied`, `onended`, `onpause`, `onplay`, `onplaying`, `onratechange`, `onseeked`, `onseeking`, `onstalled`, `onstop`, `onsuspend`, `ontimeupdate`, `ontoggle`, `onvolumechange`, `onwaiting` EventHandlers holds attributes to configure the mappings between event names and event argument types.

EVENT	CLASS	DOM EVENTS AND NOTES
Focus	FocusEventArgs	`onfocus`, `onblur`, `onfocusin`, `onfocusout` Doesn't include support for `relatedTarget`.
Input	ChangeEventArgs	`onchange`, `oninput`
Keyboard	KeyboardEventArgs	`onkeydown`, `onkeypress`, `onkeyup`
Mouse	MouseEventArgs	`onclick`, `oncontextmenu`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`
Mouse pointer	PointerEventArgs	`onpointerdown`, `onpointerup`, `onpointercancel`, `onpointermove`, `onpointerover`, `onpointerout`, `onpointerenter`, `onpointerleave`, `ongotpointercapture`, `onlostpointercapture`
Mouse wheel	WheelEventArgs	`onwheel`, `onmousewheel`
Progress	ProgressEventArgs	`onabort`, `onload`, `onloadend`, `onloadstart`, `onprogress`, `ontimeout`
Touch	TouchEventArgs	`ontouchstart`, `ontouchend`, `ontouchmove`, `ontouchenter`, `ontouchleave`, `ontouchcancel` TouchPoint represents a single contact point on a touch-sensitive device.

EVENT	CLASS	DOM EVENTS AND NOTES
Clipboard	ClipboardEventArgs	`oncut`, `oncopy`, `onpaste`
Drag	DragEventArgs	`ondrag`, `ondragstart`, `ondragenter`, `ondragleave`, `ondragover`, `ondrop`, `ondragend` DataTransfer and DataTransferItem hold dragged item data.
Error	ErrorEventArgs	`onerror`

EVENT	CLASS	DOM EVENTS AND NOTES
Event	EventArgs	*General* `onactivate`, `onbeforeactivate`, `onbeforedeactivate`, `ondeactivate`, `onfullscreenchange`, `onfullscreenerror`, `onloadeddata`, `onloadedmetadata`, `onpointerlockchange`, `onpointerlockerror`, `onreadystatechange`, `onscroll` *Clipboard* `onbeforecut`, `onbeforecopy`, `onbeforepaste` *Input* `oninvalid`, `onreset`, `onselect`, `onselectionchange`, `onselectstart`, `onsubmit` *Media* `oncanplay`, `oncanplaythrough`, `oncuechange`, `ondurationchange`, `onemptied`, `onended`, `onpause`, `onplay`, `onplaying`, `onratechange`, `onseeked`, `onseeking`, `onstalled`, `onstop`, `onsuspend`, `ontimeupdate`, `onvolumechange`, `onwaiting` EventHandlers holds attributes to configure the mappings between event names and event argument types.
Focus	FocusEventArgs	`onfocus`, `onblur`, `onfocusin`, `onfocusout` Doesn't include support for `relatedTarget`.
Input	ChangeEventArgs	`onchange`, `oninput`
Keyboard	KeyboardEventArgs	`onkeydown`, `onkeypress`, `onkeyup`
Mouse	MouseEventArgs	`onclick`, `oncontextmenu`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`
Mouse pointer	PointerEventArgs	`onpointerdown`, `onpointerup`, `onpointercancel`, `onpointermove`, `onpointerover`, `onpointerout`, `onpointerenter`, `onpointerleave`, `ongotpointercapture`, `onlostpointercapture`
Mouse wheel	WheelEventArgs	`onwheel`, `onmousewheel`

EVENT	CLASS	DOM EVENTS AND NOTES
Progress	ProgressEventArgs	`onabort`, `onload`, `onloadend`, `onloadstart`, `onprogress`, `ontimeout`
Touch	TouchEventArgs	`ontouchstart`, `ontouchend`, `ontouchmove`, `ontouchenter`, `ontouchleave`, `ontouchcancel` TouchPoint represents a single contact point on a touch-sensitive device.

For more information, see the following resources:

- `EventArgs` classes in the ASP.NET Core reference source (dotnet/aspnetcore `master` branch). The `master` branch represents API under development for the *next* ASP.NET Core release. For the current release, select the appropriate GitHub repository branch (for example, `release/3.1`).
- MDN web docs: GlobalEventHandlers: Includes information on which HTML elements support each DOM event.

Lambda expressions

Lambda expressions can also be used:

```
<button @onclick="@(e => Console.WriteLine("Hello, world!"))">Say hello</button>
```

It's often convenient to close over additional values, such as when iterating over a set of elements. The following example creates three buttons, each of which calls `UpdateHeading` passing an event argument (MouseEventArgs) and its button number (`buttonNumber`) when selected in the UI:

```
<h2>@message</h2>

@for (var i = 1; i < 4; i++)
{
    var buttonNumber = i;

    <button class="btn btn-primary"
            @onclick="@(e => UpdateHeading(e, buttonNumber))">
        Button #@i
    </button>
}

@code {
    private string message = "Select a button to learn its position.";

    private void UpdateHeading(MouseEventArgs e, int buttonNumber)
    {
        message = $"You selected Button #{buttonNumber} at " +
            $"mouse position: {e.ClientX} X {e.ClientY}.";
    }
}
```

NOTE

Do **not** use a loop variable directly in a lambda expression, such as `i` in the preceding `for` loop example. Otherwise, the same variable is used by all lambda expressions, which results in use of the same value in all lambdas. Always capture the variable's value in a local variable and then use it. In the preceding example, the loop variable `i` is assigned to `buttonNumber`.

EventCallback

A common scenario with nested components is the desire to run a parent component's method when a child component event occurs. An `onclick` event occurring in the child component is a common use case. To expose events across components, use an EventCallback. A parent component can assign a callback method to a child component's EventCallback.

The `ChildComponent` in the sample app (`Components/ChildComponent.razor`) demonstrates how a button's `onclick` handler is set up to receive an EventCallback delegate from the sample's `ParentComponent`. The EventCallback is typed with `MouseEventArgs`, which is appropriate for an `onclick` event from a peripheral device:

```
<div class="panel panel-default">
    <div class="panel-heading">@Title</div>
    <div class="panel-body">@ChildContent</div>

    <button class="btn btn-primary" @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

The `ParentComponent` sets the child's EventCallback<TValue> (`OnClickCallback`) to its `ShowMessage` method.

`Pages/ParentComponent.razor`:

```
@page "/ParentComponent"

<h1>Parent-child example</h1>

<ChildComponent Title="Panel Title from Parent"
                OnClickCallback="@ShowMessage">
    Content of the child component is supplied
    by the parent component.
</ChildComponent>

<p><b>@messageText</b></p>

@code {
    private string messageText;

    private void ShowMessage(MouseEventArgs e)
    {
        messageText = $"Blaze a new trail with Blazor! ({e.ScreenX}, {e.ScreenY})";
    }
}
```

When the button is selected in the `ChildComponent`:

- The `ParentComponent`'s `ShowMessage` method is called. `messageText` is updated and displayed in the `ParentComponent`.
- A call to `StateHasChanged` isn't required in the callback's method (`ShowMessage`). StateHasChanged is called automatically to rerender the `ParentComponent`, just as child events trigger component rerendering in event handlers that execute within the child.

EventCallback and EventCallback<TValue> permit asynchronous delegates. EventCallback is weakly typed and allows passing any type argument in `InvokeAsync(Object)`. EventCallback<TValue> is strongly typed and requires passing a `T` argument in `InvokeAsync(T)` that's assignable to `TValue`.

```
<ChildComponent
    OnClickCallback="@(async () => { await Task.Yield(); messageText = "Blaze It!"; })" />
```

Invoke an EventCallback or EventCallback<TValue> with InvokeAsync and await the Task:

```
await OnClickCallback.InvokeAsync(arg);
```

Use EventCallback and EventCallback<TValue> for event handling and binding component parameters.

Prefer the strongly typed EventCallback<TValue> over EventCallback. EventCallback<TValue> provides better error feedback to users of the component. Similar to other UI event handlers, specifying the event parameter is optional. Use EventCallback when there's no value passed to the callback.

Prevent default actions

Use the `@on{EVENT}:preventDefault` directive attribute to prevent the default action for an event.

When a key is selected on an input device and the element focus is on a text box, a browser normally displays the key's character in the text box. In the following example, the default behavior is prevented by specifying the `@onkeypress:preventDefault` directive attribute. The counter increments, and the + key isn't captured into the `<input>` element's value:

```
<input value="@count" @onkeypress="KeyHandler" @onkeypress:preventDefault />

@code {
    private int count = 0;

    private void KeyHandler(KeyboardEventArgs e)
    {
        if (e.Key == "+")
        {
            count++;
        }
    }
}
```

Specifying the `@on{EVENT}:preventDefault` attribute without a value is equivalent to `@on{EVENT}:preventDefault="true"`.

The value of the attribute can also be an expression. In the following example, `shouldPreventDefault` is a `bool` field set to either `true` or `false`:

```
<input @onkeypress:preventDefault="shouldPreventDefault" />
```

Stop event propagation

Use the `@on{EVENT}:stopPropagation` directive attribute to stop event propagation.

In the following example, selecting the check box prevents click events from the second child `<div>` from propagating to the parent `<div>`:

```
<label>
    <input @bind="stopPropagation" type="checkbox" />
    Stop Propagation
</label>

<div @onclick="OnSelectParentDiv">
    <h3>Parent div</h3>

    <div @onclick="OnSelectChildDiv">
        Child div that doesn't stop propagation when selected.
    </div>

    <div @onclick="OnSelectChildDiv" @onclick:stopPropagation="stopPropagation">
        Child div that stops propagation when selected.
    </div>
</div>

@code {
    private bool stopPropagation = false;

    private void OnSelectParentDiv() =>
        Console.WriteLine($"The parent div was selected. {DateTime.Now}");
    private void OnSelectChildDiv() =>
        Console.WriteLine($"A child div was selected. {DateTime.Now}");
}
```

ASP.NET Core Blazor lifecycle

9/22/2020 • 11 minutes to read • Edit Online

By Luke Latham and Daniel Roth

The Blazor framework includes synchronous and asynchronous lifecycle methods. Override lifecycle methods to perform additional operations on components during component initialization and rendering.

Lifecycle methods

Before parameters are set

SetParametersAsync sets parameters supplied by the component's parent in the render tree:

```
public override async Task SetParametersAsync(ParameterView parameters)
{
    await ...

    await base.SetParametersAsync(parameters);
}
```

ParameterView contains the set of parameter values for the component each time SetParametersAsync is called.

The default implementation of SetParametersAsync sets the value of each property with the `[Parameter]` or `[CascadingParameter]` attribute that has a corresponding value in the ParameterView. Parameters that don't have a corresponding value in ParameterView are left unchanged.

If `base.SetParametersAync` isn't invoked, the custom code can interpret the incoming parameters value in any way required. For example, there's no requirement to assign the incoming parameters to the properties on the class.

If any event handlers are set up, unhook them on disposal. For more information, see the Component disposal with `IDisposable` section.

Component initialization methods

OnInitializedAsync and OnInitialized are invoked when the component is initialized after having received its initial parameters from its parent component in SetParametersAsync.

Use OnInitializedAsync when the component performs an asynchronous operation and should refresh when the operation is completed.

For a synchronous operation, override OnInitialized:

```
protected override void OnInitialized()
{
    ...
}
```

To perform an asynchronous operation, override OnInitializedAsync and use the `await` operator on the operation:

```
protected override async Task OnInitializedAsync()
{
    await ...
}
```

Blazor Server apps that prerender their content call OnInitializedAsync ***twice***:

- Once when the component is initially rendered statically as part of the page.
- A second time when the browser establishes a connection back to the server.

To prevent developer code in OnInitializedAsync from running twice, see the Stateful reconnection after prerendering section.

While a Blazor Server app is prerendering, certain actions, such as calling into JavaScript, aren't possible because a connection with the browser hasn't been established. Components may need to render differently when prerendered. For more information, see the Detect when the app is prerendering section.

If any event handlers are set up, unhook them on disposal. For more information, see the Component disposal with `IDisposable` section.

After parameters are set

OnParametersSetAsync or OnParametersSet are called:

- After the component is initialized in OnInitialized or OnInitializedAsync.
- When the parent component re-renders and supplies:
  - Only known primitive immutable types of which at least one parameter has changed.
  - Any complex-typed parameters. The framework can't know whether the values of a complex-typed parameter have mutated internally, so it treats the parameter set as changed.

```
protected override async Task OnParametersSetAsync()
{
    await ...
}
```

> **NOTE**
> Asynchronous work when applying parameters and property values must occur during the OnParametersSetAsync lifecycle event.

```
protected override void OnParametersSet()
{
    ...
}
```

If any event handlers are set up, unhook them on disposal. For more information, see the Component disposal with `IDisposable` section.

After component render

OnAfterRenderAsync and OnAfterRender are called after a component has finished rendering. Element and component references are populated at this point. Use this stage to perform additional initialization steps using the rendered content, such as activating third-party JavaScript libraries that operate on the rendered DOM elements.

The `firstRender` parameter for OnAfterRenderAsync and OnAfterRender:

- Is set to `true` the first time that the component instance is rendered.
- Can be used to ensure that initialization work is only performed once.

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        await ...
    }
}
```

> **NOTE**
>
> Asynchronous work immediately after rendering must occur during the OnAfterRenderAsync lifecycle event.
>
> Even if you return a Task from OnAfterRenderAsync, the framework doesn't schedule a further render cycle for your component once that task completes. This is to avoid an infinite render loop. It's different from the other lifecycle methods, which schedule a further render cycle once the returned task completes.

```
protected override void OnAfterRender(bool firstRender)
{
    if (firstRender)
    {
        ...
    }
}
```

OnAfterRender and OnAfterRenderAsync *aren't called during the prerendering process on the server*. The methods are called when the component is rendered interactively after prerendering is finished. When the app prerenders:

1. The component executes on the server to produce some static HTML markup in the HTTP response. During this phase, OnAfterRender and OnAfterRenderAsync aren't called.
2. When `blazor.server.js` or `blazor.webassembly.js` start up in the browser, the component is restarted in an interactive rendering mode. After a component is restarted, OnAfterRender and OnAfterRenderAsync **are** called because the app isn't inside the prerendering phase any longer.

If any event handlers are set up, unhook them on disposal. For more information, see the Component disposal with `IDisposable` section.

Suppress UI refreshing

Override ShouldRender to suppress UI refreshing. If the implementation returns `true`, the UI is refreshed:

```
protected override bool ShouldRender()
{
    var renderUI = true;

    return renderUI;
}
```

ShouldRender is called each time the component is rendered.

Even if ShouldRender is overridden, the component is always initially rendered.

For more information, see ASP.NET Core Blazor WebAssembly performance best practices.

State changes

StateHasChanged notifies the component that its state has changed. When applicable, calling StateHasChanged

causes the component to be rerendered.

StateHasChanged is called automatically for EventCallback methods. For more information, see ASP.NET Core Blazor event handling.

Handle incomplete async actions at render

Asynchronous actions performed in lifecycle events might not have completed before the component is rendered. Objects might be `null` or incompletely populated with data while the lifecycle method is executing. Provide rendering logic to confirm that objects are initialized. Render placeholder UI elements (for example, a loading message) while objects are `null`.

In the `FetchData` component of the Blazor templates, OnInitializedAsync is overridden to asychronously receive forecast data (`forecasts`). When `forecasts` is `null`, a loading message is displayed to the user. After the `Task` returned by OnInitializedAsync completes, the component is rerendered with the updated state.

`Pages/FetchData.razor` in the Blazor Server template:

```
@page "/fetchdata"
@using MyBlazorApp.Data
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <!-- forecast data in table element content -->
    </table>
}

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }
}
```

Handle errors

For information on handling errors during lifecycle method execution, see Handle errors in ASP.NET Core Blazor apps.

Stateful reconnection after prerendering

In a Blazor Server app when RenderMode is ServerPrerendered, the component is initially rendered statically as part of the page. Once the browser establishes a connection back to the server, the component is rendered *again*, and the component is now interactive. If the `OnInitialized{Async}` lifecycle method for initializing the component is present, the method is executed *twice*:

- When the component is prerendered statically.

- After the server connection has been established.

This can result in a noticeable change in the data displayed in the UI when the component is finally rendered.

To avoid the double-rendering scenario in a Blazor Server app:

- Pass in an identifier that can be used to cache the state during prerendering and to retrieve the state after the app restarts.
- Use the identifier during prerendering to save component state.
- Use the identifier after prerendering to retrieve the cached state.

The following code demonstrates an updated `WeatherForecastService` in a template-based Blazor Server app that avoids the double rendering:

```
public class WeatherForecastService
{
    private static readonly string[] summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild",
        "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };

    public WeatherForecastService(IMemoryCache memoryCache)
    {
        MemoryCache = memoryCache;
    }

    public IMemoryCache MemoryCache { get; }

    public Task<WeatherForecast[]> GetForecastAsync(DateTime startDate)
    {
        return MemoryCache.GetOrCreateAsync(startDate, async e =>
        {
            e.SetOptions(new MemoryCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow =
                    TimeSpan.FromSeconds(30)
            });

            var rng = new Random();

            await Task.Delay(TimeSpan.FromSeconds(10));

            return Enumerable.Range(1, 5).Select(index => new WeatherForecast
            {
                Date = startDate.AddDays(index),
                TemperatureC = rng.Next(-20, 55),
                Summary = summaries[rng.Next(summaries.Length)]
            }).ToArray();
        });
    }
}
```

For more information on the RenderMode, see ASP.NET Core Blazor hosting model configuration.

Detect when the app is prerendering

While a Blazor Server app is prerendering, certain actions, such as calling into JavaScript, aren't possible because a connection with the browser hasn't been established. Components may need to render differently when prerendered.

To delay JavaScript interop calls until after the connection with the browser is established, you can use the

OnAfterRenderAsync component lifecycle event. This event is only called after the app is fully rendered and the client connection is established.

```
@using Microsoft.JSInterop
@inject IJSRuntime JSRuntime

<div @ref="divElement">Text during render</div>

@code {
    private ElementReference divElement;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            await JSRuntime.InvokeVoidAsync(
                "setElementText", divElement, "Text after render");
        }
    }
}
```

For the preceding example code, provide a `setElementText` JavaScript function inside the `<head>` element of `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server). The function is called with JSRuntimeExtensions.InvokeVoidAsync and doesn't return a value:

```
<script>
  window.setElementText = (element, text) => element.innerText = text;
</script>
```

WARNING

The preceding example modifies the Document Object Model (DOM) directly for demonstration purposes only. Directly modifying the DOM with JavaScript isn't recommended in most scenarios because JavaScript can interfere with Blazor's change tracking.

The following component demonstrates how to use JavaScript interop as part of a component's initialization logic in a way that's compatible with prerendering. The component shows that it's possible to trigger a rendering update from inside OnAfterRenderAsync. The developer must avoid creating an infinite loop in this scenario.

Where JSRuntime.InvokeAsync is called, `ElementRef` is only used in OnAfterRenderAsync and not in any earlier lifecycle method because there's no JavaScript element until after the component is rendered.

StateHasChanged is called to rerender the component with the new state obtained from the JavaScript interop call. The code doesn't create an infinite loop because `StateHasChanged` is only called when `infoFromJs` is `null`.

```
@page "/prerendered-interop"
@using Microsoft.AspNetCore.Components
@using Microsoft.JSInterop
@inject IJSRuntime JSRuntime

<p>
    Get value via JS interop call:
    <strong id="val-get-by-interop">@(infoFromJs ?? "No value yet")</strong>
</p>

Set value via JS interop call:
<div id="val-set-by-interop" @ref="divElement"></div>

@code {
    private string infoFromJs;
    private ElementReference divElement;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender && infoFromJs == null)
        {
            infoFromJs = await JSRuntime.InvokeAsync<string>(
                "setElementText", divElement, "Hello from interop call!");

            StateHasChanged();
        }
    }
}
```

For the preceding example code, provide a `setElementText` JavaScript function inside the `<head>` element of `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server). The function is called withIJSRuntime.InvokeAsync and returns a value:

```
<script>
  window.setElementText = (element, text) => {
    element.innerText = text;
    return text;
  };
</script>
```

WARNING

The preceding example modifies the Document Object Model (DOM) directly for demonstration purposes only. Directly modifying the DOM with JavaScript isn't recommended in most scenarios because JavaScript can interfere with Blazor's change tracking.

Component disposal with IDisposable

If a component implements IDisposable, the `Dispose` method is called when the component is removed from the UI. Disposal can occur at any time, including during component initialization. The following component uses `@implements IDisposable` and the `Dispose` method:

```
@using System
@implements IDisposable

...

@code {
    public void Dispose()
    {
        ...
    }
}
```

NOTE

Calling StateHasChanged in `Dispose` isn't supported. StateHasChanged might be invoked as part of tearing down the renderer, so requesting UI updates at that point isn't supported.

Unsubscribe event handlers from .NET events. The following Blazor form examples show how to unhook an event handler in the `Dispose` method:

- Private field and lambda approach

```
@implements IDisposable

<EditForm EditContext="@editContext">

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    ...
    private EventHandler<FieldChangedEventArgs> fieldChanged;

    protected override void OnInitialized()
    {
        editContext = new EditContext(...);

        fieldChanged = (_, __) =>
        {
            ...
        };

        editContext.OnFieldChanged += fieldChanged;
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= fieldChanged;
    }
}
```

- Private method approach

```
@implements IDisposable

<EditForm EditContext="@editContext">

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    ...

    protected override void OnInitialized()
    {
        editContext = new EditContext(...);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        ...
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

Cancelable background work

Components often perform long-running background work, such as making network calls (HttpClient) and interacting with databases. It's desirable to stop the background work to conserve system resources in several situations. For example, background asynchronous operations don't automatically stop when a user navigates away from a component.

Other reasons why background work items might require cancellation include:

- An executing background task was started with faulty input data or processing parameters.
- The current set of executing background work items must be replaced with a new set of work items.
- The priority of currently executing tasks must be changed.
- The app has to be shut down in order to redeploy it to the server.
- Server resources become limited, necessitating the rescheduling of background work items.

To implement a cancelable background work pattern in a component:

- Use a CancellationTokenSource and CancellationToken.
- On disposal of the component and at any point cancellation is desired by manually cancelling the token, call `CancellationTokenSource.Cancel` to signal that the background work should be cancelled.
- After the asynchronous call returns, call ThrowIfCancellationRequested on the token.

In the following example:

- `await Task.Delay(5000, cts.Token);` represents long-running asynchronous background work.
- `BackgroundResourceMethod` represents a long-running background method that shouldn't start if the `Resource` is disposed before the method is called.

```
@implements IDisposable
@using System.Threading

<button @onclick="LongRunningWork">Trigger long running work</button>

@code {
    private Resource resource = new Resource();
    private CancellationTokenSource cts = new CancellationTokenSource();

    protected async Task LongRunningWork()
    {
        await Task.Delay(5000, cts.Token);

        cts.Token.ThrowIfCancellationRequested();
        resource.BackgroundResourceMethod();
    }

    public void Dispose()
    {
        cts.Cancel();
        cts.Dispose();
        resource.Dispose();
    }

    private class Resource : IDisposable
    {
        private bool disposed;

        public void BackgroundResourceMethod()
        {
            if (disposed)
            {
                throw new ObjectDisposedException(nameof(Resource));
            }

            ...
        }

        public void Dispose()
        {
            disposed = true;
        }
    }
}
```

ASP.NET Core Blazor component virtualization

9/22/2020 • 3 minutes to read • Edit Online

By Daniel Roth

Improve the perceived performance of component rendering using the Blazor framework's built-in virtualization support. Virtualization is a technique for limiting UI rendering to just the parts that are currently visible. For example, virtualization is helpful when the app must render a long list or a table with many rows and only a subset of items is required to be visible at any given time. Blazor provides the `Virtualize` component that can be used to add virtualization to an app's components.

Without virtualization, a typical list or table-based component might use a C# `foreach` loop to render each item in the list or each row in the table:

```
<table>
    @foreach (var employee in employees)
    {
        <tr>
            <td>@employee.FirstName</td>
            <td>@employee.LastName</td>
            <td>@employee.JobTitle</td>
        </tr>
    }
</table>
```

If the list contains thousands of items, then rendering the list may take a long time. The user may experience a noticeable UI lag.

Instead of rendering each item in the list all at one time, replace the `foreach` loop with the `Virtualize` component and specify a fixed item source with `Items`. Only the items that are currently visible are rendered:

```
<table>
    <Virtualize Context="employee" Items="@employees">
        <tr>
            <td>@employee.FirstName</td>
            <td>@employee.LastName</td>
            <td>@employee.JobTitle</td>
        </tr>
    </Virtualize>
</table>
```

If not specifying a context to the component with `Context`, use the `context` value (`@context.{PROPERTY}`) in the item content template:

```
<table>
    <Virtualize Items="@employees">
        <tr>
            <td>@context.FirstName</td>
            <td>@context.LastName</td>
            <td>@context.JobTitle</td>
        </tr>
    </Virtualize>
</table>
```

The `Virtualize` component calculates how many items to render based on the height of the container and the size of the rendered items.

Item provider delegate

If you don't want to load all of the items into memory, you can specify an items provider delegate method to the component's `ItemsProvider` parameter that asynchronously retrieves the requested items on demand:

```
<table>
    <Virtualize Context="employee" ItemsProvider="@LoadEmployees">
         <tr>
            <td>@employee.FirstName</td>
            <td>@employee.LastName</td>
            <td>@employee.JobTitle</td>
        </tr>
    </Virtualize>
</table>
```

The items provider receives an `ItemsProviderRequest`, which specifies the required number of items starting at a specific start index. The items provider then retrieves the requested items from a database or other service and returns them as an `ItemsProviderResult<TItem>` along with a count of the total items. The items provider can choose to retrieve the items with each request or cache them so that they're readily available. Don't attempt to use an items provider and assign a collection to `Items` for the same `Virtualize` component.

The following example loads employees from an `EmployeeService`:

```
private async ValueTask<ItemsProviderResult<Employee>> LoadEmployees(
    ItemsProviderRequest request)
{
    var numEmployees = Math.Min(request.Count, totalEmployees - request.StartIndex);
    var employees = await EmployeesService.GetEmployeesAsync(request.StartIndex,
        numEmployees, request.CancellationToken);

    return new ItemsProviderResult<Employee>(employees, totalEmployees);
}
```

Placeholder

Because requesting items from a remote data source might take some time, you have the option to render a placeholder (`<Placeholder>...</Placeholder>`) until the item data is available:

```
<table>
    <Virtualize Context="employee" ItemsProvider="@LoadEmployees">
        <ItemContent>
            <tr>
                <td>@employee.FirstName</td>
                <td>@employee.LastName</td>
                <td>@employee.JobTitle</td>
            </tr>
        </ItemContent>
        <Placeholder>
            <tr>
                <td>Loading...</td>
            </tr>
        </Placeholder>
    </Virtualize>
</table>
```

Item size

The size of each item in pixels can be set with `ItemSize` (default: 50px):

```
<table>
    <Virtualize Context="employee" Items="@employees" ItemSize="25">
        ...
    </Virtualize>
</table>
```

Overscan count

`OverscanCount` determines how many additional items are rendered before and after the visible region. This setting helps to reduce the frequency of rendering during scrolling. However, higher values result in more elements rendered in the page (default: 3):

```
<table>
    <Virtualize Context="employee" Items="@employees" OverscanCount="4">
        ...
    </Virtualize>
</table>
```

For example, a grid or list that renders hundreds of rows containing components is processor intensive to render. Consider virtualizing a grid or list layout so that only a subset of the components is rendered at any given time. For an example of component subset rendering, see the following components in the `Virtualization` sample app (aspnet/samples GitHub repository):

- `Virtualize` component (`Shared/Virtualize.razor`): A component written in C# that implements ComponentBase to render a set of weather data rows based on user scrolling.
- `FetchData` component (`Pages/FetchData.razor`): Uses the `Virtualize` component to display 25 rows of weather data at a time.

State changes

When making changes to items rendered by the `Virtualize` component, call StateHasChanged to force re-evaluation and rerendering of the component.

ASP.NET Core Blazor templated components

9/22/2020 • 2 minutes to read • Edit Online

By Luke Latham and Daniel Roth

Templated components are components that accept one or more UI templates as parameters, which can then be used as part of the component's rendering logic. Templated components allow you to author higher-level components that are more reusable than regular components. A couple of examples include:

- A table component that allows a user to specify templates for the table's header, rows, and footer.
- A list component that allows a user to specify a template for rendering items in a list.

Template parameters

A templated component is defined by specifying one or more component parameters of type RenderFragment or RenderFragment<TValue>. A render fragment represents a segment of UI to render. RenderFragment<TValue> takes a type parameter that can be specified when the render fragment is invoked.

`TableTemplate` component:

```
@typeparam TItem

<table class="table">
    <thead>
        <tr>@TableHeader</tr>
    </thead>
    <tbody>
        @foreach (var item in Items)
        {
            <tr>@RowTemplate(item)</tr>
        }
    </tbody>
</table>

@code {
    [Parameter]
    public RenderFragment TableHeader { get; set; }

    [Parameter]
    public RenderFragment<TItem> RowTemplate { get; set; }

    [Parameter]
    public IReadOnlyList<TItem> Items { get; set; }
}
```

When using a templated component, the template parameters can be specified using child elements that match the names of the parameters (`TableHeader` and `RowTemplate` in the following example):

```
<TableTemplate Items="pets">
    <TableHeader>
        <th>ID</th>
        <th>Name</th>
    </TableHeader>
    <RowTemplate>
        <td>@context.PetId</td>
        <td>@context.Name</td>
    </RowTemplate>
</TableTemplate>
```

> **NOTE**
>
> Generic type constraints will be supported in a future release. For more information, see Allow generic type constraints (dotnet/aspnetcore #8433).

Template context parameters

Component arguments of type RenderFragment<TValue> passed as elements have an implicit parameter named `context` (for example from the preceding code sample, `@context.PetId`), but you can change the parameter name using the `Context` attribute on the child element. In the following example, the `RowTemplate` element's `Context` attribute specifies the `pet` parameter:

```
<TableTemplate Items="pets">
    <TableHeader>
        <th>ID</th>
        <th>Name</th>
    </TableHeader>
    <RowTemplate Context="pet">
        <td>@pet.PetId</td>
        <td>@pet.Name</td>
    </RowTemplate>
</TableTemplate>
```

Alternatively, you can specify the `Context` attribute on the component element. The specified `Context` attribute applies to all specified template parameters. This can be useful when you want to specify the content parameter name for implicit child content (without any wrapping child element). In the following example, the `Context` attribute appears on the `TableTemplate` element and applies to all template parameters:

```
<TableTemplate Items="pets" Context="pet">
    <TableHeader>
        <th>ID</th>
        <th>Name</th>
    </TableHeader>
    <RowTemplate>
        <td>@pet.PetId</td>
        <td>@pet.Name</td>
    </RowTemplate>
</TableTemplate>
```

Generic-typed components

Templated components are often generically typed. For example, a generic `ListViewTemplate` component can be used to render `IEnumerable<T>` values. To define a generic component, use the `@typeparam` directive to specify type parameters:

```
@typeparam TItem

<ul>
    @foreach (var item in Items)
    {
        @ItemTemplate(item)
    }
</ul>

@code {
    [Parameter]
    public RenderFragment<TItem> ItemTemplate { get; set; }

    [Parameter]
    public IReadOnlyList<TItem> Items { get; set; }
}
```

When using generic-typed components, the type parameter is inferred if possible:

```
<ListViewTemplate Items="pets">
    <ItemTemplate Context="pet">
        <li>@pet.Name</li>
    </ItemTemplate>
</ListViewTemplate>
```

Otherwise, the type parameter must be explicitly specified using an attribute that matches the name of the type parameter. In the following example, `TItem="Pet"` specifies the type:

```
<ListViewTemplate Items="pets" TItem="Pet">
    <ItemTemplate Context="pet">
        <li>@pet.Name</li>
    </ItemTemplate>
</ListViewTemplate>
```

Integrate ASP.NET Core Razor components into Razor Pages and MVC apps

9/22/2020 • 5 minutes to read • Edit Online

By Luke Latham and Daniel Roth

Razor components can be integrated into Razor Pages and MVC apps. When the page or view is rendered, components can be prerendered at the same time.

After preparing the app, use the guidance in the following sections depending on the app's requirements:

- Routable components: For components that are directly routable from user requests. Follow this guidance when visitors should be able to make an HTTP request in their browser for a component with an `@page` directive.
  - Use routable components in a Razor Pages app
  - Use routable components in an MVC app
- Render components from a page or view: For components that aren't directly routable from user requests. Follow this guidance when the app embeds components into existing pages and views with the Component Tag Helper.

Prepare the app

An existing Razor Pages or MVC app can integrate Razor components into pages and views:

1. In the app's layout file (`_Layout.cshtml`):
   - Add the following `<base>` tag to the `<head>` element:

     ```
     <base href="~/" />
     ```

     The `href` value (the *app base path*) in the preceding example assumes that the app resides at the root URL path (`/`). If the app is a sub-application, follow the guidance in the *App base path* section of the Host and deploy ASP.NET Core Blazor article.

     The `_Layout.cshtml` file is located in the *Pages/Shared* folder in a Razor Pages app or *Views/Shared* folder in an MVC app.

   - Add a `<script>` tag for the *blazor.server.js* script immediately before the closing `</body>` tag:

     ```
     <script src="_framework/blazor.server.js"></script>
     ```

     The framework adds the *blazor.server.js* script to the app. There's no need to manually add the script to the app.
2. Add an `_Imports.razor` file to the root folder of the project with the following content (change the last namespace, `MyAppNamespace`, to the namespace of the app):

```
@using System.Net.Http
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using MyAppNamespace
```

3. In `Startup.ConfigureServices`, register the Blazor Server service:

```
services.AddServerSideBlazor();
```

4. In `Startup.Configure`, add the Blazor Hub endpoint to `app.UseEndpoints`:

```
endpoints.MapBlazorHub();
```

5. Integrate components into any page or view. For more information, see the Render components from a page or view section.

Use routable components in a Razor Pages app

This section pertains to adding components that are directly routable from user requests.

To support routable Razor components in Razor Pages apps:

1. Follow the guidance in the Prepare the app section.
2. Add an `App.razor` file to the project root with the following content:

```
@using Microsoft.AspNetCore.Components.Routing

<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" />
    </Found>
    <NotFound>
        <h1>Page not found</h1>
        <p>Sorry, but there's nothing here!</p>
    </NotFound>
</Router>
```

3. Add a `_Host.cshtml` file to the `Pages` folder with the following content:

```
@page "/blazor"
@{
    Layout = "_Layout";
}

<app>
    <component type="typeof(App)" render-mode="ServerPrerendered" />
</app>
```

Components use the shared `_Layout.cshtml` file for their layout.

RenderMode configures whether the `App` component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

RENDER MODE	DESCRIPTION
ServerPrerendered	Renders the `App` component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a Blazor Server app. Output from the `App` component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the `App` component into static HTML.

For more information on the Component Tag Helper, see Component Tag Helper in ASP.NET Core.

4. Add a low-priority route for the `_Host.cshtml` page to endpoint configuration in `Startup.Configure`:

```
app.UseEndpoints(endpoints =>
{
    ...

    endpoints.MapFallbackToPage("/_Host");
});
```

5. Add routable components to the app. For example:

```
@page "/counter"

<h1>Counter</h1>

...
```

For more information on namespaces, see the Component namespaces section.

Use routable components in an MVC app

This section pertains to adding components that are directly routable from user requests.

To support routable Razor components in MVC apps:

1. Follow the guidance in the Prepare the app section.
2. Add an `App.razor` file to the root of the project with the following content:

```
@using Microsoft.AspNetCore.Components.Routing

<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" />
    </Found>
    <NotFound>
        <h1>Page not found</h1>
        <p>Sorry, but there's nothing here!</p>
    </NotFound>
</Router>
```

3. Add a `_Host.cshtml` file to the `Views/Home` folder with the following content:

```
@{
    Layout = "_Layout";
}

<app>
    <component type="typeof(App)" render-mode="ServerPrerendered" />
</app>
```

Components use the shared `_Layout.cshtml` file for their layout.

RenderMode configures whether the `App` component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

RENDER MODE	DESCRIPTION
ServerPrerendered	Renders the `App` component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a Blazor Server app. Output from the `App` component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the `App` component into static HTML.

For more information on the Component Tag Helper, see Component Tag Helper in ASP.NET Core.

4. Add an action to the Home controller:

```
public IActionResult Blazor()
{
   return View("_Host");
}
```

5. Add a low-priority route for the controller action that returns the `_Host.cshtml` view to the endpoint configuration in `Startup.Configure`:

```
app.UseEndpoints(endpoints =>
{
    ...

    endpoints.MapFallbackToController("Blazor", "Home");
});
```

6. Create a `Pages` folder and add routable components to the app. For example:

```
@page "/counter"

<h1>Counter</h1>

...
```

For more information on namespaces, see the Component namespaces section.

Render components from a page or view

This section pertains to adding components to pages or views, where the components aren't directly routable from user requests.

To render a component from a page or view, use the Component Tag Helper.

Render stateful interactive components

Stateful interactive components can be added to a Razor page or view.

When the page or view renders:

- The component is prerendered with the page or view.
- The initial component state used for prerendering is lost.
- New component state is created when the SignalR connection is established.

The following Razor page renders a `Counter` component:

```
<h1>My Razor Page</h1>

<component type="typeof(Counter)" render-mode="ServerPrerendered"
    param-InitialValue="InitialValue" />

@functions {
    [BindProperty(SupportsGet=true)]
    public int InitialValue { get; set; }
}
```

For more information, see Component Tag Helper in ASP.NET Core.

Render noninteractive components

In the following Razor page, the `Counter` component is statically rendered with an initial value that's specified using a form. Since the component is statically rendered, the component isn't interactive:

```
<h1>My Razor Page</h1>

<form>
    <input type="number" asp-for="InitialValue" />
    <button type="submit">Set initial value</button>
</form>

<component type="typeof(Counter)" render-mode="Static"
    param-InitialValue="InitialValue" />

@functions {
    [BindProperty(SupportsGet=true)]
    public int InitialValue { get; set; }
}
```

For more information, see Component Tag Helper in ASP.NET Core.

Component namespaces

When using a custom folder to hold the app's components, add the namespace representing the folder to either the page/view or to the `_ViewImports.cshtml` file. In the following example:

- Change `MyAppNamespace` to the app's namespace.
- If a folder named *Components* isn't used to hold the components, change `Components` to the folder where the components reside.

```
@using MyAppNamespace.Components
```

The `_ViewImports.cshtml` file is located in the `Pages` folder of a Razor Pages app or the `Views` folder of an MVC app.

For more information, see Create and use ASP.NET Core Razor components.

ASP.NET Core Razor components class libraries

9/22/2020 • 5 minutes to read • Edit Online

By Simon Timms

Components can be shared in a Razor class library (RCL) across projects. A *Razor components class library* can be included from:

- Another project in the solution.
- A NuGet package.
- A referenced .NET library.

Just as components are regular .NET types, components provided by an RCL are normal .NET assemblies.

Create an RCL

- Visual Studio
- .NET Core CLI

1. Create a new project.
2. Select **Razor Class Library**. Select **Next**.
3. In the **Create a new Razor class library** dialog, select **Create**.
4. Provide a project name in the **Project name** field or accept the default project name. The examples in this topic use the project name `ComponentLibrary`. Select **Create**.
5. Add the RCL to a solution:
   a. Right-click the solution. Select **Add** > **Existing Project**.
   b. Navigate to the RCL's project file.
   c. Select the RCL's project file (`.csproj`).
6. Add a reference the RCL from the app:
   a. Right-click the app project. Select **Add** > **Reference**.
   b. Select the RCL project. Select **OK**.

> **NOTE**
> If the **Support pages and views** check box is selected when generating the RCL from the template, then also add an `_Imports.razor` file to root of the generated project with the following contents to enable Razor component authoring:
>
> ```
> @using Microsoft.AspNetCore.Components.Web
> ```
>
> Manually add the file the root of the generated project.

Consume a library component

In order to consume components defined in a library in another project, use either of the following approaches:

- Use the full type name with the namespace.
- Use Razor's `@using` directive. Individual components can be added by name.

In the following examples, `ComponentLibrary` is a component library containing the `Component1` component (

`Component1.razor`). The `Component1` component is an example component automatically added by the RCL project template when the library is created.

Reference the `Component1` component using its namespace:

```
<h1>Hello, world!</h1>

Welcome to your new app.

<ComponentLibrary.Component1 />
```

Alternatively, bring the library into scope with an `@using` directive and use the component without its namespace:

```
@using ComponentLibrary

<h1>Hello, world!</h1>

Welcome to your new app.

<Component1 />
```

Optionally, include the `@using ComponentLibrary` directive in the top-level `_Import.razor` file to make the library's components available to an entire project. Add the directive to an `_Import.razor` file at any level to apply the namespace to a single component or set of components within a folder.

To provide `Component1`'s `my-component` CSS class to the component, link to the library's stylesheet using the framework's `Link` component in `Component1.razor`:

```
<div class="my-component">
    <Link href="_content/ComponentLibrary/styles.css" rel="stylesheet" />

    <p>
        This Blazor component is defined in the <strong>ComponentLibrary</strong> package.
    </p>
</div>
```

To provide the stylesheet across the app, you can alternatively link to the library's stylesheet in the app's `wwwroot/index.html` file (Blazor WebAssembly) or `Pages/_Host.cshtml` file (Blazor Server):

```
<head>
    ...
    <link href="_content/ComponentLibrary/styles.css" rel="stylesheet" />
</head>
```

When the `Link` component is used in a child component, the linked asset is also available to any other child component of the parent component as long as the child with the `Link` component is rendered. The distinction between using the `Link` component in a child component and placing a `<link>` HTML tag in `wwwroot/index.html` or `Pages/_Host.cshtml` is that a framework component's rendered HTML tag:

- Can be modified by application state. A hard-coded `<link>` HTML tag can't be modified by application state.
- Is removed from the HTML `<head>` when the parent component is no longer rendered.

To provide `Component1`'s `my-component` CSS class, link to the library's stylesheet in the app's `wwwroot/index.html` file (Blazor WebAssembly) or `Pages/_Host.cshtml` file (Blazor Server):

```
<head>
    ...
    <link href="_content/ComponentLibrary/styles.css" rel="stylesheet" />
</head>
```

Create a Razor components class library with static assets

An RCL can include static assets. The static assets are available to any app that consumes the library. For more information, see Reusable Razor UI in class libraries with ASP.NET Core.

Supply components and static assets to multiple hosted Blazor apps

For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

Browser compatibility analyzer for Blazor WebAssembly

Blazor WebAssembly apps target the full .NET API surface area, but not all .NET APIs are supported on WebAssembly due to browser sandbox constraints. Unsupported APIs throw PlatformNotSupportedException when running on WebAssembly. A platform compatibility analyzer warns the developer when the app uses APIs that aren't supported by the app's target platforms. For Blazor WebAssembly apps, this means checking that APIs are supported in browsers. Annotating .NET framework APIs for the compatibility analyzer is an on-going process, so not all .NET framework API is currently annotated.

Blazor WebAssembly and Razor class library projects *automatically* enable browser compatibilty checks by adding `browser` as a supported platform with the `SupportedPlatform` MSBuild item. Library developers can manually add the `SupportedPlatform` item to a library's project file to enable the feature:

```
<ItemGroup>
  <SupportedPlatform Include="browser" />
</ItemGroup>
```

When authoring a library, indicate that a particular API isn't supported in browsers by specifying `browser` to UnsupportedOSPlatformAttribute:

```
[UnsupportedOSPlatform("browser")]
private static string GetLoggingDirectory()
{
    ...
}
```

For more information, see Annotating APIs as unsupported on specific platforms (dotnet/designs GitHub repository.

Blazor JavaScript isolation and object references

Blazor enables JavaScript isolation in standard JavaScript modules. JavaScript isolation provides the following benefits:

- Imported JavaScript no longer pollutes the global namespace.
- Consumers of the library and components aren't required to manually import the related JavaScript.

For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Build, pack, and ship to NuGet

Because component libraries are standard .NET libraries, packaging and shipping them to NuGet is no different from packaging and shipping any library to NuGet. Packaging is performed using the `dotnet pack` command in a command shell:

```
dotnet pack
```

Upload the package to NuGet using the `dotnet nuget push` command in a command shell.

Additional resources

- Reusable Razor UI in class libraries with ASP.NET Core
- Add an XML Intermediate Language (IL) Trimmer configuration file to a library

- Reusable Razor UI in class libraries with ASP.NET Core
- Add an XML Intermediate Language (IL) Linker configuration file to a library

ASP.NET Core Blazor globalization and localization

9/22/2020 • 4 minutes to read • Edit Online

By Luke Latham and Daniel Roth

Razor components can be made accessible to users in multiple cultures and languages. The following .NET globalization and localization scenarios are available:

- .NET's resources system
- Culture-specific number and date formatting

A limited set of ASP.NET Core's localization scenarios are currently supported:

- IStringLocalizer and IStringLocalizer<T> *are supported* in Blazor apps.
- IHtmlLocalizer, IViewLocalizer, and Data Annotations localization are ASP.NET Core MVC scenarios and **not supported** in Blazor apps.

For more information, see Globalization and localization in ASP.NET Core.

Globalization

Blazor's `@bind` functionality performs formats and parses values for display based on the user's current culture.

The current culture can be accessed from the System.Globalization.CultureInfo.CurrentCulture property.

CultureInfo.InvariantCulture is used for the following field types (`<input type="{TYPE}" />`):

- `date`
- `number`

The preceding field types:

- Are displayed using their appropriate browser-based formatting rules.
- Can't contain free-form text.
- Provide user interaction characteristics based on the browser's implementation.

The following field types have specific formatting requirements and aren't currently supported by Blazor because they aren't supported by all major browsers:

- `datetime-local`
- `month`
- `week`

`@bind` supports the `@bind:culture` parameter to provide a System.Globalization.CultureInfo for parsing and formatting a value. Specifying a culture isn't recommended when using the `date` and `number` field types. `date` and `number` have built-in Blazor support that provides the required culture.

Localization

Blazor WebAssembly

Blazor WebAssembly apps set the culture using the user's language preference.

To explicitly configure the culture, set CultureInfo.DefaultThreadCurrentCulture and

CultureInfo.DefaultThreadCurrentUICulture in `Program.Main`.

By default, Blazor WebAssembly carries globalization resources required to display values, such as dates and currency, in the user's culture. If the app doesn't require localization, you may configure the app to support the invariant culture, which is based on the `en-US` culture:

```
<PropertyGroup>
  <InvariantGlobalization>true</InvariantGlobalization>
</PropertyGroup>
```

By default, the Intermediate Language (IL) Linker configuration for Blazor WebAssembly apps strips out internationalization information except for locales explicitly requested. For more information, see Configure the Linker for ASP.NET Core Blazor.

While the culture that Blazor selects by default might be sufficient for most users, consider offering a way for users to specify their preferred locale. For a Blazor WebAssembly sample app with a culture picker, see the `LocSample` localization sample app.

Blazor Server

Blazor Server apps are localized using Localization Middleware. The middleware selects the appropriate culture for users requesting resources from the app.

The culture can be set using one of the following approaches:

- Cookies
- Provide UI to choose the culture

For more information and examples, see Globalization and localization in ASP.NET Core.

Cookies

A localization culture cookie can persist the user's culture. The Localization Middleware reads the cookie on subsequent requests to set the user's culture.

Use of a cookie ensures that the WebSocket connection can correctly propagate the culture. If localization schemes are based on the URL path or query string, the scheme might not be able to work with WebSockets, thus fail to persist the culture. Therefore, use of a localization culture cookie is the recommended approach.

Any technique can be used to assign a culture if the culture is persisted in a localization cookie. If the app already has an established localization scheme for server-side ASP.NET Core, continue to use the app's existing localization infrastructure and set the localization culture cookie within the app's scheme.

The following example shows how to set the current culture in a cookie that can be read by the Localization Middleware. Create a Razor expression in the `Pages/_Host.cshtml` file immediately inside the opening `<body>` tag:

```
@using System.Globalization
@using Microsoft.AspNetCore.Localization

...

<body>
    @{
        this.HttpContext.Response.Cookies.Append(
            CookieRequestCultureProvider.DefaultCookieName,
            CookieRequestCultureProvider.MakeCookieValue(
                new RequestCulture(
                    CultureInfo.CurrentCulture,
                    CultureInfo.CurrentUICulture)));
    }

    ...
</body>
```

Localization is handled by the app in the following sequence of events:

1. The browser sends an initial HTTP request to the app.
2. The culture is assigned by the Localization Middleware.
3. The Razor expression in the `_Host` page (`_Host.cshtml`) persists the culture in a cookie as part of the response.
4. The browser opens a WebSocket connection to create an interactive Blazor Server session.
5. The Localization Middleware reads the cookie and assigns the culture.
6. The Blazor Server session begins with the correct culture.

Provide UI to choose the culture

To provide UI to allow a user to select a culture, a *redirect-based approach* is recommended. The process is similar to what happens in a web app when a user attempts to access a secure resource. The user is redirected to a sign-in page and then redirected back to the original resource.

The app persists the user's selected culture via a redirect to a controller. The controller sets the user's selected culture into a cookie and redirects the user back to the original URI.

Establish an HTTP endpoint on the server to set the user's selected culture in a cookie and perform the redirect back to the original URI:

```
[Route("[controller]/[action]")]
public class CultureController : Controller
{
    public IActionResult SetCulture(string culture, string redirectUri)
    {
        if (culture != null)
        {
            HttpContext.Response.Cookies.Append(
                CookieRequestCultureProvider.DefaultCookieName,
                CookieRequestCultureProvider.MakeCookieValue(
                    new RequestCulture(culture)));
        }

        return LocalRedirect(redirectUri);
    }
}
```

> **WARNING**
>
> Use the LocalRedirect action result to prevent open redirect attacks. For more information, see Prevent open redirect attacks in ASP.NET Core.

If the app isn't configured to process controller actions:

- Add MVC services to the service collection in `Startup.ConfigureServices`:

```
services.AddControllers();
```

- Add controller endpoint routing in `Startup.Configure`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/_Host");
});
```

The following component shows an example of how to perform the initial redirection when the user selects a culture:

```
@inject NavigationManager NavigationManager

<h3>Select your language</h3>

<select @onchange="OnSelected">
    <option>Select...</option>
    <option value="en-US">English</option>
    <option value="fr-FR">Français</option>
</select>

@code {
    private void OnSelected(ChangeEventArgs e)
    {
        var culture = (string)e.Value;
        var uri = new Uri(NavigationManager.Uri)
            .GetComponents(UriComponents.PathAndQuery, UriFormat.Unescaped);
        var query = $"?culture={Uri.EscapeDataString(culture)}&" +
            $"redirectUri={Uri.EscapeDataString(uri)}";

        NavigationManager.NavigateTo("/Culture/SetCulture" + query, forceLoad: true);
    }
}
```

Additional resources

- Globalization and localization in ASP.NET Core

ASP.NET Core Blazor layouts

9/22/2020 • 4 minutes to read • Edit Online

By Rainer Stropek and Luke Latham

Some app elements, such as menus, copyright messages, and company logos, are usually part of app's overall layout and used by every component in the app. Copying the code of these elements into all of the components of an app isn't an efficient approach. Every time one of the elements requires an update, every component must be updated. Such duplication is difficult to maintain and can lead to inconsistent content over time. *Layouts* solve this problem.

Technically, a layout is just another component. A layout is defined in a Razor template or in C# code and can use data binding, dependency injection, and other component scenarios.

To turn a *component* into a *layout*, the component:

- Inherits from LayoutComponentBase, which defines a Body property for the rendered content inside the layout.
- Uses the Razor syntax `@Body` to specify the location in the layout markup where the content is rendered.

The following code sample shows the Razor template of a layout component, `MainLayout.razor`. The layout inherits LayoutComponentBase and sets the `@Body` between the navigation bar and the footer:

```
@inherits LayoutComponentBase

<header>
    <h1>Doctor Who&trade; Episode Database</h1>
</header>

<nav>
    <a href="masterlist">Master Episode List</a>
    <a href="search">Search</a>
    <a href="new">Add Episode</a>
</nav>

@Body

<footer>
    @TrademarkMessage
</footer>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/";
}
```

`MainLayout` component

In an app based on one of the Blazor project templates, the `MainLayout` component (`MainLayout.razor`) is in the app's `Shared` folder:

```
@inherits LayoutComponentBase

<div class="sidebar">
    <NavMenu />
</div>

<div class="main">
    <div class="content px-4">
        @Body
    </div>
</div>
```

Default layout

Specify the default app layout in the Router component in the app's `App.razor` file. The following Router component, which is provided by the default Blazor templates, sets the default layout to the `MainLayout` component:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

To supply a default layout for NotFound content, specify a LayoutView for NotFound content:

```
<Router AppAssembly="@typeof(Startup).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <h1>Page not found</h1>
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

For more information on the Router component, see ASP.NET Core Blazor routing.

Specifying the layout as a default layout in the router is a useful practice because it can be overridden on a per-component or per-folder basis. Prefer using the router to set the app's default layout because it's the most general technique.

Specify a layout in a component

Use the Razor directive `@layout` to apply a layout to a component. The compiler converts `@layout` into a LayoutAttribute, which is applied to the component class.

The content of the following `MasterList` component is inserted into the `MasterLayout` at the position of `@Body`:

```
@layout MasterLayout
@page "/masterlist"

<h1>Master Episode List</h1>
```

Specifying the layout directly in a component overrides a *default layout* set in the router or an `@layout` directive imported from `_Imports.razor`.

Centralized layout selection

Every folder of an app can optionally contain a template file named `_Imports.razor`. The compiler includes the directives specified in the imports file in all of the Razor templates in the same folder and recursively in all of its subfolders. Therefore, an `_Imports.razor` file containing `@layout MyCoolLayout` ensures that all of the components in a folder use `MyCoolLayout`. There's no need to repeatedly add `@layout MyCoolLayout` to all of the `.razor` files within the folder and subfolders. `@using` directives are also applied to components in the same way.

The following `_Imports.razor` file imports:

- `MyCoolLayout`.
- All Razor components in the same folder and any subfolders.
- The `BlazorApp1.Data` namespace.

```
@layout MyCoolLayout
@using Microsoft.AspNetCore.Components
@using BlazorApp1.Data
```

The `_Imports.razor` file is similar to the _ViewImports.cshtml file for Razor views and pages but applied specifically to Razor component files.

Specifying a layout in `_Imports.razor` overrides a layout specified as the router's *default layout*.

WARNING

Do **not** add a Razor `@layout` directive to the root `_Imports.razor` file, which results in an infinite loop of layouts in the app. To control the default app layout, specify the layout in the `Router` component. For more information, see the Default layout section.

Nested layouts

Apps can consist of nested layouts. A component can reference a layout which in turn references another layout. For example, nesting layouts are used to create a multi-level menu structure.

The following example shows how to use nested layouts. The `EpisodesComponent.razor` file is the component to display. The component references the `MasterListLayout`:

```
@layout MasterListLayout
@page "/masterlist/episodes"

<h1>Episodes</h1>
```

The `MasterListLayout.razor` file provides the `MasterListLayout`. The layout references another layout, `MasterLayout`, where it's rendered. `EpisodesComponent` is rendered where `@Body` appears:

```
@layout MasterLayout
@inherits LayoutComponentBase

<nav>
    <!-- Menu structure of master list -->
    ...
</nav>

@Body
```

Finally, `MasterLayout` in `MasterLayout.razor` contains the top-level layout elements, such as the header, main menu, and footer. `MasterListLayout` with the `EpisodesComponent` is rendered where `@Body` appears:

```
@inherits LayoutComponentBase

<header>...</header>
<nav>...</nav>

@Body

<footer>
    @TrademarkMessage
</footer>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/";
}
```

Share a Razor Pages layout with integrated components

When routable components are integrated into a Razor Pages app, the app's shared layout can be used with the components. For more information, see Integrate ASP.NET Core Razor components into Razor Pages and MVC apps.

Additional resources

- Layout in ASP.NET Core

ASP.NET Core Blazor forms and validation

9/22/2020 • 22 minutes to read • Edit Online

By Daniel Roth, Rémi Bourgarel, and Luke Latham

Forms and validation are supported in Blazor using data annotations.

The following `ExampleModel` type defines validation logic using data annotations:

```
using System.ComponentModel.DataAnnotations;

public class ExampleModel
{
    [Required]
    [StringLength(10, ErrorMessage = "Name is too long.")]
    public string Name { get; set; }
}
```

A form is defined using the EditForm component. The following form demonstrates typical elements, components, and Razor code:

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }
}
```

In the preceding example:

- The form validates user input in the `name` field using the validation defined in the `ExampleModel` type. The model is created in the component's `@code` block and held in a private field (`exampleModel`). The field is assigned to the `Model` attribute of the `<EditForm>` element.
- The InputText component's `@bind-Value` binds:
  - The model property (`exampleModel.Name`) to the InputText component's `Value` property. For more information on property binding, see ASP.NET Core Blazor data binding.
  - A change event delegate to the InputText component's `ValueChanged` property.
- The DataAnnotationsValidator validator component attaches validation support using data annotations.
- The ValidationSummary component summarizes validation messages.
- `HandleValidSubmit` is triggered when the form successfully submits (passes validation).

Built-in forms components

A set of built-in components are available to receive and validate user input. Inputs are validated when they're changed and when a form is submitted. Available input components are shown in the following table.

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
`InputFile`	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
`InputRadio`	`<input type="radio">`
`InputRadioGroup`	`<input type="radio">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

INPUT COMPONENT	RENDERED AS...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputNumber<TValue>	`<input type="number">`
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

> **NOTE**
>
> The `InputRadio` and `InputRadioGroup` components are available in ASP.NET Core 5.0 or later. For more information, select a 5.0 or later version of this article.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed, including updating the field CSS class to reflect the field state. Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?`).

The following `Starship` type defines validation logic using a larger set of properties and data annotations than the

earlier `ExampleModel`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character limit).")]
    public string Identifier { get; set; }

    public string Description { get; set; }

    [Required]
    public string Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true",
        ErrorMessage = "This form disallows unapproved ships.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

In the preceding example, `Description` is optional because no data annotations are present.

The following form validates user input using the validation defined in the `Starship` model:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
```

```
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" />
        </label>
    </p>

    <button type="submit">Submit</button>

    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        ...
    }
}
```

The EditForm creates an EditContext as a cascading value that tracks metadata about the edit process, including which fields have been modified and the current validation messages.

Assign **either** an EditContext **or** an EditForm.Model to an EditForm. Assignment of both isn't supported and generates a **runtime error**.

The EditForm provides convenient events for valid and invalid form submission:

- OnValidSubmit
- OnInvalidSubmit

Use OnSubmit to use custom code to trigger validation and check field values.

In the following example:

- The `HandleSubmit` method executes when the `Submit` button is selected.
- The form is validated by calling EditContext.Validate.
- Additional code is executed depending on the validation result. Place business logic in the method assigned to OnSubmit.

```
<EditForm EditContext="@editContext" OnSubmit="@HandleSubmit">

    ...

    <button type="submit">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
    }

    private async Task HandleSubmit()
    {
        var isValid = editContext.Validate();

        if (isValid)
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

NOTE

Framework API doesn't exist to clear validation messages directly from an EditContext. Therefore, we don't generally recommend adding validation messages to a new ValidationMessageStore in a form. To manage validation messages, use a validator component with your business logic validation code, as described in this article.

Display name support

This section applies to ASP.NET Core in .NET 5 Release Candidate 1 (RC1) or later.

The following built-in components support display names with the `DisplayName` parameter:

- InputDate<TValue>
- InputNumber<TValue>
- InputSelect<TValue>

In the following `InputDate` component example:

- The display name (`DisplayName`) is set to `birthday`.
- The component is bound to the `BirthDate` property as a `DateTime` type.

```
<InputDate @bind-Value="@BirthDate" DisplayName="birthday" />

@code {
    public DateTime BirthDate { get; set; }
}
```

If the user doesn't provide a date value, the validation error appears as:

```
The birthday must be a date.
```

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). Create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing, for example client-side validation followed by server-side validation. The validator component example shown in this section, `CustomValidator`, is used in the following sections of this article:

- Business logic validation
- Server validation

> **NOTE**
>
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, any custom attributes applied to the model must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a `Dictionary<string, List<string>>`. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:
  - Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
  - A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
  - The `ClearErrors` method is called by developer code. All of the errors are cleared.

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample.Client
{
    public class CustomValidator : ComponentBase
    {
        private ValidationMessageStore messageStore;

        [CascadingParameter]
        private EditContext CurrentEditContext { get; set; }

        protected override void OnInitialized()
        {
            if (CurrentEditContext == null)
            {
                throw new InvalidOperationException(
                    $"{nameof(CustomValidator)} requires a cascading " +
                    $"parameter of type {nameof(EditContext)}. " +
                    $"For example, you can use {nameof(CustomValidator)} " +
                    $"inside an {nameof(EditForm)}.");
            }

            messageStore = new ValidationMessageStore(CurrentEditContext);

            CurrentEditContext.OnValidationRequested += (s, e) =>
                messageStore.Clear();
            CurrentEditContext.OnFieldChanged += (s, e) =>
                messageStore.Clear(e.FieldIdentifier);
        }

        public void DisplayErrors(Dictionary<string, List<string>> errors)
        {
            foreach (var err in errors)
            {
                messageStore.Add(CurrentEditContext.Field(err.Key), err.Value);
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }

        public void ClearErrors()
        {
            messageStore.Clear();
            CurrentEditContext.NotifyValidationStateChanged();
        }
    }
}
```

Business logic validation

Business logic validation can be accomplished with a validator component that receives form errors in a dictionary.

In the following example:

- The `CustomValidator` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm:

```
@page "/FormsValidation"

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    ...

</EditForm>

@code {
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private void HandleValidSubmit()
    {
        customValidator.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (starship.Classification == "Defense" &&
                string.IsNullOrEmpty(starship.Description))
        {
            errors.Add(nameof(starship.Description),
                new List<string>() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Count() > 0)
        {
            customValidator.DisplayErrors(errors);
        }
        else
        {
            // Process the form
        }
    }
}
```

NOTE

As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

Server validation

Server validation can be accomplished with a server validator component:

- Process client-side validation in the form with the DataAnnotationsValidator component.
- When the form passes client-side validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status

code (*200 - OK*). If validation fails, return a failure status code (*400 - Bad Request*) and the field validation errors.

- Either disable the form on success or display the errors.

The following example is based on:

- A hosted Blazor solution created by the Blazor Hosted project template. The example can be used with any of the secure hosted Blazor solutions described in the Security and Identity documentation.
- The *Starfleet Starship Database* form example in the preceding Built-in forms components section.
- The Blazor framework's DataAnnotationsValidator component.
- The `CustomValidator` component shown in the Validator components section.

In the following example, the server API validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

Place the `Starship` model into the solution's `Shared` project so that both the client and server apps can use the model. Since the model requires data annotations, add a package reference for `System.ComponentModel.Annotations` to the `Shared` project's project file:

```
<ItemGroup>
  <PackageReference Include="System.ComponentModel.Annotations" Version="{VERSION}" />
</ItemGroup>
```

To determine the latest non-preview version of the package, see the package **Version History** at NuGet.org.

In the server API project, add a controller to process starship validation requests (`Controllers/StarshipValidation.cs`) and return failed validation messages:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

namespace BlazorSample.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class StarshipValidationController : ControllerBase
    {
        private readonly ILogger<StarshipValidationController> logger;

        public StarshipValidationController(
            ILogger<StarshipValidationController> logger)
        {
            this.logger = logger;
        }

        [HttpPost]
        public async Task<IActionResult> Post(Starship starship)
        {
            try
            {
                if (starship.Classification == "Defense" &&
                    string.IsNullOrEmpty(starship.Description))
                {
                    ModelState.AddModelError(nameof(starship.Description),
                        "For a 'Defense' ship " +
                        "classification, 'Description' is required.");
                }
                else
                {
                    // Process the form asynchronously
                    // async ...

                    return Ok(ModelState);
                }
            }
            catch (Exception ex)
            {
                logger.LogError("Validation Error: {MESSAGE}", ex.Message);
            }

            return BadRequest(ModelState);
        }
    }
}
```

When a model binding validation error occurs on the server, an `ApiController` (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation errors, as shown in the following example when all of the fields of the *Starfleet Starship Database* form aren't submitted and the form fails validation:

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Identifier": ["The Identifier field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

If the server API returns the preceding default JSON response, it's possible for the client to parse the response to obtain the children of the `errors` node. However, it's inconvenient to parse the file. Parsing the JSON requires additional code after calling ReadFromJsonAsync in order to produce a `Dictionary<string, List<string>>` of errors for forms validation error processing. Ideally, the server API should only return the validation errors:

```
{
  "Identifier": ["The Identifier field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in `Startup.ConfigureServices`. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails:

```
using Microsoft.AspNetCore.Mvc;

...

services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

For more information, see Handle errors in ASP.NET Core web APIs.

In the client project, add the validator component shown in the Validator components section.

In the client project, the *Starfleet Starship Database* form is updated to show server validation errors with help of the `CustomValidator` component. When the server API returns validation messages, they're added to the `CustomValidator` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's ValidationSummary:

```
@page "/FormValidation"
```

```
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<FormValidation> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="@starship" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <CustomValidator @ref="customValidator" />
    <ValidationSummary />

    <p>
        <label>
            Identifier:
            <InputText @bind-Value="starship.Identifier" disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="starship.Description"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="starship.Classification" disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </p>
    <p>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="starship.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="starship.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </p>
    <p>
        <label>
            Production Date:
            <InputDate @bind-Value="starship.ProductionDate" disabled="@disabled" />
        </label>
    </p>

    <button type="submit" disabled="@disabled">Submit</button>

    <p style="@messageStyles">
        @message
    </p>
```

```
    <p>
        <a href="http://www.startrek.com/">Star Trek</a>,
        &copy;1966-2019 CBS Studios, Inc. and
        <a href="https://www.paramount.com">Paramount Pictures</a>
    </p>
</EditForm>

@code {
    private bool disabled;
    private string message;
    private string messageStyles = "visibility:hidden";
    private CustomValidator customValidator;
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };

    private async Task HandleValidSubmit(EditContext editContext)
    {
        customValidator.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Count() > 0)
            {
                customValidator.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code: {response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {MESSAGE}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
        }
    }
}
```

NOTE

As an alternative to validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server-side validation, the attributes must be executable on the server. For more information, see Model validation in ASP.NET Core MVC.

NOTE

The server-side validation approach in this section is suitable for any of the Blazor WebAssembly hosted solution examples in this documentation set:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

InputText based on the input event

Use the InputText component to create a custom component that uses the `input` event instead of the `change` event.

In the following example, the `CustomInputText` component inherits the framework's `InputText` component and sets the event binding (CreateBinder) to the `oninput` event.

`Shared/CustomInputText.razor`:

```
@inherits InputText

<input
    @attributes="AdditionalAttributes"
    class="@CssClass"
    value="@CurrentValue"
    @oninput="EventCallback.Factory.CreateBinder<string>(
         this, __value => CurrentValueAsString = __value,
         CurrentValueAsString)" />
```

The `CustomInputText` component can be used anywhere InputText is used:

`Pages/TestForm.razor`:

```
@page "/testform"
@using System.ComponentModel.DataAnnotations;

<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <CustomInputText @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>

<p>
    CurrentValue: @exampleModel.Name
</p>

@code {
    private ExampleModel exampleModel = new ExampleModel();

    private void HandleValidSubmit()
    {
        ...
    }

    public class ExampleModel
    {
        [Required]
        [StringLength(10, ErrorMessage = "Name is too long.")]
        public string Name { get; set; }
    }
}
```

Radio buttons

Use `InputRadio` components with the `InputRadioGroup` component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Built-in forms components section:

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;

[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Add the following `enums` to the app. Create a new file to hold the `enums` or add the `enums` to the `Starship.cs` file. Make the `enums` accessible to the `Starship` model and the *Starfleet Starship Database* form:

```
public enum Manufacturer { SpaceX, NASA, ULA, Virgin, Unknown }
public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue, VoyagerOrange }
public enum Engine { Ion, Plasma, Fusion, Warp }
```

Update the *Starfleet Starship Database* form described in the Built-in forms components section. Add the components to produce:

- A radio button group for the ship manufacturer.

- A nested radio button group for ship color and engine.

```
<p>
    <InputRadioGroup @bind-Value="starship.Manufacturer">
        Manufacturer:
        <br>
        @foreach (var manufacturer in (Manufacturer[])Enum
            .GetValues(typeof(Manufacturer)))
        {
            <InputRadio Value="manufacturer" />
            @manufacturer
            <br>
        }
    </InputRadioGroup>
</p>

<p>
    Pick one color and one engine:
    <InputRadioGroup Name="engine" @bind-Value="starship.Engine">
        <InputRadioGroup Name="color" @bind-Value="starship.Color">
            <InputRadio Name="color" Value="Color.ImperialRed" />Imperial Red<br>
            <InputRadio Name="engine" Value="Engine.Ion" />Ion<br>
            <InputRadio Name="color" Value="Color.SpacecruiserGreen" />
                Spacecruiser Green<br>
            <InputRadio Name="engine" Value="Engine.Plasma" />Plasma<br>
            <InputRadio Name="color" Value="Color.StarshipBlue" />Starship Blue<br>
            <InputRadio Name="engine" Value="Engine.Fusion" />Fusion<br>
            <InputRadio Name="color" Value="Color.VoyagerOrange" />
                Voyager Orange<br>
            <InputRadio Name="engine" Value="Engine.Warp" />Warp<br>
        </InputRadioGroup>
    </InputRadioGroup>
</p>
```

> **NOTE**
>
> If `Name` is omitted, `InputRadio` components are grouped by their most recent ancestor.

When working with radio buttons in a form, data binding is handled differently than other elements because radio buttons are evaluated as a group. The value of each radio button is fixed, but the value of the radio button group is the value of the selected radio button. The following example shows how to:

- Handle data binding for a radio button group.
- Support validation using a custom `InputRadio` component.

```
@using System.Globalization
@typeparam TValue
@inherits InputBase<TValue>

<input @attributes="AdditionalAttributes" type="radio" value="@SelectedValue"
       checked="@(SelectedValue.Equals(Value))" @onchange="OnChange" />

@code {
    [Parameter]
    public TValue SelectedValue { get; set; }

    private void OnChange(ChangeEventArgs args)
    {
        CurrentValueAsString = args.Value.ToString();
    }

    protected override bool TryParseValueFromString(string value,
        out TValue result, out string errorMessage)
    {
        var success = BindConverter.TryConvertTo<TValue>(
            value, CultureInfo.CurrentCulture, out var parsedValue);
        if (success)
        {
            result = parsedValue;
            errorMessage = null;

            return true;
        }
        else
        {
            result = default;
            errorMessage = $"{FieldIdentifier.FieldName} field isn't valid.";

            return false;
        }
    }
}
```

The following EditForm uses the preceding `InputRadio` component to obtain and validate a rating from the user:

```
@page "/RadioButtonExample"
@using System.ComponentModel.DataAnnotations

<h1>Radio Button Group Test</h1>

<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    @for (int i = 1; i <= 5; i++)
    {
        <label>
            <InputRadio name="rate" SelectedValue="i" @bind-Value="model.Rating" />
            @i
        </label>
    }

    <button type="submit">Submit</button>
</EditForm>

<p>You chose: @model.Rating</p>

@code {
    private Model model = new Model();

    private void HandleValidSubmit()
    {
        ...
    }

    public class Model
    {
        [Range(1, 5)]
        public int Rating { get; set; }
    }
}
```

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

The Blazor framework doesn't automatically handle `null` to empty string conversions when attempting two-way binding to a `<select>`'s value. For more information, see Fix binding `<select>` to a null value (dotnet/aspnetcore #23221).

Validation support

The DataAnnotationsValidator component attaches validation support using data annotations to the cascaded

EditContext. Enabling support for validation using data annotations requires this explicit gesture. To use a different validation system than data annotations, replace the DataAnnotationsValidator with a custom implementation. The ASP.NET Core implementation is available for inspection in the reference source: `DataAnnotationsValidator` / `AddDataAnnotationsValidation`. The preceding links to reference source provide code from the repository's `master` branch, which represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release, use the GitHub branch selector (for example `release/3.1`).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation results that aren't associated with an individual member are associated with the model rather than a field.

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```
<ValidationSummary Model="@starship" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the `For` attribute and a lambda expression naming the model property:

```
<ValidationMessage For="@(() => starship.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```
.validation-message {
    color: red;
}
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult:

```
using System;
using System.ComponentModel.DataAnnotations;

private class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

NOTE

ValidationContext.GetService is `null`. Injecting services for validation in the `IsValid` method isn't supported.

Custom validation class attributes

Custom validation class names are useful when integrating with CSS frameworks, such as Bootstrap. To specify custom validation class names, create a class derived from `FieldCssClassProvider` and set the class on the EditContext instance:

```
var editContext = new EditContext(model);
editContext.SetFieldCssClassProvider(new MyFieldClassProvider());

...

private class MyFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();

        return isValid ? "good field" : "bad field";
    }
}
```

Blazor data annotations validation package

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

NOTE

The `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package has a latest version of *release candidate* at Nuget.org. Continue to use the *experimental* release candidate package at this time. The package's assembly might be moved to either the framework or the runtime in a future release. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

[CompareProperty] attribute

The CompareAttribute doesn't work well with the DataAnnotationsValidator component because it doesn't associate the validation result with a specific member. This can result in inconsistent behavior between field-level validation and when the entire model is validated on a submit. The

`Microsoft.AspNetCore.Components.DataAnnotations.Validation` *experimental* package introduces an additional validation attribute, `ComparePropertyAttribute`, that works around these limitations. In a Blazor app, `[CompareProperty]` is a direct replacement for the `[Compare]` attribute.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* `Microsoft.AspNetCore.Components.DataAnnotations.Validation` package:

```
<EditForm Model="@model" OnValidSubmit="@HandleValidSubmit">
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } =
        new ShipDescription();

    ...
}
```

`ShipDescription.cs`:

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation:

- Use the form's EditContext to assign the model when the component is initialized.
- Validate the form in the context's OnFieldChanged callback to enable and disable the submit button.

- Unhook the event handler in the `Dispose` method. For more information, see ASP.NET Core Blazor lifecycle.

> **NOTE**
> When using an EditContext, don't also assign a Model to the EditForm.

```
@implements IDisposable

<EditForm EditContext="@editContext">
    <DataAnnotationsValidator />
    <ValidationSummary />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private Starship starship = new Starship() { ProductionDate = DateTime.UtcNow };
    private bool formInvalid = true;
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(starship);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
    {
        formInvalid = !editContext.Validate();
        StateHasChanged();
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

In the preceding example, set `formInvalid` to `false` if:

- The form is preloaded with valid default values.
- You want the submit button enabled when the form loads.

A side effect of the preceding approach is that a ValidationSummary component is populated with invalid fields after the user interacts with any one field. This scenario can be addressed in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `HandleValidSubmit` method).

```
<EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void HandleValidSubmit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm has a Model **or** EditContext. Don't use both for the same form.

When assigning a Model to the form, confirm that the model type is instantiated, as the following example shows:

```
private ExampleModel exampleModel = new ExampleModel();
```

Additional resources

- ASP.NET Core Blazor file uploads

ASP.NET Core Blazor file uploads

9/22/2020 • 2 minutes to read • Edit Online

By Daniel Roth

Use the `InputFile` component to read browser file data into .NET code, including for file uploads. The `InputFile` component renders as an HTML input of type `file`.

By default, the user selects single files. Add the `multiple` attribute to permit the user to upload multiple files at once. When one or more files is selected by the user, the `InputFile` component fires an `OnChange` event and passes in an `InputFileChangeEventArgs` that provides access to the selected file list and details about each file.

A component that receives an image file can call the `RequestImageFileAsync` convenience method on the file to resize the image data within the browser's JavaScript runtime before the image is streamed into the app.

The following example demonstrates multiple image file upload in a component:

```
<h3>Upload PNG images</h3>

<p>
    <InputFile OnChange="@OnInputFileChange" multiple />
</p>

@if (imageDataUrls.Count > 0)
{
    <h3>Images</h3>

    <div class="card" style="width:30rem;">
        <div class="card-body">
            @foreach (var imageDataUrl in imageDataUrls)
            {
                <img class="rounded m-1" src="@imageDataUrl" />
            }
        </div>
    </div>
}

@code {
    IList<string> imageDataUrls = new List<string>();

    private async Task OnInputFileChange(InputFileChangeEventArgs e)
    {
        var imageFiles = e.GetMultipleFiles();
        var format = "image/png";

        foreach (var imageFile in imageFiles)
        {
            var resizedImageFile = await imageFile.RequestImageFileAsync(format,
                100, 100);
            var buffer = new byte[resizedImageFile.Size];
            await resizedImageFile.OpenReadStream().ReadAsync(buffer);
            var imageDataUrl =
                $"data:{format};base64,{Convert.ToBase64String(buffer)}";
            imageDataUrls.Add(imageDataUrl);
        }
    }
}
```

To read data from a user-selected file, call `OpenReadStream` on the file and read from the returned stream. In a Blazor WebAssembly app, the data is streamed directly into the .NET code within the browser. In a Blazor Server app, the file data is streamed into .NET code on the server as the file is read from the stream.

Call JavaScript functions from .NET methods in ASP.NET Core Blazor

9/22/2020 • 16 minutes to read • Edit Online

By Javier Calvarro Nelson, Daniel Roth, and Luke Latham

A Blazor app can invoke JavaScript functions from .NET methods and .NET methods from JavaScript functions. These scenarios are called *JavaScript interoperability* (*JS interop*).

This article covers invoking JavaScript functions from .NET. For information on how to call .NET methods from JavaScript, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor.

View or download sample code (how to download)

To call into JavaScript from .NET, use the IJSRuntime abstraction. To issue JS interop calls, inject the IJSRuntime abstraction in your component. InvokeAsync takes an identifier for the JavaScript function that you wish to invoke along with any number of JSON-serializable arguments. The function identifier is relative to the global scope (`window`). If you wish to call `window.someScope.someFunction` , the identifier is `someScope.someFunction` . There's no need to register the function before it's called. The return type `T` must also be JSON serializable. `T` should match the .NET type that best maps to the JSON type returned.

For Blazor Server apps with prerendering enabled, calling into JavaScript isn't possible during the initial prerendering. JavaScript interop calls must be deferred until after the connection with the browser is established. For more information, see the Detect when a Blazor Server app is prerendering section.

The following example is based on `TextDecoder` , a JavaScript-based decoder. The example demonstrates how to invoke a JavaScript function from a C# method that offloads a requirement from developer code to an existing JavaScript API. The JavaScript function accepts a byte array from a C# method, decodes the array, and returns the text to the component for display.

Inside the `<head>` element of `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server), provide a JavaScript function that uses `TextDecoder` to decode a passed array and return the decoded value:

```
<script>
  window.convertArray = (win1251Array) => {
    var win1251decoder = new TextDecoder('windows-1251');
    var bytes = new Uint8Array(win1251Array);
    var decodedArray = win1251decoder.decode(bytes);
    console.log(decodedArray);
    return decodedArray;
  };
</script>
```

JavaScript code, such as the code shown in the preceding example, can also be loaded from a JavaScript file (`.js`) with a reference to the script file:

```
<script src="exampleJsInterop.js"></script>
```

The following component:

- Invokes the `convertArray` JavaScript function using `JSRuntime` when a component button (`Convert Array`) is selected.

- After the JavaScript function is called, the passed array is converted into a string. The string is returned to the component for display.

```
@page "/call-js-example"
@inject IJSRuntime JSRuntime;

<h1>Call JavaScript Function Example</h1>

<button type="button" class="btn btn-primary" @onclick="ConvertArray">
    Convert Array
</button>

<p class="mt-2" style="font-size:1.6em">
    <span class="badge badge-success">
        @convertedText
    </span>
</p>

@code {
    // Quote (c)2005 Universal Pictures: Serenity
    // https://www.uphe.com/movies/serenity
    // David Krumholtz on IMDB: https://www.imdb.com/name/nm0472710/

    private MarkupString convertedText =
        new MarkupString("Select the <b>Convert Array</b> button.");

    private uint[] quoteArray = new uint[]
        {
            60, 101, 109, 62, 67, 97, 110, 39, 116, 32, 115, 116, 111, 112, 32,
            116, 104, 101, 32, 115, 105, 103, 110, 97, 108, 44, 32, 77, 97,
            108, 46, 60, 47, 101, 109, 62, 32, 45, 32, 77, 114, 46, 32, 85, 110,
            105, 118, 101, 114, 115, 101, 10, 10,
        };

    private async Task ConvertArray()
    {
        var text =
            await JSRuntime.InvokeAsync<string>("convertArray", quoteArray);

        convertedText = new MarkupString(text);
    }
}
```

IJSRuntime

To use the IJSRuntime abstraction, adopt any of the following approaches:

- Inject the IJSRuntime abstraction into the Razor component (`.razor`):

  ```
  @inject IJSRuntime JSRuntime

  @code {
      protected override void OnInitialized()
      {
          StocksService.OnStockTickerUpdated += stockUpdate =>
          {
              JSRuntime.InvokeVoidAsync("handleTickerChanged",
                  stockUpdate.symbol, stockUpdate.price);
          };
      }
  }
  ```

  Inside the `<head>` element of `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor

Server), provide a `handleTickerChanged` JavaScript function. The function is called with JSRuntimeExtensions.InvokeVoidAsync and doesn't return a value:

```
<script>
  window.handleTickerChanged = (symbol, price) => {
    // ... client-side processing/display code ...
  };
</script>
```

- Inject the IJSRuntime abstraction into a class (`.cs`):

```
public class JsInteropClasses
{
    private readonly IJSRuntime jsRuntime;

    public JsInteropClasses(IJSRuntime jsRuntime)
    {
        this.jsRuntime = jsRuntime;
    }

    public ValueTask<string> TickerChanged(string data)
    {
        return jsRuntime.InvokeAsync<string>(
            "handleTickerChanged",
            stockUpdate.symbol,
            stockUpdate.price);
    }
}
```

Inside the `<head>` element of `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server), provide a `handleTickerChanged` JavaScript function. The function is called with `JSRuntime.InvokeAsync` and returns a value:

```
<script>
  window.handleTickerChanged = (symbol, price) => {
    // ... client-side processing/display code ...
    return 'Done!';
  };
</script>
```

- For dynamic content generation with BuildRenderTree, use the `[Inject]` attribute:

```
[Inject]
IJSRuntime JSRuntime { get; set; }
```

In the client-side sample app that accompanies this topic, two JavaScript functions are available to the app that interact with the DOM to receive user input and display a welcome message:

- `showPrompt` : Produces a prompt to accept user input (the user's name) and returns the name to the caller.
- `displayWelcome` : Assigns a welcome message from the caller to a DOM object with an `id` of `welcome` .

`wwwroot/exampleJsInterop.js` :

```
window.exampleJsFunctions = {
  showPrompt: function (text) {
    return prompt(text, 'Type your name here');
  },
  displayWelcome: function (welcomeMessage) {
    document.getElementById('welcome').innerText = welcomeMessage;
  },
  returnArrayAsyncJs: function () {
    DotNet.invokeMethodAsync('BlazorSample', 'ReturnArrayAsync')
      .then(data => {
        data.push(4);
          console.log(data);
      });
  },
  sayHello: function (dotnetHelper) {
    return dotnetHelper.invokeMethodAsync('SayHello')
      .then(r => console.log(r));
  }
};
```

Place the `<script>` tag that references the JavaScript file in the `wwwroot/index.html` file (Blazor WebAssembly) or `Pages/_Host.cshtml` file (Blazor Server).

`wwwroot/index.html` (Blazor WebAssembly):

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-
scalable=no" />
    <title>Blazor WebAssembly Sample</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
</head>

<body>
    <app>Loading...</app>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">🗙</a>
    </div>
    <script src="_framework/blazor.webassembly.js"></script>
    <script src="exampleJsInterop.js"></script>
</body>

</html>
```

`Pages/_Host.cshtml` (Blazor Server):

```
@page "/"
@namespace BlazorSample.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
    Layout = null;
}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Blazor Server Sample</title>
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
</head>
<body>
    <app>
        <component type="typeof(App)" render-mode="ServerPrerendered" />
    </app>

    <div id="blazor-error-ui">
        <environment include="Staging,Production">
            An error has occurred. This application may no longer respond until reloaded.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred. See browser dev tools for details.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">🗙</a>
    </div>

    <script src="_framework/blazor.server.js"></script>
    <script src="exampleJsInterop.js"></script>
</body>
</html>
```

Don't place a `<script>` tag in a component file because the `<script>` tag can't be updated dynamically.

.NET methods interop with the JavaScript functions in the `exampleJsInterop.js` file by calling IJSRuntime.InvokeAsync.

The IJSRuntime abstraction is asynchronous to allow for Blazor Server scenarios. If the app is a Blazor WebAssembly app and you want to invoke a JavaScript function synchronously, downcast to IJSInProcessRuntime and call Invoke instead. We recommend that most JS interop libraries use the async APIs to ensure that the libraries are available in all scenarios.

> **NOTE**
>
> To enable JavaScript isolation in standard JavaScript modules, see the Blazor JavaScript isolation and object references section.

The sample app includes a component to demonstrate JS interop. The component:

- Receives user input via a JavaScript prompt.
- Returns the text to the component for processing.
- Calls a second JavaScript function that interacts with the DOM to display a welcome message.

`Pages/JsInterop.razor`:

```
@page "/JSInterop"
@using {APP ASSEMBLY}.JsInteropClasses
@inject IJSRuntime JSRuntime

<h1>JavaScript Interop</h1>

<h2>Invoke JavaScript functions from .NET methods</h2>

<button type="button" class="btn btn-primary" @onclick="TriggerJsPrompt">
    Trigger JavaScript Prompt
</button>

<h3 id="welcome" style="color:green;font-style:italic"></h3>

@code {
    public async Task TriggerJsPrompt()
    {
        var name = await JSRuntime.InvokeAsync<string>(
                "exampleJsFunctions.showPrompt",
                "What's your name?");

        await JSRuntime.InvokeVoidAsync(
                "exampleJsFunctions.displayWelcome",
                $"Hello {name}! Welcome to Blazor!");
    }
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

1. When `TriggerJsPrompt` is executed by selecting the component's `Trigger JavaScript Prompt` button, the JavaScript `showPrompt` function provided in the `wwwroot/exampleJsInterop.js` file is called.
2. The `showPrompt` function accepts user input (the user's name), which is HTML-encoded and returned to the component. The component stores the user's name in a local variable, `name`.
3. The string stored in `name` is incorporated into a welcome message, which is passed to a JavaScript function, `displayWelcome`, which renders the welcome message into a heading tag.

Call a void JavaScript function

JavaScript functions that return void(0)/void 0 or undefined are called with JSRuntimeExtensions.InvokeVoidAsync.

Detect when a Blazor Server app is prerendering

While a Blazor Server app is prerendering, certain actions, such as calling into JavaScript, aren't possible because a connection with the browser hasn't been established. Components may need to render differently when prerendered.

To delay JavaScript interop calls until after the connection with the browser is established, you can use the OnAfterRenderAsync component lifecycle event. This event is only called after the app is fully rendered and the client connection is established.

```
@using Microsoft.JSInterop
@inject IJSRuntime JSRuntime

<div @ref="divElement">Text during render</div>

@code {
    private ElementReference divElement;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            await JSRuntime.InvokeVoidAsync(
                "setElementText", divElement, "Text after render");
        }
    }
}
```

For the preceding example code, provide a `setElementText` JavaScript function inside the `<head>` element of `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server). The function is called with JSRuntimeExtensions.InvokeVoidAsync and doesn't return a value:

```
<script>
  window.setElementText = (element, text) => element.innerText = text;
</script>
```

WARNING

The preceding example modifies the Document Object Model (DOM) directly for demonstration purposes only. Directly modifying the DOM with JavaScript isn't recommended in most scenarios because JavaScript can interfere with Blazor's change tracking.

The following component demonstrates how to use JavaScript interop as part of a component's initialization logic in a way that's compatible with prerendering. The component shows that it's possible to trigger a rendering update from inside OnAfterRenderAsync. The developer must avoid creating an infinite loop in this scenario.

Where JSRuntime.InvokeAsync is called, `ElementRef` is only used in OnAfterRenderAsync and not in any earlier lifecycle method because there's no JavaScript element until after the component is rendered.

StateHasChanged is called to rerender the component with the new state obtained from the JavaScript interop call. The code doesn't create an infinite loop because `StateHasChanged` is only called when `infoFromJs` is `null`.

```
@page "/prerendered-interop"
@using Microsoft.AspNetCore.Components
@using Microsoft.JSInterop
@inject IJSRuntime JSRuntime

<p>
    Get value via JS interop call:
    <strong id="val-get-by-interop">@(infoFromJs ?? "No value yet")</strong>
</p>

Set value via JS interop call:
<div id="val-set-by-interop" @ref="divElement"></div>

@code {
    private string infoFromJs;
    private ElementReference divElement;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender && infoFromJs == null)
        {
            infoFromJs = await JSRuntime.InvokeAsync<string>(
                "setElementText", divElement, "Hello from interop call!");

            StateHasChanged();
        }
    }
}
```

For the preceding example code, provide a `setElementText` JavaScript function inside the `<head>` element of `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server). The function is called withIJSRuntime.InvokeAsync and returns a value:

```
<script>
  window.setElementText = (element, text) => {
    element.innerText = text;
    return text;
  };
</script>
```

WARNING

The preceding example modifies the Document Object Model (DOM) directly for demonstration purposes only. Directly modifying the DOM with JavaScript isn't recommended in most scenarios because JavaScript can interfere with Blazor's change tracking.

Capture references to elements

Some JS interop scenarios require references to HTML elements. For example, a UI library may require an element reference for initialization, or you might need to call command-like APIs on an element, such as `focus` or `play`.

Capture references to HTML elements in a component using the following approach:

- Add an `@ref` attribute to the HTML element.
- Define a field of type ElementReference whose name matches the value of the `@ref` attribute.

The following example shows capturing a reference to the `username` `<input>` element:

```
<input @ref="username" ... />

@code {
    ElementReference username;
}
```

WARNING

Only use an element reference to mutate the contents of an empty element that doesn't interact with Blazor. This scenario is useful when a third-party API supplies content to the element. Because Blazor doesn't interact with the element, there's no possibility of a conflict between Blazor's representation of the element and the DOM.

In the following example, it's *dangerous* to mutate the contents of the unordered list (`ul`) because Blazor interacts with the DOM to populate this element's list items (`<li>`):

```
<ul ref="MyList">
    @foreach (var item in Todos)
    {
        <li>@item.Text</li>
    }
</ul>
```

If JS interop mutates the contents of element `MyList` and Blazor attempts to apply diffs to the element, the diffs won't match the DOM.

As far as .NET code is concerned, an ElementReference is an opaque handle. The *only* thing you can do with ElementReference is pass it through to JavaScript code via JS interop. When you do so, the JavaScript-side code receives an `HTMLElement` instance, which it can use with normal DOM APIs.

For example, the following code defines a .NET extension method that enables setting the focus on an element:

`exampleJsInterop.js`:

```
window.exampleJsFunctions = {
  focusElement : function (element) {
    element.focus();
  }
}
```

To call a JavaScript function that doesn't return a value, use JSRuntimeExtensions.InvokeVoidAsync. The following code sets the focus on the username input by calling the preceding JavaScript function with the captured ElementReference:

```
@inject IJSRuntime JSRuntime

<input @ref="username" />
<button @onclick="SetFocus">Set focus on username</button>

@code {
    private ElementReference username;

    public async Task SetFocus()
    {
        await JSRuntime.InvokeVoidAsync(
            "exampleJsFunctions.focusElement", username);
    }
}
```

To use an extension method, create a static extension method that receives the IJSRuntime instance:

```
public static async Task Focus(this ElementReference elementRef, IJSRuntime jsRuntime)
{
    await jsRuntime.InvokeVoidAsync(
        "exampleJsFunctions.focusElement", elementRef);
}
```

The `Focus` method is called directly on the object. The following example assumes that the `Focus` method is available from the `JsInteropClasses` namespace:

```
@inject IJSRuntime JSRuntime
@using JsInteropClasses

<input @ref="username" />
<button @onclick="SetFocus">Set focus on username</button>

@code {
    private ElementReference username;

    public async Task SetFocus()
    {
        await username.Focus(JSRuntime);
    }
}
```

IMPORTANT

The `username` variable is only populated after the component is rendered. If an unpopulated ElementReference is passed to JavaScript code, the JavaScript code receives a value of `null`. To manipulate element references after the component has finished rendering (to set the initial focus on an element) use the `OnAfterRenderAsync` or `OnAfterRender` component lifecycle methods.

When working with generic types and returning a value, use ValueTask<TResult>:

```
public static ValueTask<T> GenericMethod<T>(this ElementReference elementRef,
    IJSRuntime jsRuntime)
{
    return jsRuntime.InvokeAsync<T>(
        "exampleJsFunctions.doSomethingGeneric", elementRef);
}
```

`GenericMethod` is called directly on the object with a type. The following example assumes that the `GenericMethod` is available from the `JsInteropClasses` namespace:

```
@inject IJSRuntime JSRuntime
@using JsInteropClasses

<input @ref="username" />
<button @onclick="OnClickMethod">Do something generic</button>

<p>
    returnValue: @returnValue
</p>

@code {
    private ElementReference username;
    private string returnValue;

    private async Task OnClickMethod()
    {
        returnValue = await username.GenericMethod<string>(JSRuntime);
    }
}
```

Reference elements across components

An ElementReference is only guaranteed valid in a component's OnAfterRender method (and an element reference is a `struct`), so an element reference can't be passed between components.

For a parent component to make an element reference available to other components, the parent component can:

- Allow child components to register callbacks.
- Invoke the registered callbacks during the OnAfterRender event with the passed element reference. Indirectly, this approach allows child components to interact with the parent's element reference.

The following Blazor WebAssembly example illustrates the approach.

In the `<head>` of `wwwroot/index.html`:

```
<style>
    .red { color: red }
</style>
```

In the `<body>` of `wwwroot/index.html`:

```
<script>
    function setElementClass(element, className) {
        /** @type {HTMLElement} **/
        var myElement = element;
        myElement.classList.add(className);
    }
</script>
```

`Pages/Index.razor` (parent component):

```
@page "/"

<h1 @ref="title">Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Parent="this" Title="How is Blazor working for you?" />
```

`Pages/Index.razor.cs`:

```
using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Components;

namespace {APP ASSEMBLY}.Pages
{
    public partial class Index :
        ComponentBase, IObservable<ElementReference>, IDisposable
    {
        private bool disposing;
        private IList<IObserver<ElementReference>> subscriptions =
            new List<IObserver<ElementReference>>();
        private ElementReference title;

        protected override void OnAfterRender(bool firstRender)
        {
            base.OnAfterRender(firstRender);

            foreach (var subscription in subscriptions)
            {
                try
                {
                    subscription.OnNext(title);
                }
                catch (Exception)
                {
                    throw;
                }
            }
        }

        public void Dispose()
        {
            disposing = true;

            foreach (var subscription in subscriptions)
            {
                try
                {
                    subscription.OnCompleted();
                }
                catch (Exception)
                {
                }
            }

            subscriptions.Clear();
        }

        public IDisposable Subscribe(IObserver<ElementReference> observer)
        {
            if (disposing)
            {
                throw new InvalidOperationException("Parent being disposed");
            }

            subscriptions.Add(observer);

            return new Subscription(observer, this);
        }

        private class Subscription : IDisposable
        {
            public Subscription(IObserver<ElementReference> observer, Index self)
            {
                Observer = observer;
```

```
                Self = self;
            }

            public IObserver<ElementReference> Observer { get; }
            public Index Self { get; }

            public void Dispose()
            {
                Self.subscriptions.Remove(Observer);
            }
        }
    }
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

`Shared/SurveyPrompt.razor` (child component):

```
@inject IJSRuntime JS

<div class="alert alert-secondary mt-4" role="alert">
    <span class="oi oi-pencil mr-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold"
            href="https://go.microsoft.com/fwlink/?linkid=2109206">brief survey</a>
    </span>
    and tell us what you think.
</div>

@code {
    [Parameter]
    public string Title { get; set; }
}
```

`Shared/SurveyPrompt.razor.cs`:

```
using System;
using Microsoft.AspNetCore.Components;

namespace {APP ASSEMBLY}.Shared
{
    public partial class SurveyPrompt :
        ComponentBase, IObserver<ElementReference>, IDisposable
    {
        private IDisposable subscription = null;

        [Parameter]
        public IObservable<ElementReference> Parent { get; set; }

        protected override void OnParametersSet()
        {
            base.OnParametersSet();

            if (subscription != null)
            {
                subscription.Dispose();
            }

            subscription = Parent.Subscribe(this);
        }

        public void OnCompleted()
        {
            subscription = null;
        }

        public void OnError(Exception error)
        {
            subscription = null;
        }

        public void OnNext(ElementReference value)
        {
            JS.InvokeAsync<object>(
                "setElementClass", new object[] { value, "red" });
        }

        public void Dispose()
        {
            subscription?.Dispose();
        }
    }
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

Harden JS interop calls

JS interop may fail due to networking errors and should be treated as unreliable. By default, a Blazor Server app times out JS interop calls on the server after one minute. If an app can tolerate a more aggressive timeout, set the timeout using one of the following approaches:

- Globally in `Startup.ConfigureServices`, specify the timeout:

  ```
  services.AddServerSideBlazor(
      options => options.JSInteropDefaultCallTimeout = TimeSpan.FromSeconds({SECONDS}));
  ```

- Per-invocation in component code, a single call can specify the timeout:

```
var result = await JSRuntime.InvokeAsync<string>("MyJSOperation",
    TimeSpan.FromSeconds({SECONDS}), new[] { "Arg1" });
```

For more information on resource exhaustion, see Threat mitigation guidance for ASP.NET Core Blazor Server.

Share interop code in a class library

JS interop code can be included in a class library, which allows you to share the code in a NuGet package.

The class library handles embedding JavaScript resources in the built assembly. The JavaScript files are placed in the `wwwroot` folder. The tooling takes care of embedding the resources when the library is built.

The built NuGet package is referenced in the app's project file the same way that any NuGet package is referenced. After the package is restored, app code can call into JavaScript as if it were C#.

For more information, see ASP.NET Core Razor components class libraries.

Avoid circular object references

Objects that contain circular references can't be serialized on the client for either:

- .NET method calls.
- JavaScript method calls from C# when the return type has circular references.

For more information, see the following issues:

- Circular references are not supported, take two (dotnet/aspnetcore #20525)
- Proposal: Add mechanism to handle circular references when serializing (dotnet/runtime #30820)

Blazor JavaScript isolation and object references

Blazor enables JavaScript isolation in standard JavaScript modules. JavaScript isolation provides the following benefits:

- Imported JavaScript no longer pollutes the global namespace.
- Consumers of a library and components aren't required to import the related JavaScript.

For example, the following JavaScript module exports a JavaScript function for showing a browser prompt:

```
export function showPrompt(message) {
  return prompt(message, 'Type anything here');
}
```

Add the preceding JavaScript module to a .NET library as a static web asset (`wwwroot/exampleJsInterop.js`) and then import the module into the .NET code using the IJSRuntime service. The service is injected as `jsRuntime` (not shown) for the following example:

```
var module = await jsRuntime.InvokeAsync<JSObjectReference>(
    "import", "./_content/MyComponents/exampleJsInterop.js");
```

The `import` identifier in the preceding example is a special identifier used specifically for importing a JavaScript module. Specify the module using its stable static web asset path: `_content/{LIBRARY NAME}/{PATH UNDER WWWROOT}`. The placeholder `{LIBRARY NAME}` is the library name. The placeholder `{PATH UNDER WWWROOT}` is the path to the script under `wwwroot` .

IJSRuntime imports the module as a `JSObjectReference`, which represents a reference to a JavaScript object from .NET code. Use the `JSObjectReference` to invoke exported JavaScript functions from the module:

```
public async ValueTask<string> Prompt(string message)
{
    return await module.InvokeAsync<string>("showPrompt", message);
}
```

Additional resources

- Call .NET methods from JavaScript functions in ASP.NET Core Blazor
- InteropComponent.razor example (dotnet/AspNetCore GitHub repository, 3.1 release branch)
- Perform large data transfers in Blazor Server apps

Call .NET methods from JavaScript functions in ASP.NET Core Blazor

9/22/2020 • 8 minutes to read • Edit Online

By Javier Calvarro Nelson, Daniel Roth, Shashikant Rudrawadi, and Luke Latham

A Blazor app can invoke JavaScript functions from .NET methods and .NET methods from JavaScript functions. These scenarios are called *JavaScript interoperability* (*JS interop*).

This article covers invoking .NET methods from JavaScript. For information on how to call JavaScript functions from .NET, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

View or download sample code (how to download)

Static .NET method call

To invoke a static .NET method from JavaScript, use the `DotNet.invokeMethod` or `DotNet.invokeMethodAsync` functions. Pass in the identifier of the static method you wish to call, the name of the assembly containing the function, and any arguments. The asynchronous version is preferred to support Blazor Server scenarios. The .NET method must be public, static, and have the `[JSInvokable]` attribute. Calling open generic methods isn't currently supported.

The sample app includes a C# method to return an `int` array. The `[JSInvokable]` attribute is applied to the method.

`Pages/JsInterop.razor`:

```
<button type="button" class="btn btn-primary"
        onclick="exampleJsFunctions.returnArrayAsyncJs()">
    Trigger .NET static method ReturnArrayAsync
</button>

@code {
    [JSInvokable]
    public static Task<int[]> ReturnArrayAsync()
    {
        return Task.FromResult(new int[] { 1, 2, 3 });
    }
}
```

JavaScript served to the client invokes the C# .NET method.

`wwwroot/exampleJsInterop.js`:

```
window.exampleJsFunctions = {
  showPrompt: function (text) {
    return prompt(text, 'Type your name here');
  },
  displayWelcome: function (welcomeMessage) {
    document.getElementById('welcome').innerText = welcomeMessage;
  },
  returnArrayAsyncJs: function () {
    DotNet.invokeMethodAsync('BlazorSample', 'ReturnArrayAsync')
      .then(data => {
        data.push(4);
          console.log(data);
      });
  },
  sayHello: function (dotnetHelper) {
    return dotnetHelper.invokeMethodAsync('SayHello')
      .then(r => console.log(r));
  }
};
```

When the `Trigger .NET static method ReturnArrayAsync` button is selected, examine the console output in the browser's web developer tools.

The console output is:

```
Array(4) [ 1, 2, 3, 4 ]
```

The fourth array value is pushed to the array (`data.push(4);`) returned by `ReturnArrayAsync` .

By default, the method identifier is the method name, but you can specify a different identifier using the `[JSInvokable]` attribute constructor:

```
@code {
    [JSInvokable("DifferentMethodName")]
    public static Task<int[]> ReturnArrayAsync()
    {
        return Task.FromResult(new int[] { 1, 2, 3 });
    }
}
```

In the client-side JavaScript file:

```
returnArrayAsyncJs: function () {
  DotNet.invokeMethodAsync('{APP ASSEMBLY}', 'DifferentMethodName')
    .then(data => {
      data.push(4);
      console.log(data);
    });
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

Instance method call

You can also call .NET instance methods from JavaScript. To invoke a .NET instance method from JavaScript:

- Pass the .NET instance by reference to JavaScript:
  - Make a static call to DotNetObjectReference.Create.

    - Wrap the instance in a DotNetObjectReference instance and call Create on the DotNetObjectReference instance. Dispose of DotNetObjectReference objects (an example appears later in this section).
- Invoke .NET instance methods on the instance using the `invokeMethod` or `invokeMethodAsync` functions. The .NET instance can also be passed as an argument when invoking other .NET methods from JavaScript.

> **NOTE**
>
> The sample app logs messages to the client-side console. For the following examples demonstrated by the sample app, examine the browser's console output in the browser's developer tools.

When the `Trigger .NET instance method HelloHelper.SayHello` button is selected, `ExampleJsInterop.CallHelloHelperSayHello` is called and passes a name, `Blazor`, to the method.

`Pages/JsInterop.razor`:

```
<button type="button" class="btn btn-primary" @onclick="TriggerNetInstanceMethod">
    Trigger .NET instance method HelloHelper.SayHello
</button>

@code {
    public async Task TriggerNetInstanceMethod()
    {
        var exampleJsInterop = new ExampleJsInterop(JSRuntime);
        await exampleJsInterop.CallHelloHelperSayHello("Blazor");
    }
}
```

`CallHelloHelperSayHello` invokes the JavaScript function `sayHello` with a new instance of `HelloHelper`.

`JsInteropClasses/ExampleJsInterop.cs`:

```
public class ExampleJsInterop : IDisposable
{
    private readonly IJSRuntime jsRuntime;
    private DotNetObjectReference<HelloHelper> objRef;

    public ExampleJsInterop(IJSRuntime jsRuntime)
    {
        this.jsRuntime = jsRuntime;
    }

    public ValueTask<string> CallHelloHelperSayHello(string name)
    {
        objRef = DotNetObjectReference.Create(new HelloHelper(name));

        return jsRuntime.InvokeAsync<string>(
            "exampleJsFunctions.sayHello",
            objRef);
    }

    public void Dispose()
    {
        objRef?.Dispose();
    }
}
```

`wwwroot/exampleJsInterop.js`:

```
window.exampleJsFunctions = {
  showPrompt: function (text) {
    return prompt(text, 'Type your name here');
  },
  displayWelcome: function (welcomeMessage) {
    document.getElementById('welcome').innerText = welcomeMessage;
  },
  returnArrayAsyncJs: function () {
    DotNet.invokeMethodAsync('BlazorSample', 'ReturnArrayAsync')
      .then(data => {
        data.push(4);
          console.log(data);
      });
  },
  sayHello: function (dotnetHelper) {
    return dotnetHelper.invokeMethodAsync('SayHello')
      .then(r => console.log(r));
  }
};
```

The name is passed to `HelloHelper`'s constructor, which sets the `HelloHelper.Name` property. When the JavaScript function `sayHello` is executed, `HelloHelper.SayHello` returns the `Hello, {Name}!` message, which is written to the console by the JavaScript function.

`JsInteropClasses/HelloHelper.cs`:

```
public class HelloHelper
{
    public HelloHelper(string name)
    {
        Name = name;
    }

    public string Name { get; set; }

    [JSInvokable]
    public string SayHello() => $"Hello, {Name}!";
}
```

Console output in the browser's web developer tools:

```
Hello, Blazor!
```

To avoid a memory leak and allow garbage collection on a component that creates a DotNetObjectReference, adopt one of the following approaches:

- Dispose of the object in the class that created the DotNetObjectReference instance:

```
public class ExampleJsInterop : IDisposable
{
    private readonly IJSRuntime jsRuntime;
    private DotNetObjectReference<HelloHelper> objRef;

    public ExampleJsInterop(IJSRuntime jsRuntime)
    {
        this.jsRuntime = jsRuntime;
    }

    public ValueTask<string> CallHelloHelperSayHello(string name)
    {
        objRef = DotNetObjectReference.Create(new HelloHelper(name));

        return jsRuntime.InvokeAsync<string>(
            "exampleJsFunctions.sayHello",
            objRef);
    }

    public void Dispose()
    {
        objRef?.Dispose();
    }
}
```

The preceding pattern shown in the `ExampleJsInterop` class can also be implemented in a component:

```
@page "/JSInteropComponent"
@using {APP ASSEMBLY}.JsInteropClasses
@implements IDisposable
@inject IJSRuntime JSRuntime

<h1>JavaScript Interop</h1>

<button type="button" class="btn btn-primary" @onclick="TriggerNetInstanceMethod">
    Trigger .NET instance method HelloHelper.SayHello
</button>

@code {
    private DotNetObjectReference<HelloHelper> objRef;

    public async Task TriggerNetInstanceMethod()
    {
        objRef = DotNetObjectReference.Create(new HelloHelper("Blazor"));

        await JSRuntime.InvokeAsync<string>(
            "exampleJsFunctions.sayHello",
            objRef);
    }

    public void Dispose()
    {
        objRef?.Dispose();
    }
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

- When the component or class doesn't dispose of the DotNetObjectReference, dispose of the object on the client by calling `.dispose()`:

```
window.myFunction = (dotnetHelper) => {
  dotnetHelper.invokeMethodAsync('{APP ASSEMBLY}', 'MyMethod');
  dotnetHelper.dispose();
}
```

Component instance method call

To invoke a component's .NET methods:

- Use the `invokeMethod` or `invokeMethodAsync` function to make a static method call to the component.
- The component's static method wraps the call to its instance method as an invoked Action.

> **NOTE**
>
> For Blazor Server apps, where several users might be concurrently using the same component, use a helper class to invoke instance methods.
>
> For more information, see the Component instance method helper class section.

In the client-side JavaScript:

```
function updateMessageCallerJS() {
  DotNet.invokeMethodAsync('{APP ASSEMBLY}', 'UpdateMessageCaller');
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

`Pages/JSInteropComponent.razor`:

```
@page "/JSInteropComponent"

<p>
    Message: @message
</p>

<p>
    <button onclick="updateMessageCallerJS()">Call JS Method</button>
</p>

@code {
    private static Action action;
    private string message = "Select the button.";

    protected override void OnInitialized()
    {
        action = UpdateMessage;
    }

    private void UpdateMessage()
    {
        message = "UpdateMessage Called!";
        StateHasChanged();
    }

    [JSInvokable]
    public static void UpdateMessageCaller()
    {
        action.Invoke();
    }
}
```

To pass arguments to the instance method:

- Add parameters to the JS method invocation. In the following example, a name is passed to the method. Additional parameters can be added to the list as needed.

  ```
  function updateMessageCallerJS(name) {
    DotNet.invokeMethodAsync('{APP ASSEMBLY}', 'UpdateMessageCaller', name);
  }
  ```

  The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

- Provide the correct types to the Action for the parameters. Provide the parameter list to the C# methods. Invoke the Action (`UpdateMessage`) with the parameters (`action.Invoke(name)`).

  `Pages/JSInteropComponent.razor`:

```
@page "/JSInteropComponent"

<p>
    Message: @message
</p>

<p>
    <button onclick="updateMessageCallerJS('Sarah Jane')">
        Call JS Method
    </button>
</p>

@code {
    private static Action<string> action;
    private string message = "Select the button.";

    protected override void OnInitialized()
    {
        action = UpdateMessage;
    }

    private void UpdateMessage(string name)
    {
        message = $"{name}, UpdateMessage Called!";
        StateHasChanged();
    }

    [JSInvokable]
    public static void UpdateMessageCaller(string name)
    {
        action.Invoke(name);
    }
}
```

Output `message` when the **Call JS Method** button is selected:

```
Sarah Jane, UpdateMessage Called!
```

Component instance method helper class

The helper class is used to invoke an instance method as an Action. Helper classes are useful when:

- Several components of the same type are rendered on the same page.
- A Blazor Server app is used, where multiple users might be using a component concurrently.

In the following example:

- The `JSInteropExample` component contains several `ListItem` components.
- Each `ListItem` component is composed of a message and a button.
- When a `ListItem` component button is selected, that `ListItem`'s `UpdateMessage` method changes the list item text and hides the button.

`MessageUpdateInvokeHelper.cs`:

```
using System;
using Microsoft.JSInterop;

public class MessageUpdateInvokeHelper
{
    private Action action;

    public MessageUpdateInvokeHelper(Action action)
    {
        this.action = action;
    }

    [JSInvokable("{APP ASSEMBLY}")]
    public void UpdateMessageCaller()
    {
        action.Invoke();
    }
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

In the client-side JavaScript:

```
window.updateMessageCallerJS = (dotnetHelper) => {
    dotnetHelper.invokeMethodAsync('{APP ASSEMBLY}', 'UpdateMessageCaller');
    dotnetHelper.dispose();
}
```

The placeholder `{APP ASSEMBLY}` is the app's app assembly name (for example, `BlazorSample`).

`Shared/ListItem.razor`:

```
@inject IJSRuntime JsRuntime

<li>
    @message
    <button @onclick="InteropCall" style="display:@display">InteropCall</button>
</li>

@code {
    private string message = "Select one of these list item buttons.";
    private string display = "inline-block";
    private MessageUpdateInvokeHelper messageUpdateInvokeHelper;

    protected override void OnInitialized()
    {
        messageUpdateInvokeHelper = new MessageUpdateInvokeHelper(UpdateMessage);
    }

    protected async Task InteropCall()
    {
        await JsRuntime.InvokeVoidAsync("updateMessageCallerJS",
            DotNetObjectReference.Create(messageUpdateInvokeHelper));
    }

    private void UpdateMessage()
    {
        message = "UpdateMessage Called!";
        display = "none";
        StateHasChanged();
    }
}
```

`Pages/JSInteropExample.razor`:

```
@page "/JSInteropExample"

<h1>List of components</h1>

<ul>
    <ListItem />
    <ListItem />
    <ListItem />
    <ListItem />
</ul>
```

Share interop code in a class library

JS interop code can be included in a class library, which allows you to share the code in a NuGet package.

The class library handles embedding JavaScript resources in the built assembly. The JavaScript files are placed in the `wwwroot` folder. The tooling takes care of embedding the resources when the library is built.

The built NuGet package is referenced in the app's project file the same way that any NuGet package is referenced. After the package is restored, app code can call into JavaScript as if it were C#.

For more information, see ASP.NET Core Razor components class libraries.

Avoid circular object references

Objects that contain circular references can't be serialized on the client for either:

- .NET method calls.
- JavaScript method calls from C# when the return type has circular references.

For more information, see the following issues:

- Circular references are not supported, take two (dotnet/aspnetcore #20525)
- Proposal: Add mechanism to handle circular references when serializing (dotnet/runtime #30820)

Additional resources

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- `InteropComponent.razor` example (dotnet/AspNetCore GitHub repository, 3.1 release branch)
- Perform large data transfers in Blazor Server apps

Call a web API from ASP.NET Core Blazor

9/22/2020 • 8 minutes to read • Edit Online

By Luke Latham, Daniel Roth, and Juan De la Cruz

> **NOTE**
> This topic applies to Blazor WebAssembly. Blazor Server apps call web APIs using HttpClient instances, typically created using IHttpClientFactory. For guidance that applies to Blazor Server, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

Blazor WebAssembly apps call web APIs using a preconfigured HttpClient service. Compose requests, which can include JavaScript Fetch API options, using Blazor JSON helpers or with HttpRequestMessage. The HttpClient service in Blazor WebAssembly apps is focused on making requests back to the server of origin. The guidance in this topic only pertains to Blazor WebAssembly apps.

View or download sample code (how to download): Select the `BlazorWebAssemblySample` app.

See the following components in the `BlazorWebAssemblySample` sample app:

- Call Web API (`Pages/CallWebAPI.razor`)
- HTTP Request Tester (`Components/HTTPRequestTester.razor`)

Packages

Reference the `System.Net.Http.Json` NuGet package in the project file.

Add the HttpClient service

In `Program.Main`, add an HttpClient service if it doesn't already exist:

```
builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
    });
```

HttpClient and JSON helpers

In a Blazor WebAssembly app, `HttpClient` is available as a preconfigured service for making requests back to the origin server.

A Blazor Server app doesn't include an HttpClient service by default. Provide an HttpClient to the app using the `HttpClient` factory infrastructure.

HttpClient and JSON helpers are also used to call third-party web API endpoints. HttpClient is implemented using the browser Fetch API and is subject to its limitations, including enforcement of the same origin policy.

The client's base address is set to the originating server's address. Inject an HttpClient instance using the `@inject` directive:

```
@using System.Net.Http
@inject HttpClient Http
```

In the following examples, a Todo web API processes create, read, update, and delete (CRUD) operations. The examples are based on a `TodoItem` class that stores the:

- ID (`Id`, `long`): Unique ID of the item.
- Name (`Name`, `string`): Name of the item.
- Status (`IsComplete`, `bool`): Indication if the Todo item is finished.

```
private class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

JSON helper methods send requests to a URI (a web API in the following examples) and process the response:

- GetFromJsonAsync: Sends an HTTP GET request and parses the JSON response body to create an object.

  In the following code, the `todoItems` are displayed by the component. The `GetTodoItems` method is triggered when the component is finished rendering (`OnInitializedAsync`). See the sample app for a complete example.

  ```
  @using System.Net.Http
  @inject HttpClient Http

  @code {
      private TodoItem[] todoItems;

      protected override async Task OnInitializedAsync() =>
          todoItems = await Http.GetFromJsonAsync<TodoItem[]>("api/TodoItems");
  }
  ```

- PostAsJsonAsync: Sends an HTTP POST request, including JSON-encoded content, and parses the JSON response body to create an object.

  In the following code, `newItemName` is provided by a bound element of the component. The `AddItem` method is triggered by selecting a `<button>` element. See the sample app for a complete example.

  ```
  @using System.Net.Http
  @inject HttpClient Http

  <input @bind="newItemName" placeholder="New Todo Item" />
  <button @onclick="@AddItem">Add</button>

  @code {
      private string newItemName;

      private async Task AddItem()
      {
          var addItem = new TodoItem { Name = newItemName, IsComplete = false };
          await Http.PostAsJsonAsync("api/TodoItems", addItem);
      }
  }
  ```

  Calls to PostAsJsonAsync return an HttpResponseMessage. To deserialize the JSON content from the

response message, use the `ReadFromJsonAsync<T>` extension method:

```
var content = response.Content.ReadFromJsonAsync<WeatherForecast>();
```

- PutAsJsonAsync: Sends an HTTP PUT request, including JSON-encoded content.

  In the following code, `editItem` values for `Name` and `IsCompleted` are provided by bound elements of the component. The item's `Id` is set when the item is selected in another part of the UI and `EditItem` is called. The `SaveItem` method is triggered by selecting the Save `<button>` element. See the sample app for a complete example.

```
@using System.Net.Http
@inject HttpClient Http

<input type="checkbox" @bind="editItem.IsComplete" />
<input @bind="editItem.Name" />
<button @onclick="@SaveItem">Save</button>

@code {
    private TodoItem editItem = new TodoItem();

    private void EditItem(long id)
    {
        editItem = todoItems.Single(i => i.Id == id);
    }

    private async Task SaveItem() =>
        await Http.PutAsJsonAsync($"api/TodoItems/{editItem.Id}, editItem);
}
```

  Calls to PutAsJsonAsync return an HttpResponseMessage. To deserialize the JSON content from the response message, use the ReadFromJsonAsync extension method:

```
var content = response.Content.ReadFromJsonAsync<WeatherForecast>();
```

System.Net.Http includes additional extension methods for sending HTTP requests and receiving HTTP responses. HttpClient.DeleteAsync is used to send an HTTP DELETE request to a web API.

In the following code, the Delete `<button>` element calls the `DeleteItem` method. The bound `<input>` element supplies the `id` of the item to delete. See the sample app for a complete example.

```
@using System.Net.Http
@inject HttpClient Http

<input @bind="id" />
<button @onclick="@DeleteItem">Delete</button>

@code {
    private long id;

    private async Task DeleteItem() =>
        await Http.DeleteAsync($"api/TodoItems/{id}");
}
```

Named HttpClient with IHttpClientFactory

IHttpClientFactory services and the configuration of a named HttpClient are supported.

Reference the `Microsoft.Extensions.Http` NuGet package in the project file.

`Program.Main` (`Program.cs`):

```
builder.Services.AddHttpClient("ServerAPI", client =>
    client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));
```

`FetchData` component (`Pages/FetchData.razor`):

```
@inject IHttpClientFactory ClientFactory

...

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        var client = ClientFactory.CreateClient("ServerAPI");

        forecasts = await client.GetFromJsonAsync<WeatherForecast[]>(
            "WeatherForecast");
    }
}
```

Typed HttpClient

Typed HttpClient uses one or more of the app's HttpClient instances, default or named, to return data from one or more web API endpoints.

`WeatherForecastClient.cs` :

```
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;

public class WeatherForecastClient
{
    private readonly HttpClient client;

    public WeatherForecastClient(HttpClient client)
    {
        this.client = client;
    }

    public async Task<WeatherForecast[]> GetForecastAsync()
    {
        var forecasts = new WeatherForecast[0];

        try
        {
            forecasts = await client.GetFromJsonAsync<WeatherForecast[]>(
                "WeatherForecast");
        }
        catch
        {
            ...
        }

        return forecasts;
    }
}
```

`Program.Main` (`Program.cs`):

```
builder.Services.AddHttpClient<WeatherForecastClient>(client =>
    client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));
```

Components inject the typed HttpClient to call the web API.

`FetchData` component (`Pages/FetchData.razor`):

```
@inject WeatherForecastClient Client

...

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await Client.GetForecastAsync();
    }
}
```

`HttpClient` and `HttpRequestMessage` with Fetch API request options

When running on WebAssembly in a Blazor WebAssembly app, `HttpClient` (API documentation) and HttpRequestMessage can be used to customize requests. For example, you can specify the HTTP method and request headers. The following component makes a `POST` request to a To Do List API endpoint on the server and shows the response body:

```
@page "/todorequest"
@using System.Net.Http
@using System.Net.Http.Headers
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject HttpClient Http
@inject IAccessTokenProvider TokenProvider

<h1>ToDo Request</h1>

<button @onclick="PostRequest">Submit POST request</button>

<p>Response body returned by the server:</p>

<p>@responseBody</p>

@code {
    private string responseBody;

    private async Task PostRequest()
    {
        var requestMessage = new HttpRequestMessage()
        {
            Method = new HttpMethod("POST"),
            RequestUri = new Uri("https://localhost:10000/api/TodoItems"),
            Content =
                JsonContent.Create(new TodoItem
                {
                    Name = "My New Todo Item",
                    IsComplete = false
                })
        };

        var tokenResult = await TokenProvider.RequestAccessToken();

        if (tokenResult.TryGetToken(out var token))
        {
            requestMessage.Headers.Authorization =
                new AuthenticationHeaderValue("Bearer", token.Value);

            requestMessage.Content.Headers.TryAddWithoutValidation(
                "x-custom-header", "value");

            var response = await Http.SendAsync(requestMessage);
            var responseStatusCode = response.StatusCode;

            responseBody = await response.Content.ReadAsStringAsync();
        }
    }

    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

.NET WebAssembly's implementation of HttpClient uses WindowOrWorkerGlobalScope.fetch(). Fetch allows configuring several request-specific options.

HTTP fetch request options can be configured with HttpRequestMessage extension methods shown in the following table.

EXTENSION METHOD	FETCH REQUEST PROPERTY
SetBrowserRequestCredentials	`credentials`
SetBrowserRequestCache	`cache`
SetBrowserRequestMode	`mode`
SetBrowserRequestIntegrity	`integrity`

You can set additional options using the more generic SetBrowserRequestOption extension method.

The HTTP response is typically buffered in a Blazor WebAssembly app to enable support for sync reads on the response content. To enable support for response streaming, use the SetBrowserResponseStreamingEnabled extension method on the request.

To include credentials in a cross-origin request, use the SetBrowserRequestCredentials extension method:

```
requestMessage.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
```

For more information on Fetch API options, see MDN web docs: WindowOrWorkerGlobalScope.fetch():Parameters.

Handle errors

When errors occur while interacting with a web API, they can be handled by developer code. For example, GetFromJsonAsync expects a JSON response from the server API with a `Content-Type` of `application/json`. If the response isn't in JSON format, content validation throws a NotSupportedException.

In the following example, the URI endpoint for the weather forecast data request is misspelled. The URI should be to `WeatherForecast` but appears in the call as `WeatherForcast` (missing "e").

The GetFromJsonAsync call expects JSON to be returned, but the server returns HTML for an unhandled exception on the server with a `Content-Type` of `text/html`. The unhandled exception occurs on the server because the path isn't found and middleware can't serve a page or view for the request.

In OnInitializedAsync on the client, NotSupportedException is thrown when the response content is validated as non-JSON. The exception is caught in the `catch` block, where custom logic could log the error or present a friendly error message to the user:

```
protected override async Task OnInitializedAsync()
{
    try
    {
        forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>(
            "WeatherForcast");
    }
    catch (NotSupportedException exception)
    {
        ...
    }
}
```

> **NOTE**
>
> The preceding example is for demonstration purposes. A web API server app can be configured to return JSON even when an endpoint doesn't exist or an unhandled exception on the server occurs.

For more information, see Handle errors in ASP.NET Core Blazor apps.

Cross-origin resource sharing (CORS)

Browser security prevents a webpage from making requests to a different domain than the one that served the webpage. This restriction is called the *same-origin policy*. The same-origin policy prevents a malicious site from reading sensitive data from another site. To make requests from the browser to an endpoint with a different origin, the *endpoint* must enable cross-origin resource sharing (CORS).

The Blazor WebAssembly sample app (BlazorWebAssemblySample) demonstrates the use of CORS in the Call Web API component (`Pages/CallWebAPI.razor`).

For more information on CORS with secure requests in Blazor apps, see ASP.NET Core Blazor WebAssembly additional security scenarios.

For general information on CORS with ASP.NET Core apps, see Enable Cross-Origin Requests (CORS) in ASP.NET Core.

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios: Includes coverage on using HttpClient to make secure web API requests.
- Make HTTP requests using IHttpClientFactory in ASP.NET Core
- Enforce HTTPS in ASP.NET Core
- Kestrel HTTPS endpoint configuration
- Cross Origin Resource Sharing (CORS) at W3C

ASP.NET Core Blazor authentication and authorization

9/22/2020 • 10 minutes to read • Edit Online

By Steve Sanderson and Luke Latham

ASP.NET Core supports the configuration and management of security in Blazor apps.

Security scenarios differ between Blazor Server and Blazor WebAssembly apps. Because Blazor Server apps run on the server, authorization checks are able to determine:

- The UI options presented to a user (for example, which menu entries are available to a user).
- Access rules for areas of the app and components.

Blazor WebAssembly apps run on the client. Authorization is *only* used to determine which UI options to show. Since client-side checks can be modified or bypassed by a user, a Blazor WebAssembly app can't enforce authorization access rules.

Razor Pages authorization conventions don't apply to routable Razor components. If a non-routable Razor component is embedded in a page, the page's authorization conventions indirectly affect the Razor component along with the rest of the page's content.

> **NOTE**
> SignInManager<TUser> and UserManager<TUser> aren't supported in Razor components.

Authentication

Blazor uses the existing ASP.NET Core authentication mechanisms to establish the user's identity. The exact mechanism depends on how the Blazor app is hosted, Blazor WebAssembly or Blazor Server.

Blazor WebAssembly authentication

In Blazor WebAssembly apps, authentication checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks or native apps for any operating system.

Add the following:

- A package reference for `Microsoft.AspNetCore.Components.Authorization` to the app's project file.
- The `Microsoft.AspNetCore.Components.Authorization` namespace to the app's `_Imports.razor` file.

To handle authentication, use of a built-in or custom AuthenticationStateProvider service is covered in the following sections.

For more information on creating apps and configuration, see Secure ASP.NET Core Blazor WebAssembly.

Blazor Server authentication

Blazor Server apps operate over a real-time connection that's created using SignalR. Authentication in SignalR-based apps is handled when the connection is established. Authentication can be based on a cookie or some other bearer token.

The built-in AuthenticationStateProvider service for Blazor Server apps obtains authentication state data from

ASP.NET Core's `HttpContext.User`. This is how authentication state integrates with existing ASP.NET Core authentication mechanisms.

For more information on creating apps and configuration, see Secure ASP.NET Core Blazor Server apps.

AuthenticationStateProvider service

AuthenticationStateProvider is the underlying service used by the AuthorizeView component and CascadingAuthenticationState component to get the authentication state.

You don't typically use AuthenticationStateProvider directly. Use the `AuthorizeView` component or `Task<AuthenticationState>` approaches described later in this article. The main drawback to using AuthenticationStateProvider directly is that the component isn't notified automatically if the underlying authentication state data changes.

The AuthenticationStateProvider service can provide the current user's ClaimsPrincipal data, as shown in the following example:

```
@page "/"
@using System.Security.Claims
@using Microsoft.AspNetCore.Components.Authorization
@inject AuthenticationStateProvider AuthenticationStateProvider

<h3>ClaimsPrincipal Data</h3>

<button @onclick="GetClaimsPrincipalData">Get ClaimsPrincipal Data</button>

<p>@_authMessage</p>

@if (_claims.Count() > 0)
{
    <ul>
        @foreach (var claim in _claims)
        {
            <li>@claim.Type: @claim.Value</li>
        }
    </ul>
}

<p>@_surnameMessage</p>

@code {
    private string _authMessage;
    private string _surnameMessage;
    private IEnumerable<Claim> _claims = Enumerable.Empty<Claim>();

    private async Task GetClaimsPrincipalData()
    {
        var authState = await AuthenticationStateProvider.GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity.IsAuthenticated)
        {
            _authMessage = $"{user.Identity.Name} is authenticated.";
            _claims = user.Claims;
            _surnameMessage =
                $"Surname: {user.FindFirst(c => c.Type == ClaimTypes.Surname)?.Value}";
        }
        else
        {
            _authMessage = "The user is NOT authenticated.";
        }
    }
}
```

If `user.Identity.IsAuthenticated` is `true` and because the user is a ClaimsPrincipal, claims can be enumerated and membership in roles evaluated.

For more information on dependency injection (DI) and services, see ASP.NET Core Blazor dependency injection and Dependency injection in ASP.NET Core.

Implement a custom AuthenticationStateProvider

If the app requires a custom provider, implement AuthenticationStateProvider and override `GetAuthenticationStateAsync`:

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
    {
        var identity = new ClaimsIdentity(new[]
        {
            new Claim(ClaimTypes.Name, "mrfibuli"),
        }, "Fake authentication type");

        var user = new ClaimsPrincipal(identity);

        return Task.FromResult(new AuthenticationState(user));
    }
}
```

In a Blazor WebAssembly app, the `CustomAuthStateProvider` service is registered in `Main` of `Program.cs`:

```
using Microsoft.AspNetCore.Components.Authorization;

...

builder.Services.AddScoped<AuthenticationStateProvider, CustomAuthStateProvider>();
```

In a Blazor Server app, the `CustomAuthStateProvider` service is registered in `Startup.ConfigureServices`:

```
using Microsoft.AspNetCore.Components.Authorization;

...

services.AddScoped<AuthenticationStateProvider, CustomAuthStateProvider>();
```

Using the `CustomAuthStateProvider` in the preceding example, all users are authenticated with the username `mrfibuli`.

Expose the authentication state as a cascading parameter

If authentication state data is required for procedural logic, such as when performing an action triggered by the user, obtain the authentication state data by defining a cascading parameter of type `Task<` AuthenticationState `>`:

```
@page "/"

<button @onclick="LogUsername">Log username</button>

<p>@_authMessage</p>

@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private string _authMessage;

    private async Task LogUsername()
    {
        var authState = await authenticationStateTask;
        var user = authState.User;

        if (user.Identity.IsAuthenticated)
        {
            _authMessage = $"{user.Identity.Name} is authenticated.";
        }
        else
        {
            _authMessage = "The user is NOT authenticated.";
        }
    }
}
```

If `user.Identity.IsAuthenticated` is `true`, claims can be enumerated and membership in roles evaluated.

Set up the `Task<` AuthenticationState `>` cascading parameter using the AuthorizeRouteView and CascadingAuthenticationState components in the `App` component (`App.razor`):

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)" />
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

In a Blazor WebAssembly App, add services for options and authorization to `Program.Main`:

```
builder.Services.AddOptions();
builder.Services.AddAuthorizationCore();
```

In a Blazor Server app, services for options and authorization are already present, so no further action is required.

Authorization

After a user is authenticated, *authorization* rules are applied to control what the user can do.

Access is typically granted or denied based on whether:

- A user is authenticated (signed in).

- A user is in a *role*.
- A user has a *claim*.
- A *policy* is satisfied.

Each of these concepts is the same as in an ASP.NET Core MVC or Razor Pages app. For more information on ASP.NET Core security, see the articles under ASP.NET Core Security and Identity.

AuthorizeView component

The AuthorizeView component selectively displays UI depending on whether the user is authorized to see it. This approach is useful when you only need to *display* data for the user and don't need to use the user's identity in procedural logic.

The component exposes a `context` variable of type AuthenticationState, which you can use to access information about the signed-in user:

```
<AuthorizeView>
    <h1>Hello, @context.User.Identity.Name!</h1>
    <p>You can only see this content if you're authenticated.</p>
</AuthorizeView>
```

You can also supply different content for display if the user isn't authenticated:

```
<AuthorizeView>
    <Authorized>
        <h1>Hello, @context.User.Identity.Name!</h1>
        <p>You can only see this content if you're authenticated.</p>
    </Authorized>
    <NotAuthorized>
        <h1>Authentication Failure!</h1>
        <p>You're not signed in.</p>
    </NotAuthorized>
</AuthorizeView>
```

The AuthorizeView component can be used in the `NavMenu` component (`Shared/NavMenu.razor`) to display a list item (`<li>...</li>`) for a `NavLink` component (NavLink), but note that this approach only removes the list item from the rendered output. It doesn't prevent the user from navigating to the component.

The content of `<Authorized>` and `<NotAuthorized>` tags can include arbitrary items, such as other interactive components.

Authorization conditions, such as roles or policies that control UI options or access, are covered in the Authorization section.

If authorization conditions aren't specified, AuthorizeView uses a default policy and treats:

- Authenticated (signed-in) users as authorized.
- Unauthenticated (signed-out) users as unauthorized.

Role-based and policy-based authorization

The AuthorizeView component supports *role-based* or *policy-based* authorization.

For role-based authorization, use the Roles parameter:

```
<AuthorizeView Roles="admin, superuser">
    <p>You can only see this if you're an admin or superuser.</p>
</AuthorizeView>
```

For more information, see Role-based authorization in ASP.NET Core.

For policy-based authorization, use the Policy parameter:

```
<AuthorizeView Policy="content-editor">
    <p>You can only see this if you satisfy the "content-editor" policy.</p>
</AuthorizeView>
```

Claims-based authorization is a special case of policy-based authorization. For example, you can define a policy that requires users to have a certain claim. For more information, see Policy-based authorization in ASP.NET Core.

These APIs can be used in either Blazor Server or Blazor WebAssembly apps.

If neither Roles nor Policy is specified, AuthorizeView uses the default policy.

Content displayed during asynchronous authentication

Blazor allows for authentication state to be determined *asynchronously*. The primary scenario for this approach is in Blazor WebAssembly apps that make a request to an external endpoint for authentication.

While authentication is in progress, AuthorizeView displays no content by default. To display content while authentication occurs, use the `<Authorizing>` tag:

```
<AuthorizeView>
    <Authorized>
        <h1>Hello, @context.User.Identity.Name!</h1>
        <p>You can only see this content if you're authenticated.</p>
    </Authorized>
    <Authorizing>
        <h1>Authentication in progress</h1>
        <p>You can only see this content while authentication is in progress.</p>
    </Authorizing>
</AuthorizeView>
```

This approach isn't normally applicable to Blazor Server apps. Blazor Server apps know the authentication state as soon as the state is established. Authorizing content can be provided in a Blazor Server app's AuthorizeView component, but the content is never displayed.

[Authorize] attribute

The `[Authorize]` attribute can be used in Razor components:

```
@page "/"
@attribute [Authorize]

You can only see this if you're signed in.
```

IMPORTANT

Only use `[Authorize]` on `@page` components reached via the Blazor Router. Authorization is only performed as an aspect of routing and *not* for child components rendered within a page. To authorize the display of specific parts within a page, use AuthorizeView instead.

The `[Authorize]` attribute also supports role-based or policy-based authorization. For role-based authorization, use the Roles parameter:

```
@page "/"
@attribute [Authorize(Roles = "admin, superuser")]

<p>You can only see this if you're in the 'admin' or 'superuser' role.</p>
```

For policy-based authorization, use the Policy parameter:

```
@page "/"
@attribute [Authorize(Policy = "content-editor")]

<p>You can only see this if you satisfy the 'content-editor' policy.</p>
```

If neither Roles nor Policy is specified, `[Authorize]` uses the default policy, which by default is to treat:

- Authenticated (signed-in) users as authorized.
- Unauthenticated (signed-out) users as unauthorized.

Customize unauthorized content with the Router component

The Router component, in conjunction with the AuthorizeRouteView component, allows the app to specify custom content if:

- Content isn't found.
- The user fails an `[Authorize]` condition applied to the component. The `[Authorize]` attribute is covered in the `[Authorize]` attribute section.
- Asynchronous authentication is in progress.

In the default Blazor Server project template, the `App` component (`App.razor`) demonstrates how to set custom content:

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    <h1>Sorry</h1>
                    <p>You're not authorized to reach this page.</p>
                    <p>You may need to log in as a different user.</p>
                </NotAuthorized>
                <Authorizing>
                    <h1>Authentication in progress</h1>
                    <p>Only visible while authentication is in progress.</p>
                </Authorizing>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <h1>Sorry</h1>
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

The content of `<NotFound>`, `<NotAuthorized>`, and `<Authorizing>` tags can include arbitrary items, such as other interactive components.

If the `<NotAuthorized>` tag isn't specified, the AuthorizeRouteView uses the following fallback message:

```
Not authorized.
```

Notification about authentication state changes

If the app determines that the underlying authentication state data has changed (for example, because the user signed out or another user has changed their roles), a custom `AuthenticationStateProvider` can optionally invoke the method NotifyAuthenticationStateChanged on the AuthenticationStateProvider base class. This notifies consumers of the authentication state data (for example, AuthorizeView) to rerender using the new data.

Procedural logic

If the app is required to check authorization rules as part of procedural logic, use a cascaded parameter of type `Task<` AuthenticationState `>` to obtain the user's ClaimsPrincipal. `Task<` AuthenticationState `>` can be combined with other services, such as `IAuthorizationService`, to evaluate policies.

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<button @onclick="@DoSomething">Do something important</button>

@code {
    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private async Task DoSomething()
    {
        var user = (await authenticationStateTask).User;

        if (user.Identity.IsAuthenticated)
        {
            // Perform an action only available to authenticated (signed-in) users.
        }

        if (user.IsInRole("admin"))
        {
            // Perform an action only available to users in the 'admin' role.
        }

        if ((await AuthorizationService.AuthorizeAsync(user, "content-editor"))
            .Succeeded)
        {
            // Perform an action only available to users satisfying the
            // 'content-editor' policy.
        }
    }
}
```

NOTE

In a Blazor WebAssembly app component, add the Microsoft.AspNetCore.Authorization and Microsoft.AspNetCore.Components.Authorization namespaces:

```
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
```

These namespaces can be provided globally by adding them to the app's `_Imports.razor` file.

Troubleshoot errors

Common errors:

- **Authorization requires a cascading parameter of type `Task\<AuthenticationState>`. Consider using `CascadingAuthenticationState` to supply this.**
- **`null` value is received for `authenticationStateTask`**

It's likely that the project wasn't created using a Blazor Server template with authentication enabled. Wrap a `<CascadingAuthenticationState>` around some part of the UI tree, for example in the `App` component (`App.razor`) as follows:

```
<CascadingAuthenticationState>
    <Router AppAssembly="typeof(Startup).Assembly">
        ...
    </Router>
</CascadingAuthenticationState>
```

The CascadingAuthenticationState supplies the `Task<` AuthenticationState `>` cascading parameter, which in turn it receives from the underlying AuthenticationStateProvider DI service.

Additional resources

- Overview of ASP.NET Core Security
- Configure Windows Authentication in ASP.NET Core
- Awesome Blazor: Authentication community sample links

Secure ASP.NET Core Blazor WebAssembly

9/22/2020 • 5 minutes to read • Edit Online

By Javier Calvarro Nelson

Blazor WebAssembly apps are secured in the same manner as Single Page Applications (SPAs). There are several approaches for authenticating users to SPAs, but the most common and comprehensive approach is to use an implementation based on the OAuth 2.0 protocol, such as OpenID Connect (OIDC).

Authentication library

Blazor WebAssembly supports authenticating and authorizing apps using OIDC via the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` library. The library provides a set of primitives for seamlessly authenticating against ASP.NET Core backends. The library integrates ASP.NET Core Identity with API authorization support built on top of Identity Server. The library can authenticate against any third-party Identity Provider (IP) that supports OIDC, which are called OpenID Providers (OP).

The authentication support in Blazor WebAssembly is built on top of the `oidc-client.js` library, which is used to handle the underlying authentication protocol details.

Other options for authenticating SPAs exist, such as the use of SameSite cookies. However, the engineering design of Blazor WebAssembly is settled on OAuth and OIDC as the best option for authentication in Blazor WebAssembly apps. Token-based authentication based on JSON Web Tokens (JWTs) was chosen over cookie-based authentication for functional and security reasons:

- Using a token-based protocol offers a smaller attack surface area, as the tokens aren't sent in all requests.
- Server endpoints don't require protection against Cross-Site Request Forgery (CSRF) because the tokens are sent explicitly. This allows you to host Blazor WebAssembly apps alongside MVC or Razor pages apps.
- Tokens have narrower permissions than cookies. For example, tokens can't be used to manage the user account or change a user's password unless such functionality is explicitly implemented.
- Tokens have a short lifetime, one hour by default, which limits the attack window. Tokens can also be revoked at any time.
- Self-contained JWTs offer guarantees to the client and server about the authentication process. For example, a client has the means to detect and validate that the tokens it receives are legitimate and were emitted as part of a given authentication process. If a third party attempts to switch a token in the middle of the authentication process, the client can detect the switched token and avoid using it.
- Tokens with OAuth and OIDC don't rely on the user agent behaving correctly to ensure that the app is secure.
- Token-based protocols, such as OAuth and OIDC, allow for authenticating and authorizing hosted and standalone apps with the same set of security characteristics.

Authentication process with OIDC

The `Microsoft.AspNetCore.Components.WebAssembly.Authentication` library offers several primitives to implement authentication and authorization using OIDC. In broad terms, authentication works as follows:

- When an anonymous user selects the login button or requests a page with the `[Authorize]` attribute applied, the user is redirected to the app's login page (`/authentication/login`).
- In the login page, the authentication library prepares for a redirect to the authorization endpoint. The authorization endpoint is outside of the Blazor WebAssembly app and can be hosted at a separate origin. The endpoint is responsible for determining whether the user is authenticated and for issuing one or more tokens

in response. The authentication library provides a login callback to receive the authentication response.
    - If the user isn't authenticated, the user is redirected to the underlying authentication system, which is usually ASP.NET Core Identity.
    - If the user was already authenticated, the authorization endpoint generates the appropriate tokens and redirects the browser back to the login callback endpoint (`/authentication/login-callback`).
- When the Blazor WebAssembly app loads the login callback endpoint (`/authentication/login-callback`), the authentication response is processed.
    - If the authentication process completes successfully, the user is authenticated and optionally sent back to the original protected URL that the user requested.
    - If the authentication process fails for any reason, the user is sent to the login failed page (`/authentication/login-failed`), and an error is displayed.

`Authentication` component

The `Authentication` component (`Pages/Authentication.razor`) handles remote authentication operations and permits the app to:

- Configure app routes for authentication states.
- Set UI content for authentication states.
- Manage authentication state.

Authentication actions, such as registering or signing in a user, are passed to the Blazor framework's RemoteAuthenticatorViewCore<TAuthenticationState> component, which persists and controls state across authentication operations.

For more information and examples, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Authorization

In Blazor WebAssembly apps, authorization checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks or native apps for any operating system.

Always perform authorization checks on the server within any API endpoints accessed by your client-side app.

Require authorization for the entire app

Apply the `[Authorize]` attribute (API documentation) to each Razor component of the app using one of the following approaches:

- Use the `@attribute` directive in the `_Imports.razor` file:

```
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
```

- Add the attribute to each Razor component in the `Pages` folder.

> **NOTE**
>
> Setting an AuthorizationOptions.FallbackPolicy to a policy with RequireAuthenticatedUser is **not** supported.

Refresh tokens

Refresh tokens can't be secured client-side in Blazor WebAssembly apps. Therefore, refresh tokens shouldn't be sent to the app for direct use.

Refresh tokens can be maintained and used by the server-side app in a Hosted Blazor WebAssembly solution to access third-party APIs. For more information, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Establish claims for users

Apps often require claims for users based on a web API call to a server. For example, claims are frequently used to establish authorization in an app. In these scenarios, the app requests an access token to access the service and uses the token to obtain the user data for the claims. For examples, see the following resources:

- Additional scenarios: Customize the user
- ASP.NET Core Blazor WebAssembly with Azure Active Directory groups and roles

Implementation guidance

Articles under this *Overview* provide information on authenticating users in Blazor WebAssembly apps against specific providers.

Standalone Blazor WebAssembly apps:

- General guidance for OIDC providers and the WebAssembly Authentication Library
- Microsoft Accounts
- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C

Hosted Blazor WebAssembly apps:

- Azure Active Directory (AAD)
- Azure Active Directory (AAD) B2C
- Identity Server

For further guidance on configuration, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Secure an ASP.NET Core Blazor WebAssembly standalone app with the Authentication library

9/22/2020 • 7 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

For Azure Active Directory (AAD) and Azure Active Directory B2C (AAD B2C), don't follow the guidance in this topic. See the AAD and AAD B2C topics in this table of contents node.

To create a standalone Blazor WebAssembly app that uses `Microsoft.AspNetCore.Components.WebAssembly.Authentication` library, follow the guidance for your choice of tooling.

- Visual Studio
- Visual Studio Code / .NET Core CLI
- Visual Studio for Mac

To create a new Blazor WebAssembly project with an authentication mechanism:

1. After choosing the **Blazor WebAssembly App** template in the **Create a new ASP.NET Core Web Application** dialog, select **Change** under **Authentication**.
2. Select **Individual User Accounts** with the **Store user accounts in-app** option to store users within the app using ASP.NET Core's Identity system.

Authentication package

When an app is created to use Individual User Accounts, the app automatically receives a package reference for the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package in the app's project file. The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the package to the app's project file:

```
<PackageReference
  Include="Microsoft.AspNetCore.Components.WebAssembly.Authentication"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

Authentication service support

Support for authenticating users is registered in the service container with the AddOidcAuthentication extension method provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package. This method sets up the services required for the app to interact with the Identity Provider (IP).

`Program.cs`:

```
builder.Services.AddOidcAuthentication(options =>
{
    builder.Configuration.Bind("Local", options.ProviderOptions);
});
```

Configuration is supplied by the `wwwroot/appsettings.json` file:

```
{
    "Local": {
        "Authority": "{AUTHORITY}",
        "ClientId": "{CLIENT ID}"
    }
}
```

Authentication support for standalone apps is offered using OpenID Connect (OIDC). The AddOidcAuthentication method accepts a callback to configure the parameters required to authenticate an app using OIDC. The values required for configuring the app can be obtained from the OIDC-compliant IP. Obtain the values when you register the app, which typically occurs in their online portal.

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default token scopes of the OidcProviderOptions:

```
builder.Services.AddOidcAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultScopes.Add("{SCOPE URI}");
});
```

NOTE

If the Azure portal provides the scope URI for the app and the app throws an unhandled exception when it receives a *401 Unauthorized* response from the API, try using a scope URI that doesn't include the scheme and host. For example, the Azure portal may provide one of the following scope URI formats:

- `https://{TENANT}.onmicrosoft.com/{API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`
- `api://{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`

Try supplying the scope URI without the scheme and host:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}");
```

For example:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "41451fa7-82d9-4673-8fa5-69eff5a761fd/API.Access");
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```
<script src="_content/Microsoft.AspNetCore.Components.WebAssembly.Authentication/
    AuthenticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The CascadingAuthenticationState component manages exposing the AuthenticationState to the rest of the app.
- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <p>
                            You are not authorized to access
                            this resource.
                        </p>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

RedirectToLogin component

The `RedirectToLogin` component (`Shared/RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- Preserves the current URL that the user is attempting to access so that they can be returned to that page if authentication is successful.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri));
    }
}
```

LoginDisplay component

The `LoginDisplay` component (`Shared/LoginDisplay.razor`) is rendered in the `MainLayout` component (`Shared/MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current username.
  - Offers a button to log out of the app.
- For anonymous users, offers the option to log in.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
        <button class="nav-link btn btn-link" @onclick="BeginSignOut">
            Log out
        </button>
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

@code {
    private async Task BeginSignOut(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package.
- Manages performing the appropriate actions at each stage of authentication.

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string Action { get; set; }
}
```

Troubleshoot

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE between any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in incognito or private mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in incognito or private mode. Some browsers require the URL of the app.
    - Microsoft Edge: `-inprivate`
    - Google Chrome: `--incognito --new-window https://localhost:5001`
    - Mozilla Firefox: `-private -url https://localhost:5001`
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE between any change to the app, test user, or provider configuration.

Run the Server app

When testing and troubleshooting a hosted Blazor app, make sure that you're running the app from the `Server` project. For example in Visual Studio, confirm that the Server project is highlighted in **Solution Explorer** before you start the app with any of the following approaches:

- Select the **Run** button.
- Use **Debug** > **Start Debugging** from the menu.
- Press F5.

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client

Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Accounts

9/22/2020 • 8 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

To create a standalone Blazor WebAssembly app that uses Microsoft Accounts with Azure Active Directory (AAD) for authentication:

Create an AAD tenant and web application

Register a AAD app in the **Azure Active Directory** > **App registrations** area of the Azure portal:

1. Provide a **Name** for the app (for example, **Blazor Standalone AAD Microsoft Accounts**).
2. In **Supported account types**, select **Accounts in any organizational directory**.
3. Leave the **Redirect URI** drop down set to **Web** and provide the following redirect URI: `https://localhost:{PORT}/authentication/login-callback`. The default port for an app running on Kestrel is 5001. If the app is run on a different Kestrel port, use the app's port. For IIS Express, the randomly generated port for the app can be found in the app's properties in the **Debug** panel. Since the app doesn't exist at this point and the IIS Express port isn't known, return to this step after the app is created and update the redirect URI. A remark appears later in this topic to remind IIS Express users to update the redirect URI.
4. Disable the **Permissions** > **Grant admin consent to openid and offline_access permissions** check box.
5. Select **Register**.

Record the Application (client) ID (for example, `41451fa7-82d9-4673-8fa5-69eff5a761fd`).

In **Authentication** > **Platform configurations** > **Web**:

1. Confirm the **Redirect URI** of `https://localhost:{PORT}/authentication/login-callback` is present.
2. For **Implicit grant**, select the check boxes for **Access tokens** and **ID tokens**.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button.

Create the app. Replace the placeholders in the following command with the information recorded earlier and execute the following command in a command shell:

```
dotnet new blazorwasm -au SingleOrg --client-id "{CLIENT ID}" --tenant-id "common" -o {APP NAME}
```

PLACEHOLDER	AZURE PORTAL NAME	EXAMPLE
`{APP NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`41451fa7-82d9-4673-8fa5-69eff5a761fd`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the app's name.

> **NOTE**
>
> In the Azure portal, the app's **Authentication** > **Platform configurations** > **Web** > **Redirect URI** is configured for port 5001 for apps that run on the Kestrel server with default settings.
>
> If the app is run on a random IIS Express port, the port for the app can be found in the app's properties in the **Debug** panel.
>
> If the port wasn't configured earlier with the app's known port, return to the app's registration in the Azure portal and update the redirect URI with the correct port.

After creating the app, you should be able to:

- Log into the app using a Microsoft account.
- Request access tokens for Microsoft APIs. For more information, see:
  - Access token scopes
  - Quickstart: Configure an application to expose web APIs.

Authentication package

When an app is created to use Work or School Accounts (`SingleOrg`), the app automatically receives a package reference for the Microsoft Authentication Library (`Microsoft.Authentication.WebAssembly.Msal`). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the package to the app's project file:

```
<PackageReference Include="Microsoft.Authentication.WebAssembly.Msal"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

The `Microsoft.Authentication.WebAssembly.Msal` package transitively adds the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the `Microsoft.Authentication.WebAssembly.Msal` package. This method sets up all of the services required for the app to interact with the Identity Provider (IP).

`Program.cs`:

```
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAd", options.ProviderOptions.Authentication);
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the AAD configuration when you register the app.

Configuration is supplied by the `wwwroot/appsettings.json` file:

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/common",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  }
}
```

Example:

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/common",
    "ClientId": "41451fa7-82d9-4673-8fa5-69eff5a761fd",
    "ValidateAuthority": true
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

NOTE

If the Azure portal provides the scope URI for the app and the app throws an unhandled exception when it receives a *401 Unauthorized* response from the API, try using a scope URI that doesn't include the scheme and host. For example, the Azure portal may provide one of the following scope URI formats:

- `https://{TENANT}.onmicrosoft.com/{API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`
- `api://{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`

Try supplying the scope URI without the scheme and host:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}");
```

For example:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "41451fa7-82d9-4673-8fa5-69eff5a761fd/API.Access");
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```
<script src="_content/Microsoft.Authentication.WebAssembly.Msal/
    AuthenticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The CascadingAuthenticationState component manages exposing the AuthenticationState to the rest of the app.
- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <p>
                            You are not authorized to access
                            this resource.
                        </p>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

RedirectToLogin component

The `RedirectToLogin` component (`Shared/RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- Preserves the current URL that the user is attempting to access so that they can be returned to that page if authentication is successful.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri));
    }
}
```

LoginDisplay component

The `LoginDisplay` component (`Shared/LoginDisplay.razor`) is rendered in the `MainLayout` component (`Shared/MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current username.
  - Offers a button to log out of the app.
- For anonymous users, offers the option to log in.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
        <button class="nav-link btn btn-link" @onclick="BeginLogout">
            Log out
        </button>
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

@code {
    private async Task BeginLogout(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package.
- Manages performing the appropriate actions at each stage of authentication.

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string Action { get; set; }
}
```

Troubleshoot

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser

    - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
    - Make sure that the browser is closed manually or by the IDE between any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in incognito or private mode in Visual Studio:
    - Open **Browse With** dialog box from Visual Studio's **Run** button.
    - Select the **Add** button.
    - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
        - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
        - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
        - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
    - In the **Arguments** field, provide the command-line option that the browser uses to open in incognito or private mode. Some browsers require the URL of the app.
        - Microsoft Edge: `-inprivate`
        - Google Chrome: `--incognito --new-window https://localhost:5001`
        - Mozilla Firefox: `-private -url https://localhost:5001`
    - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
    - Select the **OK** button.
    - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
    - Make sure that the browser is closed by the IDE between any change to the app, test user, or provider configuration.

Run the Server app

When testing and troubleshooting a hosted Blazor app, make sure that you're running the app from the `Server` project. For example in Visual Studio, confirm that the Server project is highlighted in **Solution Explorer** before you start the app with any of the following approaches:

- Select the **Run** button.
- Use **Debug** > **Start Debugging** from the menu.
- Press F5.

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- ASP.NET Core Blazor WebAssembly with Azure Active Directory groups and roles
- Quickstart: Register an application with the Microsoft identity platform
- Quickstart: Configure an application to expose web APIs

Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory

9/22/2020 • 8 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

To create a standalone Blazor WebAssembly app that uses Azure Active Directory (AAD) for authentication:

Create an AAD tenant and web application:

Register a AAD app in the **Azure Active Directory** > **App registrations** area of the Azure portal:

1. Provide a **Name** for the app (for example, **Blazor Standalone AAD**).
2. Choose a **Supported account types**. You may select **Accounts in this organizational directory only** for this experience.
3. Leave the **Redirect URI** drop down set to **Web** and provide the following redirect URI: `https://localhost:{PORT}/authentication/login-callback`. The default port for an app running on Kestrel is 5001. If the app is run on a different Kestrel port, use the app's port. For IIS Express, the randomly generated port for the app can be found in the app's properties in the **Debug** panel. Since the app doesn't exist at this point and the IIS Express port isn't known, return to this step after the app is created and update the redirect URI. A remark appears later in this topic to remind IIS Express users to update the redirect URI.
4. Disable the **Permissions** > **Grant admin consent to openid and offline_access permissions** check box.
5. Select **Register**.

Record the following information:

- Application (client) ID (for example, `41451fa7-82d9-4673-8fa5-69eff5a761fd`)
- Directory (tenant) ID (for example, `e86c78e2-8bb4-4c41-aefd-918e0565a45e`)

In **Authentication** > **Platform configurations** > **Web**:

1. Confirm the **Redirect URI** of `https://localhost:{PORT}/authentication/login-callback` is present.
2. For **Implicit grant**, select the check boxes for **Access tokens** and **ID tokens**.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button.

Create the app in an empty folder. Replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

```
dotnet new blazorwasm -au SingleOrg --client-id "{CLIENT ID}" -o {APP NAME} --tenant-id "{TENANT ID}"
```

PLACEHOLDER	AZURE PORTAL NAME	EXAMPLE
`{APP NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`41451fa7-82d9-4673-8fa5-69eff5a761fd`
`{TENANT ID}`	Directory (tenant) ID	`e86c78e2-8bb4-4c41-aefd-918e0565a45e`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the app's name.

> **NOTE**
>
> In the Azure portal, the app's **Authentication** > **Platform configurations** > **Web** > **Redirect URI** is configured for port 5001 for apps that run on the Kestrel server with default settings.
>
> If the app is run on a random IIS Express port, the port for the app can be found in the app's properties in the **Debug** panel.
>
> If the port wasn't configured earlier with the app's known port, return to the app's registration in the Azure portal and update the redirect URI with the correct port.

After creating the app, you should be able to:

- Log into the app using an AAD user account.
- Request access tokens for Microsoft APIs. For more information, see:
  - Access token scopes
  - Quickstart: Configure an application to expose web APIs.

Authentication package

When an app is created to use Work or School Accounts (`SingleOrg`), the app automatically receives a package reference for the Microsoft Authentication Library (`Microsoft.Authentication.WebAssembly.Msal`). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the package to the app's project file:

```
<PackageReference Include="Microsoft.Authentication.WebAssembly.Msal"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

The `Microsoft.Authentication.WebAssembly.Msal` package transitively adds the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the `Microsoft.Authentication.WebAssembly.Msal` package. This method sets up the services required for the app to interact with the Identity Provider (IP).

`Program.cs`:

```
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAd", options.ProviderOptions.Authentication);
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the AAD configuration when you register the app.

Configuration is supplied by the `wwwroot/appsettings.json` file:

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  }
}
```

Example:

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/e86c78e2-...-918e0565a45e",
    "ClientId": "41451fa7-82d9-4673-8fa5-69eff5a761fd",
    "ValidateAuthority": true
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

NOTE

If the Azure portal provides the scope URI for the app and the app throws an unhandled exception when it receives a *401 Unauthorized* response from the API, try using a scope URI that doesn't include the scheme and host. For example, the Azure portal may provide one of the following scope URI formats:

- `https://{TENANT}.onmicrosoft.com/{API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`
- `api://{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`

Try supplying the scope URI without the scheme and host:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}");
```

For example:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "41451fa7-82d9-4673-8fa5-69eff5a761fd/API.Access");
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```
<script src="_content/Microsoft.Authentication.WebAssembly.Msal/
    AuthenticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The CascadingAuthenticationState component manages exposing the AuthenticationState to the rest of the app.
- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <p>
                            You are not authorized to access
                            this resource.
                        </p>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

RedirectToLogin component

The `RedirectToLogin` component (`Shared/RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- Preserves the current URL that the user is attempting to access so that they can be returned to that page if authentication is successful.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri));
    }
}
```

LoginDisplay component

The `LoginDisplay` component (`Shared/LoginDisplay.razor`) is rendered in the `MainLayout` component (`Shared/MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current username.
  - Offers a button to log out of the app.
- For anonymous users, offers the option to log in.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
        <button class="nav-link btn btn-link" @onclick="BeginLogout">
            Log out
        </button>
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

@code {
    private async Task BeginLogout(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package.
- Manages performing the appropriate actions at each stage of authentication.

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string Action { get; set; }
}
```

Troubleshoot

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser

    - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
    - Make sure that the browser is closed manually or by the IDE between any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in incognito or private mode in Visual Studio:
    - Open **Browse With** dialog box from Visual Studio's **Run** button.
    - Select the **Add** button.
    - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
        - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
        - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
        - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
    - In the **Arguments** field, provide the command-line option that the browser uses to open in incognito or private mode. Some browsers require the URL of the app.
        - Microsoft Edge: `-inprivate`
        - Google Chrome: `--incognito --new-window https://localhost:5001`
        - Mozilla Firefox: `-private -url https://localhost:5001`
    - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
    - Select the **OK** button.
    - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
    - Make sure that the browser is closed by the IDE between any change to the app, test user, or provider configuration.

Run the Server app

When testing and troubleshooting a hosted Blazor app, make sure that you're running the app from the `Server` project. For example in Visual Studio, confirm that the Server project is highlighted in **Solution Explorer** before you start the app with any of the following approaches:

- Select the **Run** button.
- Use **Debug** > **Start Debugging** from the menu.
- Press F5.

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- ASP.NET Core Blazor WebAssembly with Azure Active Directory groups and roles
- Microsoft identity platform and Azure Active Directory with ASP.NET Core
- Microsoft identity platform documentation

Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory B2C

9/22/2020 • 9 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

To create a standalone Blazor WebAssembly app that uses Azure Active Directory (AAD) B2C for authentication:

Follow the guidance in the following topics to create a tenant and register a web app in the Azure Portal:

Create an AAD B2C tenant

Record the following information:

- AAD B2C instance (for example, `https://contoso.b2clogin.com/`, which includes the trailing slash): The instance is the scheme and host of an Azure B2C app registration, which can be found by opening the **Endpoints** window from the **App registrations** page in the Azure portal.
- AAD B2C Primary/Publisher/Tenant domain (for example, `contoso.onmicrosoft.com`): The domain is available as the **Publisher domain** in the **Branding** blade of the Azure portal for the registered app.

Follow the guidance in Tutorial: Register an application in Azure Active Directory B2C again to register an AAD app for the *Client app* and then do the following:

1. In **Azure Active Directory** > **App registrations**, select **New registration**.
2. Provide a **Name** for the app (for example, **Blazor Standalone AAD B2C**).
3. For **Supported account types**, select the multi-tenant option: **Accounts in any organizational directory or any identity provider. For authenticating users with Azure AD B2C.**
4. Leave the **Redirect URI** drop down set to **Web** and provide the following redirect URI: `https://localhost:{PORT}/authentication/login-callback`. The default port for an app running on Kestrel is 5001. If the app is run on a different Kestrel port, use the app's port. For IIS Express, the randomly generated port for the app can be found in the app's properties in the **Debug** panel. Since the app doesn't exist at this point and the IIS Express port isn't known, return to this step after the app is created and update the redirect URI. A remark appears later in this topic to remind IIS Express users to update the redirect URI.
5. Confirm that **Permissions** > **Grant admin consent to openid and offline_access permissions** is enabled.
6. Select **Register**.

Record the Application (client) ID (for example, `41451fa7-82d9-4673-8fa5-69eff5a761fd`).

In **Authentication** > **Platform configurations** > **Web**:

1. Confirm the **Redirect URI** of `https://localhost:{PORT}/authentication/login-callback` is present.
2. For **Implicit grant**, select the check boxes for **Access tokens** and **ID tokens**.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button.

In **Home** > **Azure AD B2C** > **User flows**:

Create a sign-up and sign-in user flow

At a minimum, select the **Application claims** > **Display Name** user attribute to populate the `context.User.Identity.Name` in the `LoginDisplay` component (`Shared/LoginDisplay.razor`).

Record the sign-up and sign-in user flow name created for the app (for example, `B2C_1_signupsignin`).

In an empty folder, replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

```
dotnet new blazorwasm -au IndividualB2C --aad-b2c-instance "{AAD B2C INSTANCE}" --client-id "{CLIENT ID}" --
domain "{TENANT DOMAIN}" -o {APP NAME} -ssp "{SIGN UP OR SIGN IN POLICY}"
```

PLACEHOLDER	AZURE PORTAL NAME	EXAMPLE
`{AAD B2C INSTANCE}`	Instance	`https://contoso.b2clogin.com/`
`{APP NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`41451fa7-82d9-4673-8fa5-69eff5a761fd`
`{SIGN UP OR SIGN IN POLICY}`	Sign-up/sign-in user flow	`B2C_1_signupsignin1`
`{TENANT DOMAIN}`	Primary/Publisher/Tenant domain	`contoso.onmicrosoft.com`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the app's name.

> **NOTE**
>
> In the Azure portal, the app's **Authentication** > **Platform configurations** > **Web** > **Redirect URI** is configured for port 5001 for apps that run on the Kestrel server with default settings.
>
> If the app is run on a random IIS Express port, the port for the app can be found in the app's properties in the **Debug** panel.
>
> If the port wasn't configured earlier with the app's known port, return to the app's registration in the Azure portal and update the redirect URI with the correct port.

After creating the app, you should be able to:

- Log into the app using an AAD user account.
- Request access tokens for Microsoft APIs. For more information, see:
  - Access token scopes
  - Quickstart: Configure an application to expose web APIs.

Authentication package

When an app is created to use an Individual B2C Account (`IndividualB2C`), the app automatically receives a package reference for the Microsoft Authentication Library (`Microsoft.Authentication.WebAssembly.Msal`). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the package to the app's project file:

```
<PackageReference Include="Microsoft.Authentication.WebAssembly.Msal"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

The `Microsoft.Authentication.WebAssembly.Msal` package transitively adds the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the `Microsoft.Authentication.WebAssembly.Msal` package. This method sets up all of the services required for the app to interact with the Identity Provider (IP).

`Program.cs`:

```
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAdB2C", options.ProviderOptions.Authentication);
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the AAD configuration when you register the app.

Configuration is supplied by the `wwwroot/appsettings.json` file:

```
{
  "AzureAdB2C": {
    "Authority": "{AAD B2C INSTANCE}{DOMAIN}/{SIGN UP OR SIGN IN POLICY}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": false
  }
}
```

Example:

```
{
  "AzureAdB2C": {
    "Authority": "https://contoso.b2clogin.com/contoso.onmicrosoft.com/B2C_1_signupsignin1",
    "ClientId": "41451fa7-82d9-4673-8fa5-69eff5a761fd",
    "ValidateAuthority": false
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

NOTE

If the Azure portal provides the scope URI for the app and the app throws an unhandled exception when it receives a *401 Unauthorized* response from the API, try using a scope URI that doesn't include the scheme and host. For example, the Azure portal may provide one of the following scope URI formats:

- `https://{TENANT}.onmicrosoft.com/{API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`
- `api://{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`

Try supplying the scope URI without the scheme and host:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}");
```

For example:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "41451fa7-82d9-4673-8fa5-69eff5a761fd/API.Access");
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```
<script src="_content/Microsoft.Authentication.WebAssembly.Msal/
    AuthenticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The CascadingAuthenticationState component manages exposing the AuthenticationState to the rest of the app.
- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <p>
                            You are not authorized to access
                            this resource.
                        </p>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

RedirectToLogin component

The `RedirectToLogin` component (`Shared/RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- Preserves the current URL that the user is attempting to access so that they can be returned to that page if authentication is successful.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri));
    }
}
```

LoginDisplay component

The `LoginDisplay` component (`Shared/LoginDisplay.razor`) is rendered in the `MainLayout` component (`Shared/MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current username.
  - Offers a button to log out of the app.
- For anonymous users, offers the option to log in.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
        <button class="nav-link btn btn-link" @onclick="BeginLogout">
            Log out
        </button>
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

@code {
    private async Task BeginLogout(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package.
- Manages performing the appropriate actions at each stage of authentication.

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string Action { get; set; }
}
```

Custom user flows

The Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal, NuGet package) doesn't support AAD B2C user flows by default. Create custom user flows in developer code.

For more information on how to build a challenge for a custom user flow, see User flows in Azure Active Directory B2C.

Troubleshoot

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE between any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in incognito or private mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in incognito or private mode. Some browsers require the URL of the app.
    - Microsoft Edge: `-inprivate`
    - Google Chrome: `--incognito --new-window https://localhost:5001`
    - Mozilla Firefox: `-private -url https://localhost:5001`
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.

- To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
- Make sure that the browser is closed by the IDE between any change to the app, test user, or provider configuration.

Run the Server app

When testing and troubleshooting a hosted Blazor app, make sure that you're running the app from the `Server` project. For example in Visual Studio, confirm that the Server project is highlighted in **Solution Explorer** before you start the app with any of the following approaches:

- Select the **Run** button.
- Use **Debug** > **Start Debugging** from the menu.
- Press F5.

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- Cloud authentication with Azure Active Directory B2C in ASP.NET Core
- Tutorial: Create an Azure Active Directory B2C tenant
- Microsoft identity platform documentation

Secure an ASP.NET Core Blazor WebAssembly hosted app with Azure Active Directory

9/22/2020 • 14 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

This article describes how to create a hosted Blazor WebAssembly app that uses Azure Active Directory (AAD) for authentication.

Register apps in AAD and create solution

Create a tenant

Follow the guidance in Quickstart: Set up a tenant to create a tenant in AAD.

Register a server API app

Follow the guidance in Quickstart: Register an application with the Microsoft identity platform and subsequent Azure AAD topics to register an AAD app for the *Server API app* and then do the following:

1. In **Azure Active Directory** > **App registrations**, select **New registration**.
2. Provide a **Name** for the app (for example, **Blazor Server AAD**).
3. Choose a **Supported account types**. You may select **Accounts in this organizational directory only** (single tenant) for this experience.
4. The *Server API app* doesn't require a **Redirect URI** in this scenario, so leave the drop down set to **Web** and don't enter a redirect URI.
5. Disable the **Permissions** > **Grant admin consent to openid and offline_access permissions** check box.
6. Select **Register**.

Record the following information:

- *Server API app* Application (client) ID (for example, `41451fa7-82d9-4673-8fa5-69eff5a761fd`)
- Directory (tenant) ID (for example, `e86c78e2-8bb4-4c41-aefd-918e0565a45e`)
- AAD Primary/Publisher/Tenant domain (for example, `contoso.onmicrosoft.com`): The domain is available as the **Publisher domain** in the **Branding** blade of the Azure portal for the registered app.

In **API permissions**, remove the **Microsoft Graph** > **User.Read** permission, as the app doesn't require sign in or user profile access.

In **Expose an API**:

1. Select **Add a scope**.
2. Select **Save and continue**.
3. Provide a **Scope name** (for example, `API.Access`).
4. Provide an **Admin consent display name** (for example, `Access API`).
5. Provide an **Admin consent description** (for example, `Allows the app to access server app API endpoints.`).
6. Confirm that the **State** is set to **Enabled**.
7. Select **Add scope**.

Record the following information:

- App ID URI (for example, `https://contoso.onmicrosoft.com/41451fa7-82d9-4673-8fa5-69eff5a761fd`, `api://41451fa7-82d9-4673-8fa5-69eff5a761fd`, or the custom value that you provided)
- Scope name (for example, `API.Access`)

The App ID URI might require a special configuration in the client app, which is described in the Access token scopes section later in this topic.

Register a client app

Follow the guidance in Quickstart: Register an application with the Microsoft identity platform and subsequent Azure AAD topics to register a AAD app for the *Client app* and then do the following:

1. In **Azure Active Directory** > **App registrations**, select **New registration**.
2. Provide a **Name** for the app (for example, **Blazor Client AAD**).
3. Choose a **Supported account types**. You may select **Accounts in this organizational directory only** (single tenant) for this experience.
4. Leave the **Redirect URI** drop down set to **Web** and provide the following redirect URI: `https://localhost:{PORT}/authentication/login-callback`. The default port for an app running on Kestrel is 5001. If the app is run on a different Kestrel port, use the app's port. For IIS Express, the randomly generated port for the app can be found in the Server app's properties in the **Debug** panel. Since the app doesn't exist at this point and the IIS Express port isn't known, return to this step after the app is created and update the redirect URI. A remark appears in the Create the app section to remind IIS Express users to update the redirect URI.
5. Disable the **Permissions** > **Grant admin consent to openid and offline_access permissions** check box.
6. Select **Register**.

Record the *Client app* Application (client) ID (for example, `4369008b-21fa-427c-abaa-9b53bf58e538`).

In **Authentication** > **Platform configurations** > **Web**:

1. Confirm the **Redirect URI** of `https://localhost:{PORT}/authentication/login-callback` is present.
2. For **Implicit grant**, select the check boxes for **Access tokens** and **ID tokens**.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button.

In **API permissions**:

1. Confirm that the app has **Microsoft Graph** > **User.Read** permission.
2. Select **Add a permission** followed by **My APIs**.
3. Select the *Server API app* from the **Name** column (for example, **Blazor Server AAD**).
4. Open the **API** list.
5. Enable access to the API (for example, `API.Access`).
6. Select **Add permissions**.
7. Select the **Grant admin consent for {TENANT NAME}** button. Select **Yes** to confirm.

Create the app

In an empty folder, replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

```
dotnet new blazorwasm -au SingleOrg --api-client-id "{SERVER API APP CLIENT ID}" --app-id-uri "{SERVER API APP ID URI}" --client-id "{CLIENT APP CLIENT ID}" --default-scope "{DEFAULT SCOPE}" --domain "{TENANT DOMAIN}" -ho -o {APP NAME} --tenant-id "{TENANT ID}"
```

PLACEHOLDER	AZURE PORTAL NAME	EXAMPLE
`{APP NAME}`	—	`BlazorSample`
`{CLIENT APP CLIENT ID}`	Application (client) ID for the *Client app*	`4369008b-21fa-427c-abaa-9b53bf58e538`
`{DEFAULT SCOPE}`	Scope name	`API.Access`
`{SERVER API APP CLIENT ID}`	Application (client) ID for the *Server API app*	`41451fa7-82d9-4673-8fa5-69eff5a761fd`
`{SERVER API APP ID URI}`	Application ID URI (see note)	`41451fa7-82d9-4673-8fa5-69eff5a761fd`
`{TENANT DOMAIN}`	Primary/Publisher/Tenant domain	`contoso.onmicrosoft.com`
`{TENANT ID}`	Directory (tenant) ID	`e86c78e2-8bb4-4c41-aefd-918e0565a45e`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the app's name.

> **NOTE**
>
> Pass the App ID URI to the `app-id-uri` option, but note a configuration change might be required in the client app, which is described in the Access token scopes section.

> **NOTE**
>
> In the Azure portal, the *Client app's* **Authentication** > **Platform configurations** > **Web** > **Redirect URI** is configured for port 5001 for apps that run on the Kestrel server with default settings.
>
> If the *Client app* is run on a random IIS Express port, the port for the app can be found in the *Server API app's* properties in the **Debug** panel.
>
> If the port wasn't configured earlier with the *Client app's* known port, return to the *Client app's* registration in the Azure portal and update the redirect URI with the correct port.

Server app configuration

This section pertains to the solution's `Server` app.

Authentication package

The support for authenticating and authorizing calls to ASP.NET Core Web APIs is provided by the `Microsoft.AspNetCore.Authentication.AzureAD.UI` package:

```
<PackageReference Include="Microsoft.AspNetCore.Authentication.AzureAD.UI"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

Authentication service support

The `AddAuthentication` method sets up authentication services within the app and configures the JWT Bearer handler as the default authentication method. The AddAzureADBearer method sets up the specific parameters in the JWT Bearer handler required to validate tokens emitted by the Azure Active Directory:

```
services.AddAuthentication(AzureADDefaults.BearerAuthenticationScheme)
    .AddAzureADBearer(options => Configuration.Bind("AzureAd", options));
```

UseAuthentication and UseAuthorization ensure that:

- The app attempts to parse and validate tokens on incoming requests.
- Any request attempting to access a protected resource without proper credentials fails.

```
app.UseAuthentication();
app.UseAuthorization();
```

User.Identity.Name

By default, the Server app API populates `User.Identity.Name` with the value from the `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name` claim type (for example, `2d64b3da-d9d5-42c6-9352-53d8df33d770@contoso.onmicrosoft.com`).

To configure the app to receive the value from the `name` claim type, configure the TokenValidationParameters.NameClaimType of the JwtBearerOptions in `Startup.ConfigureServices`:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;

...

services.Configure<JwtBearerOptions>(
    AzureADDefaults.JwtBearerAuthenticationScheme, options =>
    {
        options.TokenValidationParameters.NameClaimType = "name";
    });
```

App settings

The `appsettings.json` file contains the options to configure the JWT bearer handler used to validate access tokens:

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "{DOMAIN}",
    "TenantId": "{TENANT ID}",
    "ClientId": "{SERVER API APP CLIENT ID}",
  }
}
```

Example:

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "contoso.onmicrosoft.com",
    "TenantId": "e86c78e2-8bb4-4c41-aefd-918e0565a45e",
    "ClientId": "41451fa7-82d9-4673-8fa5-69eff5a761fd",
  }
}
```

WeatherForecast controller

The WeatherForecast controller (*Controllers/WeatherForecastController.cs*) exposes a protected API with the `[Authorize]` attribute applied to the controller. It's **important** to understand that:

- The `[Authorize]` attribute in this API controller is the only thing that protect this API from unauthorized access.
- The `[Authorize]` attribute used in the Blazor WebAssembly app only serves as a hint to the app that the user should be authorized for the app to work correctly.

```
[Authorize]
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    [HttpGet]
    public IEnumerable<WeatherForecast> Get()
    {
        ...
    }
}
```

Client app configuration

This section pertains to the solution's `Client` app.

Authentication package

When an app is created to use Work or School Accounts (`SingleOrg`), the app automatically receives a package reference for the Microsoft Authentication Library (`Microsoft.Authentication.WebAssembly.Msal`). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the package to the app's project file:

```
<PackageReference Include="Microsoft.Authentication.WebAssembly.Msal"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}` , the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

The `Microsoft.Authentication.WebAssembly.Msal` package transitively adds the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package to the app.

Authentication service support

Support for HttpClient instances is added that include access tokens when making requests to the server project.

`Program.cs` :

```
builder.Services.AddHttpClient("{APP ASSEMBLY}.ServerAPI", client =>
        client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>()
    .CreateClient("{APP ASSEMBLY}.ServerAPI"));
```

The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `BlazorSample.ServerAPI`).

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the `Microsoft.Authentication.WebAssembly.Msal` package. This method sets up the services required for the app to interact with the Identity Provider (IP).

`Program.cs`:

```
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAd", options.ProviderOptions.Authentication);
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the Azure Portal AAD configuration when you register the app.

Configuration is supplied by the `wwwroot/appsettings.json` file:

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT APP CLIENT ID}",
    "ValidateAuthority": true
  }
}
```

Example:

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/e86c78e2-...-918e0565a45e",
    "ClientId": "4369008b-21fa-427c-abaa-9b53bf58e538",
    "ValidateAuthority": true
  }
}
```

Access token scopes

The default access token scopes represent the list of access token scopes that are:

- Included by default in the sign in request.
- Used to provision an access token immediately after authentication.

All scopes must belong to the same app per Azure Active Directory rules. Additional scopes can be added for additional API apps as needed:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

NOTE

If the Azure portal provides the scope URI for the app and the app throws an unhandled exception when it receives a *401 Unauthorized* response from the API, try using a scope URI that doesn't include the scheme and host. For example, the Azure portal may provide one of the following scope URI formats:

- `https://{TENANT}.onmicrosoft.com/{API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`
- `api://{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`

Try supplying the scope URI without the scheme and host:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}");
```

For example:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "41451fa7-82d9-4673-8fa5-69eff5a761fd/API.Access");
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}.Client
@using {APPLICATION ASSEMBLY}.Client.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```
<script src="_content/Microsoft.Authentication.WebAssembly.Msal/
    AuthenticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The CascadingAuthenticationState component manages exposing the AuthenticationState to the rest of the app.
- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <p>
                            You are not authorized to access
                            this resource.
                        </p>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

RedirectToLogin component

The `RedirectToLogin` component (`Shared/RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- Preserves the current URL that the user is attempting to access so that they can be returned to that page if authentication is successful.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri));
    }
}
```

LoginDisplay component

The `LoginDisplay` component (`Shared/LoginDisplay.razor`) is rendered in the `MainLayout` component (`Shared/MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current username.
  - Offers a button to log out of the app.
- For anonymous users, offers the option to log in.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
        <button class="nav-link btn btn-link" @onclick="BeginLogout">
            Log out
        </button>
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

@code {
    private async Task BeginLogout(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package.
- Manages performing the appropriate actions at each stage of authentication.

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string Action { get; set; }
}
```

FetchData component

The `FetchData` component shows how to:

- Provision an access token.
- Use the access token to call a protected resource API in the *Server* app.

The `@attribute [Authorize]` directive indicates to the Blazor WebAssembly authorization system that the user must be authorized in order to visit this component. The presence of the attribute in the `Client` app doesn't prevent the API on the server from being called without proper credentials. The `Server` app also must use `[Authorize]` on the appropriate endpoints to correctly protect them.

IAccessTokenProvider.RequestAccessToken takes care of requesting an access token that can be added to the request to call the API. If the token is cached or the service is able to provision a new access token without user interaction, the token request succeeds. Otherwise, the token request fails with an AccessTokenNotAvailableException, which is caught in a `try-catch` statement.

In order to obtain the actual token to include in the request, the app must check that the request succeeded by calling `tokenResult.TryGetToken(out var token)`.

If the request was successful, the token variable is populated with the access token. The AccessToken.Value property of the token exposes the literal string to include in the `Authorization` request header.

If the request failed because the token couldn't be provisioned without user interaction, the token result contains a redirect URL. Navigating to this URL takes the user to the login page and back to the current page after a successful authentication.

```
@page "/fetchdata"
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using {APP NAMESPACE}.Shared
@attribute [Authorize]
@inject HttpClient Http

...

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        try
        {
            forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>("WeatherForecast");
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
}
```

Run the app

Run the app from the Server project. When using Visual Studio, either:

- Set the **Startup Projects** drop down list in the toolbar to the *Server API app* and select the **Run** button.
- Select the Server project in **Solution Explorer** and select the **Run** button in the toolbar or start the app from the **Debug** menu.

Troubleshoot

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE between any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in incognito or private mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.

    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in incognito or private mode. Some browsers require the URL of the app.
    - Microsoft Edge: `-inprivate`
    - Google Chrome: `--incognito --new-window https://localhost:5001`
    - Mozilla Firefox: `-private -url https://localhost:5001`
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE between any change to the app, test user, or provider configuration.

Run the Server app

When testing and troubleshooting a hosted Blazor app, make sure that you're running the app from the `Server` project. For example in Visual Studio, confirm that the Server project is highlighted in **Solution Explorer** before you start the app with any of the following approaches:

- Select the **Run** button.
- Use **Debug** > **Start Debugging** from the menu.
- Press F5.

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- ASP.NET Core Blazor WebAssembly with Azure Active Directory groups and roles
- Microsoft identity platform and Azure Active Directory with ASP.NET Core
- Microsoft identity platform documentation

Secure an ASP.NET Core Blazor WebAssembly hosted app with Azure Active Directory B2C

9/22/2020 • 15 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

This article describes how to create a hosted Blazor WebAssembly app that uses Azure Active Directory (AAD) B2C for authentication.

Register apps in AAD B2C and create solution

Create a tenant

Follow the guidance in Tutorial: Create an Azure Active Directory B2C tenant to create an AAD B2C tenant.

Record the AAD B2C instance (for example, `https://contoso.b2clogin.com/`, which includes the trailing slash). The instance is the scheme and host of an Azure B2C app registration, which can be found by opening the **Endpoints** window from the **App registrations** page in the Azure portal.

Register a server API app

Follow the guidance in Tutorial: Register an application in Azure Active Directory B2C to register an AAD app for the *Server API app* and then do the following:

1. In **Azure Active Directory** > **App registrations**, select **New registration**.
2. Provide a **Name** for the app (for example, **Blazor Server AAD B2C**).
3. For **Supported account types**, select the multi-tenant option: **Accounts in any organizational directory or any identity provider. For authenticating users with Azure AD B2C.**
4. The *Server API app* doesn't require a **Redirect URI** in this scenario, so leave the drop down set to **Web** and don't enter a redirect URI.
5. Confirm that **Permissions** > **Grant admin consent to openid and offline_access permissions** is enabled.
6. Select **Register**.

Record the following information:

- *Server API app* Application (client) ID (for example, `41451fa7-82d9-4673-8fa5-69eff5a761fd`)
- AAD Primary/Publisher/Tenant domain (for example, `contoso.onmicrosoft.com`): The domain is available as the **Publisher domain** in the **Branding** blade of the Azure portal for the registered app.

In **Expose an API**:

1. Select **Add a scope**.
2. Select **Save and continue**.
3. Provide a **Scope name** (for example, `API.Access`).
4. Provide an **Admin consent display name** (for example, `Access API`).
5. Provide an **Admin consent description** (for example, `Allows the app to access server app API endpoints.`).
6. Confirm that the **State** is set to **Enabled**.
7. Select **Add scope**.

Record the following information:

- App ID URI (for example, `https://contoso.onmicrosoft.com/41451fa7-82d9-4673-8fa5-69eff5a761fd`, `api://41451fa7-82d9-4673-8fa5-69eff5a761fd`, or the custom value that you provided)
- Scope name (for example, `API.Access`)

The App ID URI might require a special configuration in the client app, which is described in the Access token scopes section later in this topic.

Register a client app

Follow the guidance in Tutorial: Register an application in Azure Active Directory B2C again to register an AAD app for the *Client app* and then do the following:

1. In **Azure Active Directory** > **App registrations**, select **New registration**.
2. Provide a **Name** for the app (for example, **Blazor Client AAD B2C**).
3. For **Supported account types**, select the multi-tenant option: **Accounts in any organizational directory or any identity provider. For authenticating users with Azure AD B2C.**
4. Leave the **Redirect URI** drop down set to **Web** and provide the following redirect URI: `https://localhost:{PORT}/authentication/login-callback`. The default port for an app running on Kestrel is 5001. If the app is run on a different Kestrel port, use the app's port. For IIS Express, the randomly generated port for the app can be found in the Server app's properties in the **Debug** panel. Since the app doesn't exist at this point and the IIS Express port isn't known, return to this step after the app is created and update the redirect URI. A remark appears in the Create the app section to remind IIS Express users to update the redirect URI.
5. Confirm that **Permissions** > **Grant admin consent to openid and offline_access permissions** is enabled.
6. Select **Register**.

Record the Application (client) ID (for example, `4369008b-21fa-427c-abaa-9b53bf58e538`).

In **Authentication** > **Platform configurations** > **Web**:

1. Confirm the **Redirect URI** of `https://localhost:{PORT}/authentication/login-callback` is present.
2. For **Implicit grant**, select the check boxes for **Access tokens** and **ID tokens**.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button.

In **API permissions**:

1. Select **Add a permission** followed by **My APIs**.
2. Select the *Server API app* from the **Name** column (for example, **Blazor Server AAD B2C**).
3. Open the **API** list.
4. Enable access to the API (for example, `API.Access`).
5. Select **Add permissions**.
6. Select the **Grant admin consent for {TENANT NAME}** button. Select **Yes** to confirm.

In **Home** > **Azure AD B2C** > **User flows**:

Create a sign-up and sign-in user flow

At a minimum, select the **Application claims** > **Display Name** user attribute to populate the `context.User.Identity.Name` in the `LoginDisplay` component (`Shared/LoginDisplay.razor`).

Record the sign-up and sign-in user flow name created for the app (for example, `B2C_1_signupsignin`).

Create the app

Replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

```
dotnet new blazorwasm -au IndividualB2C --aad-b2c-instance "{AAD B2C INSTANCE}" --api-client-id "{SERVER API APP CLIENT ID}" --app-id-uri "{SERVER API APP ID URI}" --client-id "{CLIENT APP CLIENT ID}" --default-scope "{DEFAULT SCOPE}" --domain "{TENANT DOMAIN}" -ho -o {APP NAME} -ssp "{SIGN UP OR SIGN IN POLICY}"
```

PLACEHOLDER	AZURE PORTAL NAME	EXAMPLE
`{AAD B2C INSTANCE}`	Instance	`https://contoso.b2clogin.com/`
`{APP NAME}`	—	`BlazorSample`
`{CLIENT APP CLIENT ID}`	Application (client) ID for the *Client app*	`4369008b-21fa-427c-abaa-9b53bf58e538`
`{DEFAULT SCOPE}`	Scope name	`API.Access`
`{SERVER API APP CLIENT ID}`	Application (client) ID for the *Server API app*	`41451fa7-82d9-4673-8fa5-69eff5a761fd`
`{SERVER API APP ID URI}`	Application ID URI (see note)	`41451fa7-82d9-4673-8fa5-69eff5a761fd`
`{SIGN UP OR SIGN IN POLICY}`	Sign-up/sign-in user flow	`B2C_1_signupsignin1`
`{TENANT DOMAIN}`	Primary/Publisher/Tenant domain	`contoso.onmicrosoft.com`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the app's name.

> **NOTE**
>
> Pass the App ID URI to the `app-id-uri` option, but note a configuration change might be required in the client app, which is described in the Access token scopes section.
>
> Additionally, the scope set up by the Hosted Blazor template might have the App ID URI host repeated. Confirm that the scope configured for the `DefaultAccessTokenScopes` collection is correct in `Program.Main` (`Program.cs`) of the *Client app*.

> **NOTE**
>
> In the Azure portal, the *Client app's* **Authentication** > **Platform configurations** > **Web** > **Redirect URI** is configured for port 5001 for apps that run on the Kestrel server with default settings.
>
> If the *Client app* is run on a random IIS Express port, the port for the app can be found in the *Server API app's* properties in the **Debug** panel.
>
> If the port wasn't configured earlier with the *Client app's* known port, return to the *Client app's* registration in the Azure portal and update the redirect URI with the correct port.

Server app configuration

This section pertains to the solution's `Server` *app.*

Authentication package

The support for authenticating and authorizing calls to ASP.NET Core Web APIs is provided by the `Microsoft.AspNetCore.Authentication.AzureADB2C.UI` package:

```
<PackageReference Include="Microsoft.AspNetCore.Authentication.AzureADB2C.UI"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

Authentication service support

The `AddAuthentication` method sets up authentication services within the app and configures the JWT Bearer handler as the default authentication method. The AddAzureADB2CBearer method sets up the specific parameters in the JWT Bearer handler required to validate tokens emitted by the Azure Active Directory B2C:

```
services.AddAuthentication(AzureADB2CDefaults.BearerAuthenticationScheme)
    .AddAzureADB2CBearer(options => Configuration.Bind("AzureAdB2C", options));
```

UseAuthentication and UseAuthorization ensure that:

- The app attempts to parse and validate tokens on incoming requests.
- Any request attempting to access a protected resource without proper credentials fails.

```
app.UseAuthentication();
app.UseAuthorization();
```

User.Identity.Name

By default, the `User.Identity.Name` isn't populated.

To configure the app to receive the value from the `name` claim type, configure the TokenValidationParameters.NameClaimType of the JwtBearerOptions in `Startup.ConfigureServices`:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;

...

services.Configure<JwtBearerOptions>(
    AzureADB2CDefaults.JwtBearerAuthenticationScheme, options =>
    {
        options.TokenValidationParameters.NameClaimType = "name";
    });
```

App settings

The `appsettings.json` file contains the options to configure the JWT bearer handler used to validate access tokens.

```
{
  "AzureAdB2C": {
    "Instance": "https://{TENANT}.b2clogin.com/",
    "ClientId": "{SERVER API APP CLIENT ID}",
    "Domain": "{TENANT DOMAIN}",
    "SignUpSignInPolicyId": "{SIGN UP OR SIGN IN POLICY}"
  }
}
```

Example:

```
{
  "AzureAdB2C": {
    "Instance": "https://contoso.b2clogin.com/",
    "ClientId": "41451fa7-82d9-4673-8fa5-69eff5a761fd",
    "Domain": "contoso.onmicrosoft.com",
    "SignUpSignInPolicyId": "B2C_1_signupsignin1",
  }
}
```

WeatherForecast controller

The WeatherForecast controller (*Controllers/WeatherForecastController.cs*) exposes a protected API with the `[Authorize]` attribute applied to the controller. It's **important** to understand that:

- The `[Authorize]` attribute in this API controller is the only thing that protect this API from unauthorized access.
- The `[Authorize]` attribute used in the Blazor WebAssembly app only serves as a hint to the app that the user should be authorized for the app to work correctly.

```
[Authorize]
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    [HttpGet]
    public IEnumerable<WeatherForecast> Get()
    {
        ...
    }
}
```

Client app configuration

This section pertains to the solution's `Client` app.

Authentication package

When an app is created to use an Individual B2C Account (`IndividualB2C`), the app automatically receives a package reference for the Microsoft Authentication Library (`Microsoft.Authentication.WebAssembly.Msal`). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the package to the app's project file:

```
<PackageReference Include="Microsoft.Authentication.WebAssembly.Msal"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}` , the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

The `Microsoft.Authentication.WebAssembly.Msal` package transitively adds the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package to the app.

Authentication service support

Support for HttpClient instances is added that include access tokens when making requests to the server project.

`Program.cs`:

```
builder.Services.AddHttpClient("{APP ASSEMBLY}.ServerAPI", client =>
    client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>()
    .CreateClient("{APP ASSEMBLY}.ServerAPI"));
```

The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `BlazorSample.ServerAPI`).

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the `Microsoft.Authentication.WebAssembly.Msal` package. This method sets up the services required for the app to interact with the Identity Provider (IP).

`Program.cs`:

```
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAdB2C", options.ProviderOptions.Authentication);
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the Azure Portal AAD configuration when you register the app.

Configuration is supplied by the `wwwroot/appsettings.json` file:

```
{
  "AzureAdB2C": {
    "Authority": "{AAD B2C INSTANCE}{TENANT DOMAIN}/{SIGN UP OR SIGN IN POLICY}",
    "ClientId": "{CLIENT APP CLIENT ID}",
    "ValidateAuthority": false
  }
}
```

Example:

```
{
  "AzureAdB2C": {
    "Authority": "https://contoso.b2clogin.com/contoso.onmicrosoft.com/B2C_1_signupsignin1",
    "ClientId": "4369008b-21fa-427c-abaa-9b53bf58e538",
    "ValidateAuthority": false
  }
}
```

Access token scopes

The default access token scopes represent the list of access token scopes that are:

- Included by default in the sign in request.
- Used to provision an access token immediately after authentication.

All scopes must belong to the same app per Azure Active Directory rules. Additional scopes can be added for additional API apps as needed:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

NOTE

If the Azure portal provides the scope URI for the app and the app throws an unhandled exception when it receives a *401 Unauthorized* response from the API, try using a scope URI that doesn't include the scheme and host. For example, the Azure portal may provide one of the following scope URI formats:

- `https://{TENANT}.onmicrosoft.com/{API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`
- `api://{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}`

Try supplying the scope URI without the scheme and host:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "{SERVER API CLIENT ID OR CUSTOM VALUE}/{SCOPE NAME}");
```

For example:

```
options.ProviderOptions.DefaultAccessTokenScopes.Add(
    "41451fa7-82d9-4673-8fa5-69eff5a761fd/API.Access");
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}.Client
@using {APPLICATION ASSEMBLY}.Client.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```
<script src="_content/Microsoft.Authentication.WebAssembly.Msal/
    AuthenticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The CascadingAuthenticationState component manages exposing the AuthenticationState to the rest of the app.
- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <p>
                            You are not authorized to access
                            this resource.
                        </p>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

RedirectToLogin component

The `RedirectToLogin` component (`Shared/RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- Preserves the current URL that the user is attempting to access so that they can be returned to that page if authentication is successful.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri));
    }
}
```

LoginDisplay component

The `LoginDisplay` component (`Shared/LoginDisplay.razor`) is rendered in the `MainLayout` component (`Shared/MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current username.
  - Offers a button to log out of the app.
- For anonymous users, offers the option to log in.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity.Name!
        <button class="nav-link btn btn-link" @onclick="BeginLogout">
            Log out
        </button>
    </Authorized>
    <NotAuthorized>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

@code {
    private async Task BeginLogout(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package.
- Manages performing the appropriate actions at each stage of authentication.

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string Action { get; set; }
}
```

FetchData component

The `FetchData` component shows how to:

- Provision an access token.
- Use the access token to call a protected resource API in the *Server* app.

The `@attribute [Authorize]` directive indicates to the Blazor WebAssembly authorization system that the user must be authorized in order to visit this component. The presence of the attribute in the `Client` app doesn't prevent the API on the server from being called without proper credentials. The `Server` app also must use `[Authorize]` on the appropriate endpoints to correctly protect them.

IAccessTokenProvider.RequestAccessToken takes care of requesting an access token that can be added to the request to call the API. If the token is cached or the service is able to provision a new access token without user interaction, the token request succeeds. Otherwise, the token request fails with an AccessTokenNotAvailableException, which is caught in a `try-catch` statement.

In order to obtain the actual token to include in the request, the app must check that the request succeeded by calling `tokenResult.TryGetToken(out var token)`.

If the request was successful, the token variable is populated with the access token. The AccessToken.Value property of the token exposes the literal string to include in the `Authorization` request header.

If the request failed because the token couldn't be provisioned without user interaction, the token result contains a redirect URL. Navigating to this URL takes the user to the login page and back to the current page after a successful authentication.

```
@page "/fetchdata"
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using {APP NAMESPACE}.Shared
@attribute [Authorize]
@inject HttpClient Http

...

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        try
        {
            forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>("WeatherForecast");
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
}
```

Run the app

Run the app from the Server project. When using Visual Studio, either:

- Set the **Startup Projects** drop down list in the toolbar to the *Server API app* and select the **Run** button.
- Select the Server project in **Solution Explorer** and select the **Run** button in the toolbar or start the app from the **Debug** menu.

Custom user flows

The Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal, NuGet package) doesn't support AAD B2C user flows by default. Create custom user flows in developer code.

For more information on how to build a challenge for a custom user flow, see User flows in Azure Active Directory B2C.

Troubleshoot

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE between any change to the app, test user,

or provider configuration.

- Use a custom command to open a browser in incognito or private mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in incognito or private mode. Some browsers require the URL of the app.
    - Microsoft Edge: `-inprivate`
    - Google Chrome: `--incognito --new-window https://localhost:5001`
    - Mozilla Firefox: `-private -url https://localhost:5001`
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE between any change to the app, test user, or provider configuration.

Run the Server app

When testing and troubleshooting a hosted Blazor app, make sure that you're running the app from the `Server` project. For example in Visual Studio, confirm that the Server project is highlighted in **Solution Explorer** before you start the app with any of the following approaches:

- Select the **Run** button.
- Use **Debug** > **Start Debugging** from the menu.
- Press F5.

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- Cloud authentication with Azure Active Directory B2C in ASP.NET Core
- Tutorial: Create an Azure Active Directory B2C tenant
- Microsoft identity platform documentation

Secure an ASP.NET Core Blazor WebAssembly hosted app with Identity Server

9/22/2020 • 13 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

This article explains how to create a hosted Blazor WebAssembly app that uses IdentityServer to authenticate users and API calls.

> **NOTE**
> To configure a standalone or hosted Blazor WebAssembly app to use an existing, external Identity Server instance, follow the guidance in Secure an ASP.NET Core Blazor WebAssembly standalone app with the Authentication library.

- Visual Studio
- Visual Studio Code / .NET Core CLI
- Visual Studio for Mac

To create a new Blazor WebAssembly project with an authentication mechanism:

1. After choosing the **Blazor WebAssembly App** template in the **Create a new ASP.NET Core Web Application** dialog, select **Change** under **Authentication**.
2. Select **Individual User Accounts** with the **Store user accounts in-app** option to store users within the app using ASP.NET Core's Identity system.
3. Select the **ASP.NET Core hosted** check box in the **Advanced** section.

Server app configuration

The following sections describe additions to the project when authentication support is included.

Startup class

The `Startup` class has the following additions.

- In `Startup.ConfigureServices`:
  - ASP.NET Core Identity:

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(
        Configuration.GetConnectionString("DefaultConnection")));

services.AddDefaultIdentity<ApplicationUser>(options =>
        options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

  - IdentityServer with an additional AddApiAuthorization helper method that sets up default ASP.NET Core conventions on top of IdentityServer:

```
services.AddIdentityServer()
    .AddApiAuthorization<ApplicationUser, ApplicationDbContext>();
```

- Authentication with an additional AddIdentityServerJwt helper method that configures the app to validate JWT tokens produced by IdentityServer:

```
services.AddAuthentication()
    .AddIdentityServerJwt();
```

- In `Startup.Configure`:
  - The IdentityServer middleware exposes the OpenID Connect (OIDC) endpoints:

```
app.UseIdentityServer();
```

  - The Authentication middleware is responsible for validating request credentials and setting the user on the request context:

```
app.UseAuthentication();
```

  - Authorization Middleware enables authorization capabilities:

```
app.UseAuthentication();
app.UseAuthorization();
```

AddApiAuthorization

The AddApiAuthorization helper method configures IdentityServer for ASP.NET Core scenarios. IdentityServer is a powerful and extensible framework for handling app security concerns. IdentityServer exposes unnecessary complexity for the most common scenarios. Consequently, a set of conventions and configuration options is provided that we consider a good starting point. Once your authentication needs change, the full power of IdentityServer is available to customize authentication to suit an app's requirements.

AddIdentityServerJwt

The AddIdentityServerJwt helper method configures a policy scheme for the app as the default authentication handler. The policy is configured to allow Identity to handle all requests routed to any subpath in the Identity URL space `/Identity`. The JwtBearerHandler handles all other requests. Additionally, this method:

- Registers an `{APPLICATION NAME}API` API resource with IdentityServer with a default scope of `{APPLICATION NAME}API`.
- Configures the JWT Bearer Token Middleware to validate tokens issued by IdentityServer for the app.

WeatherForecastController

In the `WeatherForecastController` (`Controllers/WeatherForecastController.cs`), the `[Authorize]` attribute is applied to the class. The attribute indicates that the user must be authorized based on the default policy to access the resource. The default authorization policy is configured to use the default authentication scheme, which is set up by AddIdentityServerJwt. The helper method configures JwtBearerHandler as the default handler for requests to the app.

ApplicationDbContext

In the `ApplicationDbContext` (`Data/ApplicationDbContext.cs`), DbContext extends ApiAuthorizationDbContext<TUser> to include the schema for IdentityServer.

ApiAuthorizationDbContext<TUser> is derived from IdentityDbContext.

To gain full control of the database schema, inherit from one of the available Identity DbContext classes and configure the context to include the Identity schema by calling `builder.ConfigurePersistedGrantContext(_operationalStoreOptions.Value)` in the OnModelCreating method.

OidcConfigurationController

In the `OidcConfigurationController` (`Controllers/OidcConfigurationController.cs`), the client endpoint is provisioned to serve OIDC parameters.

App settings

In the app settings file (`appsettings.json`) at the project root, the `IdentityServer` section describes the list of configured clients. In the following example, there's a single client. The client name corresponds to the app name and is mapped by convention to the OAuth `ClientId` parameter. The profile indicates the app type being configured. The profile is used internally to drive conventions that simplify the configuration process for the server.

```
"IdentityServer": {
  "Clients": {
    "{APP ASSEMBLY}.Client": {
      "Profile": "IdentityServerSPA"
    }
  }
}
```

The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `BlazorSample.Client`).

Client app configuration

Authentication package

When an app is created to use Individual User Accounts (`Individual`), the app automatically receives a package reference for the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package in the app's project file. The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the package to the app's project file:

```
<PackageReference
  Include="Microsoft.AspNetCore.Components.WebAssembly.Authentication"
  Version="{VERSION}" />
```

For the placeholder `{VERSION}`, the latest stable version of the package that matches the app's shared framework version can be found in the package's **Version History** at NuGet.org.

`HttpClient` configuration

In `Program.Main` (`Program.cs`), a named HttpClient (`HostIS.ServerAPI`) is configured to supply HttpClient instances that include access tokens when making requests to the server API:

```
builder.Services.AddHttpClient("HostIS.ServerAPI",
        client => client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>()
    .CreateClient("HostIS.ServerAPI"));
```

> **NOTE**
>
> If you're configuring a Blazor WebAssembly app to use an existing Identity Server instance that isn't part of a hosted Blazor solution, change the HttpClient base address registration from IWebAssemblyHostEnvironment.BaseAddress (`builder.HostEnvironment.BaseAddress`) to the server app's API authorization endpoint URL.

API authorization support

The support for authenticating users is plugged into the service container by the extension method provided inside the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package. This method sets up the services required by the app to interact with the existing authorization system.

```
builder.Services.AddApiAuthorization();
```

By default, configuration for the app is loaded by convention from `_configuration/{client-id}`. By convention, the client ID is set to the app's assembly name. This URL can be changed to point to a separate endpoint by calling the overload with options.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}.Client
@using {APPLICATION ASSEMBLY}.Client.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```
<script src="_content/Microsoft.AspNetCore.Components.WebAssembly.Authentication/
    AuthenticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The CascadingAuthenticationState component manages exposing the AuthenticationState to the rest of the app.
- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    @if (!context.User.Identity.IsAuthenticated)
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <p>
                            You are not authorized to access
                            this resource.
                        </p>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

RedirectToLogin component

The `RedirectToLogin` component (`Shared/RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- Preserves the current URL that the user is attempting to access so that they can be returned to that page if authentication is successful.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri));
    }
}
```

LoginDisplay component

The `LoginDisplay` component (`Shared/LoginDisplay.razor`) is rendered in the `MainLayout` component (`Shared/MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager

<AuthorizeView>
    <Authorized>
        <a href="authentication/profile">Hello, @context.User.Identity.Name!</a>
        <button class="nav-link btn btn-link" @onclick="BeginSignOut">
            Log out
        </button>
    </Authorized>
    <NotAuthorized>
        <a href="authentication/register">Register</a>
        <a href="authentication/login">Log in</a>
    </NotAuthorized>
</AuthorizeView>

@code {
    private async Task BeginSignOut(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` package.
- Manages performing the appropriate actions at each stage of authentication.

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string Action { get; set; }
}
```

FetchData component

The `FetchData` component shows how to:

- Provision an access token.
- Use the access token to call a protected resource API in the *Server* app.

The `@attribute [Authorize]` directive indicates to the Blazor WebAssembly authorization system that the user must be authorized in order to visit this component. The presence of the attribute in the `Client` app doesn't prevent the API on the server from being called without proper credentials. The `Server` app also must use `[Authorize]` on the appropriate endpoints to correctly protect them.

IAccessTokenProvider.RequestAccessToken takes care of requesting an access token that can be added to the request to call the API. If the token is cached or the service is able to provision a new access token without user interaction, the token request succeeds. Otherwise, the token request fails with an

AccessTokenNotAvailableException, which is caught in a `try-catch` statement.

In order to obtain the actual token to include in the request, the app must check that the request succeeded by calling `tokenResult.TryGetToken(out var token)`.

If the request was successful, the token variable is populated with the access token. The AccessToken.Value property of the token exposes the literal string to include in the `Authorization` request header.

If the request failed because the token couldn't be provisioned without user interaction, the token result contains a redirect URL. Navigating to this URL takes the user to the login page and back to the current page after a successful authentication.

```
@page "/fetchdata"
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using {APP NAMESPACE}.Shared
@attribute [Authorize]
@inject HttpClient Http

...

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        try
        {
            forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>("WeatherForecast");
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
}
```

Run the app

Run the app from the Server project. When using Visual Studio, either:

- Set the **Startup Projects** drop down list in the toolbar to the *Server API app* and select the **Run** button.
- Select the Server project in **Solution Explorer** and select the **Run** button in the toolbar or start the app from the **Debug** menu.

Name and role claim with API authorization

Custom user factory

In the Client app, create a custom user factory. Identity Server sends multiple roles as a JSON array in a single `role` claim. A single role is sent as a string value in the claim. The factory creates an individual `role` claim for each of the user's roles.

`CustomUserFactory.cs`:

```
using System.Linq;
using System.Security.Claims;
using System.Text.Json;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;

public class CustomUserFactory
    : AccountClaimsPrincipalFactory<RemoteUserAccount>
{
    public CustomUserFactory(IAccessTokenProviderAccessor accessor)
        : base(accessor)
    {
    }

    public async override ValueTask<ClaimsPrincipal> CreateUserAsync(
        RemoteUserAccount account,
        RemoteAuthenticationUserOptions options)
    {
        var user = await base.CreateUserAsync(account, options);

        if (user.Identity.IsAuthenticated)
        {
            var identity = (ClaimsIdentity)user.Identity;
            var roleClaims = identity.FindAll(identity.RoleClaimType);

            if (roleClaims != null && roleClaims.Any())
            {
                foreach (var existingClaim in roleClaims)
                {
                    identity.RemoveClaim(existingClaim);
                }

                var rolesElem = account.AdditionalProperties[identity.RoleClaimType];

                if (rolesElem is JsonElement roles)
                {
                    if (roles.ValueKind == JsonValueKind.Array)
                    {
                        foreach (var role in roles.EnumerateArray())
                        {
                            identity.AddClaim(new Claim(options.RoleClaim, role.GetString()));
                        }
                    }
                    else
                    {
                        identity.AddClaim(new Claim(options.RoleClaim, roles.GetString()));
                    }
                }
            }
        }

        return user;
    }
}
```

In the Client app, register the factory in `Program.Main` (`Program.cs`):

```
builder.Services.AddApiAuthorization()
    .AddAccountClaimsPrincipalFactory<CustomUserFactory>();
```

In the Server app, call AddRoles on the Identity builder, which adds role-related services:

```
using Microsoft.AspNetCore.Identity;

...

services.AddDefaultIdentity<ApplicationUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Configure Identity Server

Use **one** of the following approaches:

- API authorization options
- Profile Service

API authorization options

In the Server app:

- Configure Identity Server to put the `name` and `role` claims into the ID token and access token.
- Prevent the default mapping for roles in the JWT token handler.

```
using System.IdentityModel.Tokens.Jwt;
using System.Linq;

...

services.AddIdentityServer()
    .AddApiAuthorization<ApplicationUser, ApplicationDbContext>(options => {
        options.IdentityResources["openid"].UserClaims.Add("name");
        options.ApiResources.Single().UserClaims.Add("name");
        options.IdentityResources["openid"].UserClaims.Add("role");
        options.ApiResources.Single().UserClaims.Add("role");
    });

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Remove("role");
```

Profile Service

In the Server app, create a `ProfileService` implementation.

`ProfileService.cs`:

```
using IdentityModel;
using IdentityServer4.Models;
using IdentityServer4.Services;
using System.Threading.Tasks;

public class ProfileService : IProfileService
{
    public ProfileService()
    {
    }

    public Task GetProfileDataAsync(ProfileDataRequestContext context)
    {
        var nameClaim = context.Subject.FindAll(JwtClaimTypes.Name);
        context.IssuedClaims.AddRange(nameClaim);

        var roleClaims = context.Subject.FindAll(JwtClaimTypes.Role);
        context.IssuedClaims.AddRange(roleClaims);

        return Task.CompletedTask;
    }

    public Task IsActiveAsync(IsActiveContext context)
    {
        return Task.CompletedTask;
    }
}
```

In the Server app, register the Profile Service in `Startup.ConfigureServices`:

```
using IdentityServer4.Services;

...

services.AddTransient<IProfileService, ProfileService>();
```

Use authorization mechanisms

In the Client app, component authorization approaches are functional at this point. Any of the authorization mechanisms in components can use a role to authorize the user:

- `AuthorizeView` component (Example: `<AuthorizeView Roles="admin">`)
- `[Authorize]` attribute directive (AuthorizeAttribute) (Example: `@attribute [Authorize(Roles = "admin")]`)
- Procedural logic (Example: `if (user.IsInRole("admin")) { ... }`)

  Multiple role tests are supported:

  ```
  if (user.IsInRole("admin") && user.IsInRole("developer"))
  {
      ...
  }
  ```

`User.Identity.Name` is populated in the Client app with the user's user name, which is usually their sign-in email address.

UserManager and SignInManager

Set the user identifier claim type when a Server app requires:

- UserManager<TUser> or SignInManager<TUser> in an API endpoint.
- IdentityUser details, such as the user's name, email address, or lockout end time.

In `Startup.ConfigureServices`:

```
using System.Security.Claims;

...

services.Configure<IdentityOptions>(options =>
    options.ClaimsIdentity.UserIdClaimType = ClaimTypes.NameIdentifier);
```

The following `WeatherForecastController` logs the UserName when the `Get` method is called:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Logging;
using {APP NAMESPACE}.Server.Models;
using {APP NAMESPACE}.Shared;

namespace {APP NAMESPACE}.Server.Controllers
{
    [Authorize]
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase
    {
        private readonly UserManager<ApplicationUser> userManager;

        private static readonly string[] Summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
            "Balmy", "Hot", "Sweltering", "Scorching"
        };

        private readonly ILogger<WeatherForecastController> logger;

        public WeatherForecastController(ILogger<WeatherForecastController> logger,
            UserManager<ApplicationUser> userManager)
        {
            this.logger = logger;
            this.userManager = userManager;
        }

        [HttpGet]
        public async Task<IEnumerable<WeatherForecast>> Get()
        {
            var rng = new Random();

            var user = await userManager.GetUserAsync(User);

            if (user != null)
            {
                logger.LogInformation($"User.Identity.Name: {user.UserName}");
            }

            return Enumerable.Range(1, 5).Select(index => new WeatherForecast
            {
                Date = DateTime.Now.AddDays(index),
                TemperatureC = rng.Next(-20, 55),
                Summary = Summaries[rng.Next(Summaries.Length)]
            })
            .ToArray();
        }
    }
}
```

Troubleshoot

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE between any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in incognito or private mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in incognito or private mode. Some browsers require the URL of the app.
    - Microsoft Edge: `-inprivate`
    - Google Chrome: `--incognito --new-window https://localhost:5001`
    - Mozilla Firefox: `-private -url https://localhost:5001`
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE between any change to the app, test user, or provider configuration.

Run the Server app

When testing and troubleshooting a hosted Blazor app, make sure that you're running the app from the `Server` project. For example in Visual Studio, confirm that the Server project is highlighted in **Solution Explorer** before you start the app with any of the following approaches:

- Select the **Run** button.
- Use **Debug** > **Start Debugging** from the menu.
- Press F5.

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Additional resources

- Deployment to Azure App Service
- Import a certificate from Key Vault (Azure documentation)
- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client

ASP.NET Core Blazor WebAssembly additional security scenarios

9/22/2020 • 24 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

Attach tokens to outgoing requests

AuthorizationMessageHandler is a DelegatingHandler used to attach access tokens to outgoing HttpResponseMessage instances. Tokens are acquired using the IAccessTokenProvider service, which is registered by the framework. If a token can't be acquired, an AccessTokenNotAvailableException is thrown. AccessTokenNotAvailableException has a Redirect method that can be used to navigate the user to the identity provider to acquire a new token.

For convenience, the framework provides the BaseAddressAuthorizationMessageHandler preconfigured with the app's base address as an authorized URL. **Access tokens are only added when the request URI is within the app's base URI.** When outgoing request URIs aren't within the app's base URI, use a custom `AuthorizationMessageHandler` class (*recommended*) or configure the `AuthorizationMessageHandler`.

> **NOTE**
> In addition to the client app configuration for server API access, the server API must also allow cross-origin requests (CORS) when the client and the server don't reside at the same base address. For more information on server-side CORS configuration, see the Cross-origin resource sharing (CORS) section later in this article.

In the following example:

- AddHttpClient adds IHttpClientFactory and related services to the service collection and configures a named HttpClient (`ServerAPI`). HttpClient.BaseAddress is the base address of the resource URI when sending requests. IHttpClientFactory is provided by the `Microsoft.Extensions.Http` NuGet package.
- BaseAddressAuthorizationMessageHandler is the DelegatingHandler used to attach access tokens to outgoing HttpResponseMessage instances. Access tokens are only added when the request URI is within the app's base URI.
- IHttpClientFactory.CreateClient creates and configures an HttpClient instance for outgoing requests using the configuration that corresponds to the named HttpClient (`ServerAPI`).

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddHttpClient("ServerAPI",
        client => client.BaseAddress = new Uri("https://www.example.com/base"))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>()
    .CreateClient("ServerAPI"));
```

For a Blazor app based on the Blazor WebAssembly Hosted project template, request URIs are within the app's base URI by default. Therefore, IWebAssemblyHostEnvironment.BaseAddress (

`new Uri(builder.HostEnvironment.BaseAddress)`) is assigned to the HttpClient.BaseAddress in an app generated from the project template.

The configured HttpClient is used to make authorized requests using the `try-catch` pattern:

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject HttpClient Client

...

protected override async Task OnInitializedAsync()
{
    private ExampleType[] examples;

    try
    {
        examples =
            await Client.GetFromJsonAsync<ExampleType[]>("ExampleAPIMethod");

        ...
    }
    catch (AccessTokenNotAvailableException exception)
    {
        exception.Redirect();
    }
}
```

Custom `AuthorizationMessageHandler` class

This guidance in this section is recommended for client apps that make outgoing requests to URIs that aren't within the app's base URI.

In the following example, a custom class extends AuthorizationMessageHandler for use as the DelegatingHandler for an HttpClient. ConfigureHandler configures this handler to authorize outbound HTTP requests using an access token. The access token is only attached if at least one of the authorized URLs is a base of the request URI (HttpRequestMessage.RequestUri).

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class CustomAuthorizationMessageHandler : AuthorizationMessageHandler
{
    public CustomAuthorizationMessageHandler(IAccessTokenProvider provider,
        NavigationManager navigationManager)
        : base(provider, navigationManager)
    {
        ConfigureHandler(
            authorizedUrls: new[] { "https://www.example.com/base" },
            scopes: new[] { "example.read", "example.write" });
    }
}
```

In `Program.Main` (`Program.cs`), `CustomAuthorizationMessageHandler` is registered as a scoped service and is configured as the DelegatingHandler for outgoing HttpResponseMessage instances made by a named HttpClient:

```
builder.Services.AddScoped<CustomAuthorizationMessageHandler>();

builder.Services.AddHttpClient("ServerAPI",
        client => client.BaseAddress = new Uri("https://www.example.com/base"))
    .AddHttpMessageHandler<CustomAuthorizationMessageHandler>();
```

For a Blazor app based on the Blazor WebAssembly Hosted project template, IWebAssemblyHostEnvironment.BaseAddress (`new Uri(builder.HostEnvironment.BaseAddress)`) is assigned to the HttpClient.BaseAddress by default.

The configured HttpClient is used to make authorized requests using the `try-catch` pattern. Where the client is created with CreateClient (`Microsoft.Extensions.Http` package), the HttpClient is supplied instances that include access tokens when making requests to the server API. If the request URI is a relative URI, as it is in the following example (`ExampleAPIMethod`), it's combined with the BaseAddress when the client app makes the request:

```
@inject IHttpClientFactory ClientFactory

...

@code {
    private ExampleType[] examples;

    protected override async Task OnInitializedAsync()
    {
        try
        {
            var client = ClientFactory.CreateClient("ServerAPI");

            examples =
                await client.GetFromJsonAsync<ExampleType[]>("ExampleAPIMethod");
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
}
```

Configure `AuthorizationMessageHandler`

AuthorizationMessageHandler can be configured with authorized URLs, scopes, and a return URL using the ConfigureHandler method. ConfigureHandler configures the handler to authorize outbound HTTP requests using an access token. The access token is only attached if at least one of the authorized URLs is a base of the request URI (HttpRequestMessage.RequestUri). If the request URI is a relative URI, it's combined with the BaseAddress.

In the following example, AuthorizationMessageHandler configures an HttpClient in `Program.Main` (`Program.cs`):

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddScoped(sp => new HttpClient(
    sp.GetRequiredService<AuthorizationMessageHandler>()
    .ConfigureHandler(
        authorizedUrls: new[] { "https://www.example.com/base" },
        scopes: new[] { "example.read", "example.write" }))
    {
        BaseAddress = new Uri("https://www.example.com/base")
    });
```

For a Blazor app based on the Blazor WebAssembly Hosted project template, IWebAssemblyHostEnvironment.BaseAddress is assigned to the following by default:

- The HttpClient.BaseAddress (`new Uri(builder.HostEnvironment.BaseAddress)`).
- A URL of the `authorizedUrls` array.

Graph API example

In the following example, a named HttpClient for Graph API is used to obtain a user's mobile phone number to process a call. After adding the Microsoft Graph API `User.Read` permission in the AAD area of the Azure portal, the scope is configured for the named client in the standalone app or Client app of a hosted Blazor solution.

> **NOTE**
>
> The example in this section obtains Graph API data for the user in *component code*. To create user claims from Graph API, see the following resources:
>
> - Customize the user section
> - ASP.NET Core Blazor WebAssembly with Azure Active Directory groups and roles

`GraphAuthorizationMessageHandler.cs`:

```
public class GraphAPIAuthorizationMessageHandler : AuthorizationMessageHandler
{
    public GraphAPIAuthorizationMessageHandler(IAccessTokenProvider provider,
        NavigationManager navigationManager)
        : base(provider, navigationManager)
    {
        ConfigureHandler(
            authorizedUrls: new[] { "https://graph.microsoft.com" },
            scopes: new[] { "https://graph.microsoft.com/User.Read" });
    }
}
```

In `Program.Main` (`Program.cs`):

```
builder.Services.AddScoped<GraphAPIAuthorizationMessageHandler>();

builder.Services.AddHttpClient("GraphAPI",
        client => client.BaseAddress = new Uri("https://graph.microsoft.com"))
    .AddHttpMessageHandler<GraphAPIAuthorizationMessageHandler>();
```

In a Razor component (`Pages/CallUser.razor`):

```
@page "/CallUser"
```

```
@page "/CallUser"
@using System.ComponentModel.DataAnnotations
@using System.Text.Json.Serialization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Logging
@inject IAccessTokenProvider TokenProvider
@inject IHttpClientFactory ClientFactory
@inject ILogger<CallUser> Logger
@inject ICallProcessor CallProcessor

<h3>Call User</h3>

<EditForm Model="@callInfo" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p>
        <label>
            Message:
            <InputTextArea @bind-Value="callInfo.Message" />
        </label>
    </p>

    <button type="submit">Place call</button>

    <p>
        @formStatus
    </p>
</EditForm>

@code {
    private string formStatus;
    private CallInfo callInfo = new CallInfo();

    private async Task HandleValidSubmit()
    {
        var tokenResult = await TokenProvider.RequestAccessToken(
            new AccessTokenRequestOptions
            {
                Scopes = new[] { "https://graph.microsoft.com/User.Read" }
            });

        if (tokenResult.TryGetToken(out var token))
        {
            var client = ClientFactory.CreateClient("GraphAPI");

            var userInfo = await client.GetFromJsonAsync<UserInfo>("v1.0/me");

            if (userInfo != null)
            {
                CallProcessor.Send(userInfo.MobilePhone, callInfo.Message);

                formStatus = "Form successfully processed.";
                Logger.LogInformation(
                    $"Form successfully processed at {DateTime.UtcNow}. " +
                    $"Mobile Phone: {userInfo.MobilePhone}");
            }
        }
        else
        {
            formStatus = "There was a problem processing the form.";
            Logger.LogError("Token failure");
        }
    }

    private class CallInfo
    {
        [Required]
        [StringLength(1000, ErrorMessage = "Message too long (1,000 char limit)")]
```

```
        public string Message { get; set; }
    }

    private class UserInfo
    {
        [JsonPropertyName("mobilePhone")]
        public string MobilePhone { get; set; }
    }
}
```

> **NOTE**
>
> In the preceding example, the developer implements the custom `ICallProcessor` (`CallProcessor`) to queue and then place automated calls.

Typed `HttpClient`

A typed client can be defined that handles all of the HTTP and token acquisition concerns within a single class.

`WeatherForecastClient.cs`:

```
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using static {APP ASSEMBLY}.Data;

public class WeatherForecastClient
{
    private readonly HttpClient client;

    public WeatherForecastClient(HttpClient client)
    {
        this.client = client;
    }

    public async Task<WeatherForecast[]> GetForecastAsync()
    {
        var forecasts = new WeatherForecast[0];

        try
        {
            forecasts = await client.GetFromJsonAsync<WeatherForecast[]>(
                "WeatherForecast");
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }

        return forecasts;
    }
}
```

The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `using static BlazorSample.Data;`).

`Program.Main` (`Program.cs`):

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddHttpClient<WeatherForecastClient>(
        client => client.BaseAddress = new Uri("https://www.example.com/base"))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();
```

For a Blazor app based on the Blazor WebAssembly Hosted project template, IWebAssemblyHostEnvironment.BaseAddress (`new Uri(builder.HostEnvironment.BaseAddress)`) is assigned to the HttpClient.BaseAddress by default.

`FetchData` component (`Pages/FetchData.razor`):

```
@inject WeatherForecastClient Client

...

protected override async Task OnInitializedAsync()
{
    forecasts = await Client.GetForecastAsync();
}
```

Configure the `HttpClient` handler

The handler can be further configured with ConfigureHandler for outbound HTTP requests.

`Program.Main` (`Program.cs`):

```
builder.Services.AddHttpClient<WeatherForecastClient>(
        client => client.BaseAddress = new Uri("https://www.example.com/base"))
    .AddHttpMessageHandler(sp => sp.GetRequiredService<AuthorizationMessageHandler>()
    .ConfigureHandler(
        authorizedUrls: new [] { "https://www.example.com/base" },
        scopes: new[] { "example.read", "example.write" }));
```

For a Blazor app based on the Blazor WebAssembly Hosted project template, IWebAssemblyHostEnvironment.BaseAddress is assigned to the following by default:

- The HttpClient.BaseAddress (`new Uri(builder.HostEnvironment.BaseAddress)`).
- A URL of the `authorizedUrls` array.

Unauthenticated or unauthorized web API requests in an app with a secure default client

If the Blazor WebAssembly app ordinarily uses a secure default HttpClient, the app can also make unauthenticated or unauthorized web API requests by configuring a named HttpClient:

`Program.Main` (`Program.cs`):

```
builder.Services.AddHttpClient("ServerAPI.NoAuthenticationClient",
    client => client.BaseAddress = new Uri("https://www.example.com/base"));
```

For a Blazor app based on the Blazor WebAssembly Hosted project template,

IWebAssemblyHostEnvironment.BaseAddress (`new Uri(builder.HostEnvironment.BaseAddress)`) is assigned to the HttpClient.BaseAddress by default.

The preceding registration is in addition to the existing secure default HttpClient registration.

A component creates the HttpClient from the IHttpClientFactory (`Microsoft.Extensions.Http` package) to make unauthenticated or unauthorized requests:

```
@inject IHttpClientFactory ClientFactory

...

@code {
    private WeatherForecast[] forecasts;

    protected override async Task OnInitializedAsync()
    {
        var client = ClientFactory.CreateClient("ServerAPI.NoAuthenticationClient");

        forecasts = await client.GetFromJsonAsync<WeatherForecast[]>(
            "WeatherForecastNoAuthentication");
    }
}
```

> **NOTE**
>
> The controller in the server API, `WeatherForecastNoAuthenticationController` for the preceding example, isn't marked with the `[Authorize]` attribute.

The decision whether to use a secure client or an insecure client as the default HttpClient instance is up to the developer. One way to make this decision is to consider the number of authenticated versus unauthenticated endpoints that the app contacts. If the majority of the app's requests are to secure API endpoints, use the authenticated HttpClient instance as the default. Otherwise, register the unauthenticated HttpClient instance as the default.

An alternative approach to using the IHttpClientFactory is to create a typed client for unauthenticated access to anonymous endpoints.

Request additional access tokens

Access tokens can be manually obtained by calling `IAccessTokenProvider.RequestAccessToken`. In the following example, an additional scope is required by an app for the default HttpClient. The Microsoft Authentication Library (MSAL) example configures the scope with `MsalProviderOptions`:

`Program.Main` (`Program.cs`):

```
builder.Services.AddMsalAuthentication(options =>
{
    ...

    options.ProviderOptions.AdditionalScopesToConsent.Add("{CUSTOM SCOPE 1}");
    options.ProviderOptions.AdditionalScopesToConsent.Add("{CUSTOM SCOPE 2}");
}
```

The `{CUSTOM SCOPE 1}` and `{CUSTOM SCOPE 2}` placeholders in the preceding example are custom scopes.

The `IAccessTokenProvider.RequestToken` method provides an overload that allows an app to provision an access

token with a given set of scopes.

In a Razor component:

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IAccessTokenProvider TokenProvider

...

var tokenResult = await TokenProvider.RequestAccessToken(
    new AccessTokenRequestOptions
    {
        Scopes = new[] { "{CUSTOM SCOPE 1}", "{CUSTOM SCOPE 2}" }
    });

if (tokenResult.TryGetToken(out var token))
{
    ...
}
```

The `{CUSTOM SCOPE 1}` and `{CUSTOM SCOPE 2}` placeholders in the preceding example are custom scopes.

AccessTokenResult.TryGetToken returns:

- `true` with the `token` for use.
- `false` if the token isn't retrieved.

Cross-origin resource sharing (CORS)

When sending credentials (authorization cookies/headers) on CORS requests, the `Authorization` header must be allowed by the CORS policy.

The following policy includes configuration for:

- Request origins (`http://localhost:5000`, `https://localhost:5001`).
- Any method (verb).
- `Content-Type` and `Authorization` headers. To allow a custom header (for example, `x-custom-header`), list the header when calling WithHeaders.
- Credentials set by client-side JavaScript code (`credentials` property set to `include`).

```
app.UseCors(policy =>
    policy.WithOrigins("http://localhost:5000", "https://localhost:5001")
    .AllowAnyMethod()
    .WithHeaders(HeaderNames.ContentType, HeaderNames.Authorization, "x-custom-header")
    .AllowCredentials());
```

A hosted Blazor solution based on the Blazor Hosted project template uses the same base address for the client and server apps. The client app's HttpClient.BaseAddress is set to a URI of `builder.HostEnvironment.BaseAddress` by default. CORS configuration is **not** required in the default configuration of a hosted app created from the Blazor Hosted project template. Additional client apps that aren't hosted by the server project and don't share the server app's base address **do** require CORS configuration in the server project.

For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core and the sample app's HTTP Request Tester component (`Components/HTTPRequestTester.razor`).

Handle token request errors

When a Single Page Application (SPA) authenticates a user using OpenID Connect (OIDC), the authentication

state is maintained locally within the SPA and in the Identity Provider (IP) in the form of a session cookie that's set as a result of the user providing their credentials.

The tokens that the IP emits for the user typically are valid for short periods of time, about one hour normally, so the client app must regularly fetch new tokens. Otherwise, the user would be logged-out after the granted tokens expire. In most cases, OIDC clients are able to provision new tokens without requiring the user to authenticate again thanks to the authentication state or "session" that is kept within the IP.

There are some cases in which the client can't get a token without user interaction, for example, when for some reason the user explicitly logs out from the IP. This scenario occurs if a user visits `https://login.microsoftonline.com` and logs out. In these scenarios, the app doesn't know immediately that the user has logged out. Any token that the client holds might no longer be valid. Also, the client isn't able to provision a new token without user interaction after the current token expires.

These scenarios aren't specific to token-based authentication. They are part of the nature of SPAs. An SPA using cookies also fails to call a server API if the authentication cookie is removed.

When an app performs API calls to protected resources, you must be aware of the following:

- To provision a new access token to call the API, the user might be required to authenticate again.
- Even if the client has a token that seems to be valid, the call to the server might fail because the token was revoked by the user.

When the app requests a token, there are two possible outcomes:

- The request succeeds, and the app has a valid token.
- The request fails, and the app must authenticate the user again to obtain a new token.

When a token request fails, you need to decide whether you want to save any current state before you perform a redirection. Several approaches exist with increasing levels of complexity:

- Store the current page state in session storage. During the `OnInitializedAsync` lifecycle event (OnInitializedAsync), check if state can be restored before continuing.
- Add a query string parameter and use that as a way to signal the app that it needs to re-hydrate the previously saved state.
- Add a query string parameter with a unique identifier to store data in session storage without risking collisions with other items.

The following example shows how to:

- Preserve state before redirecting to the login page.
- Recover the previous state afterward authentication using the query string parameter.

```
<EditForm Model="User" @onsubmit="OnSaveAsync">
    <label>User
        <InputText @bind-Value="User.Name" />
    </label>
    <label>Last name
        <InputText @bind-Value="User.LastName" />
    </label>
</EditForm>

@code {
    public class Profile
    {
        public string Name { get; set; }
        public string LastName { get; set; }
    }

    public Profile User { get; set; } = new Profile();

    protected async override Task OnInitializedAsync()
    {
        var currentQuery = new Uri(Navigation.Uri).Query;

        if (currentQuery.Contains("state=resumeSavingProfile"))
        {
            User = await JS.InvokeAsync<Profile>("sessionStorage.getState",
                "resumeSavingProfile");
        }
    }

    public async Task OnSaveAsync()
    {
        var httpClient = new HttpClient();
        httpClient.BaseAddress = new Uri(Navigation.BaseUri);

        var resumeUri = Navigation.Uri + $"?state=resumeSavingProfile";

        var tokenResult = await AuthenticationService.RequestAccessToken(
            new AccessTokenRequestOptions
            {
                ReturnUrl = resumeUri
            });

        if (tokenResult.TryGetToken(out var token))
        {
            httpClient.DefaultRequestHeaders.Add("Authorization",
                $"Bearer {token.Value}");
            await httpClient.PostAsJsonAsync("Save", User);
        }
        else
        {
            await JS.InvokeVoidAsync("sessionStorage.setState",
                "resumeSavingProfile", User);
            Navigation.NavigateTo(tokenResult.RedirectUrl);
        }
    }
}
```

Save app state before an authentication operation

During an authentication operation, there are cases where you want to save the app state before the browser is redirected to the IP. This can be the case when you're using a state container and want to restore the state after the authentication succeeds. You can use a custom authentication state object to preserve app-specific state or a reference to it and restore that state after the authentication operation successfully completes. The following example demonstrates the approach.

A state container class is created in the app with properties to hold the app's state values. In the following example, the container is used to maintain the counter value of the default project template's `Counter` component (`Pages/Counter.razor`). Methods for serializing and deserializing the container are based on System.Text.Json.

```
using System.Text.Json;

public class StateContainer
{
    public int CounterValue { get; set; }

    public string GetStateForLocalStorage()
    {
        return JsonSerializer.Serialize(this);
    }

    public void SetStateFromLocalStorage(string locallyStoredState)
    {
        var deserializedState =
            JsonSerializer.Deserialize<StateContainer>(locallyStoredState);

        CounterValue = deserializedState.CounterValue;
    }
}
```

The `Counter` component uses the state container to maintain the `currentCount` value outside of the component:

```
@page "/counter"
@inject StateContainer State

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    protected override void OnInitialized()
    {
        if (State.CounterValue > 0)
        {
            currentCount = State.CounterValue;
        }
    }

    private void IncrementCount()
    {
        currentCount++;
        State.CounterValue = currentCount;
    }
}
```

Create an `ApplicationAuthenticationState` from RemoteAuthenticationState. Provide an `Id` property, which serves as an identifier for the locally-stored state.

`ApplicationAuthenticationState.cs`:

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class ApplicationAuthenticationState : RemoteAuthenticationState
{
    public string Id { get; set; }
}
```

The `Authentication` component (`Pages/Authentication.razor`) saves and restores the app's state using local session storage with the `StateContainer` serialization and deserialization methods, `GetStateForLocalStorage` and `SetStateFromLocalStorage`:

```
@page "/authentication/{action}"
@inject IJSRuntime JS
@inject StateContainer State
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorViewCore Action="@Action"
                             TAuthenticationState="ApplicationAuthenticationState"
                             AuthenticationState="AuthenticationState"
                             OnLogInSucceeded="RestoreState"
                             OnLogOutSucceeded="RestoreState" />

@code {
    [Parameter]
    public string Action { get; set; }

    public ApplicationAuthenticationState AuthenticationState { get; set; } =
        new ApplicationAuthenticationState();

    protected async override Task OnInitializedAsync()
    {
        if (RemoteAuthenticationActions.IsAction(RemoteAuthenticationActions.LogIn,
            Action) ||
            RemoteAuthenticationActions.IsAction(RemoteAuthenticationActions.LogOut,
            Action))
        {
            AuthenticationState.Id = Guid.NewGuid().ToString();

            await JS.InvokeVoidAsync("sessionStorage.setItem",
                AuthenticationState.Id, State.GetStateForLocalStorage());
        }
    }

    private async Task RestoreState(ApplicationAuthenticationState state)
    {
        if (state.Id != null)
        {
            var locallyStoredState = await JS.InvokeAsync<string>(
                "sessionStorage.getItem", state.Id);

            if (locallyStoredState != null)
            {
                State.SetStateFromLocalStorage(locallyStoredState);
                await JS.InvokeVoidAsync("sessionStorage.removeItem", state.Id);
            }
        }
    }
}
```

This example uses Azure Active Directory (AAD) for authentication. In `Program.Main` (`Program.cs`):

- The `ApplicationAuthenticationState` is configured as the Microsoft Autentication Library (MSAL) `RemoteAuthenticationState` type.

- The state container is registered in the service container.

```
builder.Services.AddMsalAuthentication<ApplicationAuthenticationState>(options =>
{
    builder.Configuration.Bind("AzureAd", options.ProviderOptions.Authentication);
});

builder.Services.AddSingleton<StateContainer>();
```

Customize app routes

By default, the `Microsoft.AspNetCore.Components.WebAssembly.Authentication` library uses the routes shown in the following table for representing different authentication states.

ROUTE	PURPOSE
`authentication/login`	Triggers a sign-in operation.
`authentication/login-callback`	Handles the result of any sign-in operation.
`authentication/login-failed`	Displays error messages when the sign-in operation fails for some reason.
`authentication/logout`	Triggers a sign-out operation.
`authentication/logout-callback`	Handles the result of a sign-out operation.
`authentication/logout-failed`	Displays error messages when the sign-out operation fails for some reason.
`authentication/logged-out`	Indicates that the user has successfully logout.
`authentication/profile`	Triggers an operation to edit the user profile.
`authentication/register`	Triggers an operation to register a new user.

The routes shown in the preceding table are configurable via RemoteAuthenticationOptions<TRemoteAuthenticationProviderOptions>.AuthenticationPaths. When setting options to provide custom routes, confirm that the app has a route that handles each path.

In the following example, all the paths are prefixed with `/security`.

`Authentication` component (`Pages/Authentication.razor`):

```
@page "/security/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code{
    [Parameter]
    public string Action { get; set; }
}
```

`Program.Main` (`Program.cs`):

```
builder.Services.AddApiAuthorization(options => {
    options.AuthenticationPaths.LogInPath = "security/login";
    options.AuthenticationPaths.LogInCallbackPath = "security/login-callback";
    options.AuthenticationPaths.LogInFailedPath = "security/login-failed";
    options.AuthenticationPaths.LogOutPath = "security/logout";
    options.AuthenticationPaths.LogOutCallbackPath = "security/logout-callback";
    options.AuthenticationPaths.LogOutFailedPath = "security/logout-failed";
    options.AuthenticationPaths.LogOutSucceededPath = "security/logged-out";
    options.AuthenticationPaths.ProfilePath = "security/profile";
    options.AuthenticationPaths.RegisterPath = "security/register";
});
```

If the requirement calls for completely different paths, set the routes as described previously and render the RemoteAuthenticatorView with an explicit action parameter:

```
@page "/register"

<RemoteAuthenticatorView Action="@RemoteAuthenticationActions.Register" />
```

You're allowed to break the UI into different pages if you choose to do so.

Customize the authentication user interface

RemoteAuthenticatorView includes a default set of UI pieces for each authentication state. Each state can be customized by passing in a custom RenderFragment. To customize the displayed text during the initial login process, can change the RemoteAuthenticatorView as follows.

`Authentication` component (`Pages/Authentication.razor`):

```
@page "/security/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action">
    <LoggingIn>
        You are about to be redirected to https://login.microsoftonline.com.
    </LoggingIn>
</RemoteAuthenticatorView>

@code{
    [Parameter]
    public string Action { get; set; }
}
```

The RemoteAuthenticatorView has one fragment that can be used per authentication route shown in the following table.

ROUTE	FRAGMENT
`authentication/login`	`<LoggingIn>`
`authentication/login-callback`	`<CompletingLoggingIn>`
`authentication/login-failed`	`<LogInFailed>`
`authentication/logout`	`<LogOut>`

ROUTE	FRAGMENT
`authentication/logout-callback`	`<CompletingLogOut>`
`authentication/logout-failed`	`<LogOutFailed>`
`authentication/logged-out`	`<LogOutSucceeded>`
`authentication/profile`	`<UserProfile>`
`authentication/register`	`<Registering>`

Customize the user

Users bound to the app can be customized.

Customize the user with a payload claim

In the following example, the app's authenticated users receive an `amr` claim for each of the user's authentication methods. The `amr` claim identifies how the subject of the token was authenticated in Microsoft Identity Platform v1.0 payload claims. The example uses a custom user account class based on RemoteUserAccount.

Create a class that extends the RemoteUserAccount class. The following example sets the `AuthenticationMethod` property to the user's array of `amr` JSON property values. `AuthenticationMethod` is populated automatically by the framework when the user is authenticated.

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class CustomUserAccount : RemoteUserAccount
{
    [JsonPropertyName("amr")]
    public string[] AuthenticationMethod { get; set; }
}
```

Create a factory that extends AccountClaimsPrincipalFactory<TAccount> to create claims from the user's authentication methods stored in `CustomUserAccount.AuthenticationMethod`:

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;

public class CustomAccountFactory
    : AccountClaimsPrincipalFactory<CustomUserAccount>
{
    public CustomAccountFactory(NavigationManager navigationManager,
        IAccessTokenProviderAccessor accessor) : base(accessor)
    {
    }

    public async override ValueTask<ClaimsPrincipal> CreateUserAsync(
        CustomUserAccount account, RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity.IsAuthenticated)
        {
            foreach (var value in account.AuthenticationMethod)
            {
                ((ClaimsIdentity)initialUser.Identity)
                    .AddClaim(new Claim("amr", value));
            }
        }

        return initialUser;
    }
}
```

Register the `CustomAccountFactory` for the authentication provider in use. Any of the following registrations are valid:

- AddOidcAuthentication:

  ```
  using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

  ...

  builder.Services.AddOidcAuthentication<RemoteAuthenticationState,
      CustomUserAccount>(options =>
      {
          ...
      })
      .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
          CustomUserAccount, CustomAccountFactory>();
  ```

- AddMsalAuthentication:

  ```
  using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

  ...

  builder.Services.AddMsalAuthentication<RemoteAuthenticationState,
      CustomUserAccount>(options =>
      {
          ...
      })
      .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
          CustomUserAccount, CustomAccountFactory>();
  ```

- AddApiAuthorization:

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddApiAuthorization<RemoteAuthenticationState,
    CustomUserAccount>(options =>
    {
        ...
    })
    .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
        CustomUserAccount, CustomAccountFactory>();
```

Customize the user with Graph API claims

In the following example, the app creates a mobile phone number claim for the user from Graph API using the RemoteUserAccount. The app must have the `User.Read` Graph API permission (scope) configured in AAD.

`GraphAuthorizationMessageHandler.cs`:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class GraphAPIAuthorizationMessageHandler : AuthorizationMessageHandler
{
    public GraphAPIAuthorizationMessageHandler(IAccessTokenProvider provider,
        NavigationManager navigationManager)
        : base(provider, navigationManager)
    {
        ConfigureHandler(
            authorizedUrls: new[] { "https://graph.microsoft.com" },
            scopes: new[] { "https://graph.microsoft.com/User.Read" });
    }
}
```

A named HttpClient for Graph API is created in `Program.Main` (`Program.cs`) using the `GraphAPIAuthorizationMessageHandler`:

```
using System;

...

builder.Services.AddScoped<GraphAPIAuthorizationMessageHandler>();

builder.Services.AddHttpClient("GraphAPI",
        client => client.BaseAddress = new Uri("https://graph.microsoft.com"))
    .AddHttpMessageHandler<GraphAPIAuthorizationMessageHandler>();
```

`Models/UserInfo.cs`:

```
using System.Text.Json.Serialization;

public class UserInfo
{
    [JsonPropertyName("mobilePhone")]
    public string MobilePhone { get; set; }
}
```

In the following `CustomAccountFactory` (`CustomAccountFactory.cs`), the framework's RemoteUserAccount

represents the user's account. If the app requires a custom user account class that extends RemoteUserAccount, swap the custom user account class for RemoteUserAccount in the following code:

```
using System.Net.Http;
using System.Net.Http.Json;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;
using Microsoft.Extensions.Logging;

public class CustomAccountFactory
    : AccountClaimsPrincipalFactory<RemoteUserAccount>
{
    private readonly ILogger<CustomAccountFactory> logger;
    private readonly IHttpClientFactory clientFactory;

    public CustomAccountFactory(IAccessTokenProviderAccessor accessor,
        IHttpClientFactory clientFactory,
        ILogger<CustomAccountFactory> logger)
        : base(accessor)
    {
        this.clientFactory = clientFactory;
        this.logger = logger;
    }

    public async override ValueTask<ClaimsPrincipal> CreateUserAsync(
        RemoteUserAccount account,
        RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity.IsAuthenticated)
        {
            var userIdentity = (ClaimsIdentity)initialUser.Identity;

            try
            {
                var client = clientFactory.CreateClient("GraphAPI");

                var userInfo = await client.GetFromJsonAsync<UserInfo>("v1.0/me");

                if (userInfo != null)
                {
                    userIdentity.AddClaim(new Claim("mobilephone", userInfo.MobilePhone));
                }
            }
            catch (AccessTokenNotAvailableException exception)
            {
                logger.LogError("Graph API access token failure: {MESSAGE}",
                    exception.Message);
            }
        }

        return initialUser;
    }
}
```

In `Program.Main` (`Program.cs`), configure the app to use the custom factory. If the app uses a custom user account class that extends RemoteUserAccount, swap the custom user account class for RemoteUserAccount in the following code:

```
using Microsoft.Extensions.Configuration;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddMsalAuthentication<RemoteAuthenticationState,
    RemoteUserAccount>(options =>
    {
        builder.Configuration.Bind("AzureAd",
            options.ProviderOptions.Authentication);
    })
    .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState, RemoteUserAccount,
        CustomAccountFactory>();
```

The preceding example is for an app that uses AAD authentication with MSAL. Similar patterns exist for OIDC and API authentication. For more information, see the examples at the end of the Customize the user with a payload claim section.

AAD security groups and roles with a custom user account class

For an additional example that works with AAD security groups and AAD Administrator Roles and a custom user account class, see ASP.NET Core Blazor WebAssembly with Azure Active Directory groups and roles.

Support prerendering with authentication

After following the guidance in one of the hosted Blazor WebAssembly app topics, use the following instructions to create an app that:

- Prerenders paths for which authorization isn't required.
- Doesn't prerender paths for which authorization is required.

In the Client app's `Program` class (`Program.cs`), factor common service registrations into a separate method (for example, `ConfigureCommonServices`):

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var builder = WebAssemblyHostBuilder.CreateDefault(args);
        builder.RootComponents.Add<App>("app");

        builder.Services.AddScoped(sp =>
            new HttpClient
            {
                BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
            });

        services.Add...;

        ConfigureCommonServices(builder.Services);

        await builder.Build().RunAsync();
    }

    public static void ConfigureCommonServices(IServiceCollection services)
    {
        // Common service registrations
    }
}
```

In the server app's `Startup.ConfigureServices` , register the following additional services:

```
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Components.Server;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddRazorPages();
    services.AddScoped<AuthenticationStateProvider,
        ServerAuthenticationStateProvider>();
    services.AddScoped<SignOutSessionStateManager>();

    Client.Program.ConfigureCommonServices(services);
}
```

In the server app's `Startup.Configure` method, replace `endpoints.MapFallbackToFile("index.html")` with `endpoints.MapFallbackToPage("/_Host")`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapFallbackToPage("/_Host");
});
```

In the server app, create a `Pages` folder if it doesn't exist. Create a `_Host.cshtml` page inside the server app's `Pages` folder. Paste the contents from the Client app's `wwwroot/index.html` file into the `Pages/_Host.cshtml` file. Update the file's contents:

- Add `@page "_Host"` to the top of the file.
- Replace the `<app>Loading...</app>` tag with the following:

```
<app>
    @if (!HttpContext.Request.Path.StartsWithSegments("/authentication"))
    {
        <component type="typeof(Wasm.Authentication.Client.App)"
            render-mode="Static" />
    }
    else
    {
        <text>Loading...</text>
    }
</app>
```

Options for hosted apps and third-party login providers

When authenticating and authorizing a hosted Blazor WebAssembly app with a third-party provider, there are several options available for authenticating the user. Which one you choose depends on your scenario.

For more information, see Persist additional claims and tokens from external providers in ASP.NET Core.

Authenticate users to only call protected third party APIs

Authenticate the user with a client-side OAuth flow against the third-party API provider:

```
builder.services.AddOidcAuthentication(options => { ... });
```

In this scenario:

- The server hosting the app doesn't play a role.
- APIs on the server can't be protected.
- The app can only call protected third-party APIs.

Authenticate users with a third-party provider and call protected APIs on the host server and the third party

Configure Identity with a third-party login provider. Obtain the tokens required for third-party API access and store them.

When a user logs in, Identity collects access and refresh tokens as part of the authentication process. At that point, there are a couple of approaches available for making API calls to third-party APIs.

Use a server access token to retrieve the third-party access token

Use the access token generated on the server to retrieve the third-party access token from a server API endpoint. From there, use the third-party access token to call third-party API resources directly from Identity on the client.

We don't recommend this approach. This approach requires treating the third-party access token as if it were generated for a public client. In OAuth terms, the public app doesn't have a client secret because it can't be trusted to store secrets safely, and the access token is produced for a confidential client. A confidential client is a client that has a client secret and is assumed to be able to safely store secrets.

- The third-party access token might be granted additional scopes to perform sensitive operations based on the fact that the third-party emitted the token for a more trusted client.
- Similarly, refresh tokens shouldn't be issued to a client that isn't trusted, as doing so gives the client unlimited access unless other restrictions are put into place.

Make API calls from the client to the server API in order to call third-party APIs

Make an API call from the client to the server API. From the server, retrieve the access token for the third-party API resource and issue whatever call is necessary.

While this approach requires an extra network hop through the server to call a third-party API, it ultimately results in a safer experience:

- The server can store refresh tokens and ensure that the app doesn't lose access to third-party resources.
- The app can't leak access tokens from the server that might contain more sensitive permissions.

Use OpenID Connect (OIDC) v2.0 endpoints

The authentication library and Blazor project templates use OpenID Connect (OIDC) v1.0 endpoints. To use a v2.0 endpoint, configure the JWT Bearer JwtBearerOptions.Authority option. In the following example, AAD is configured for v2.0 by appending a `v2.0` segment to the Authority property:

```
builder.Services.Configure<JwtBearerOptions>(
    AzureADDefaults.JwtBearerAuthenticationScheme,
    options =>
    {
        options.Authority += "/v2.0";
    });
```

Alternatively, the setting can be made in the app settings (`appsettings.json`) file:

```
{
  "Local": {
    "Authority": "https://login.microsoftonline.com/common/oauth2/v2.0/",
    ...
  }
}
```

If tacking on a segment to the authority isn't appropriate for the app's OIDC provider, such as with non-AAD providers, set the Authority property directly. Either set the property in JwtBearerOptions or in the app settings file (`appsettings.json`) with the `Authority` key.

The list of claims in the ID token changes for v2.0 endpoints. For more information, see Why update to Microsoft identity platform (v2.0)?.

Configure and use gRPC in components

To configure a Blazor WebAssembly app to use the ASP.NET Core gRPC framework:

- Enable gRPC-Web on the server. For more information, see Use gRPC in browser apps.
- Register gRPC services for the app's message handler. The following example configures the app's authorization message handler to use the `GreeterClient` service from the gRPC tutorial (`Program.Main`):

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Grpc.Net.Client;
using Grpc.Net.Client.Web;
using {APP ASSEMBLY}.Shared;

...

builder.Services.AddScoped(sp =>
{
    var baseAddressMessageHandler =
        sp.GetRequiredService<BaseAddressAuthorizationMessageHandler>();
    baseAddressMessageHandler.InnerHandler = new HttpClientHandler();
    var grpcWebHandler =
        new GrpcWebHandler(GrpcWebMode.GrpcWeb, baseAddressMessageHandler);
    var channel = GrpcChannel.ForAddress(builder.HostEnvironment.BaseAddress,
        new GrpcChannelOptions { HttpHandler = grpcWebHandler });

    return new Greeter.GreeterClient(channel);
});
```

The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `BlazorSample`). Place the `.proto` file in the `Shared` project of the hosted Blazor solution.

A component in the client app can make gRPC calls using the gRPC client (`Pages/Grpc.razor`):

```
@page "/grpc"
@attribute [Authorize]
@using Microsoft.AspNetCore.Authorization
@using {APP ASSEMBLY}.Shared
@inject Greeter.GreeterClient GreeterClient

<h1>Invoke gRPC service</h1>

<p>
    <input @bind="name" placeholder="Type your name" />
    <button @onclick="GetGreeting" class="btn btn-primary">Call gRPC service</button>
</p>

Server response: <strong>@serverResponse</strong>

@code {
    private string name = "Bert";
    private string serverResponse;

    private async Task GetGreeting()
    {
        try
        {
            var request = new HelloRequest { Name = name };
            var reply = await GreeterClient.SayHelloAsync(request);
            serverResponse = reply.Message;
        }
        catch (Grpc.Core.RpcException ex)
            when (ex.Status.DebugException is
                AccessTokenNotAvailableException tokenEx)
        {
            tokenEx.Redirect();
        }
    }
}
```

The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `BlazorSample`). To use the `Status.DebugException` property, use Grpc.Net.Client version 2.30.0 or later.

For more information, see Use gRPC in browser apps.

Additional resources

- `HttpClient` and `HttpRequestMessage` with Fetch API request options

Azure AD Groups, Administrative Roles, and user-defined roles

9/22/2020 • 8 minutes to read • Edit Online

By Luke Latham and Javier Calvarro Nelson

Azure Active Directory (AAD) provides several authorization approaches that can be combined with ASP.NET Core Identity:

- User-defined groups
  - Security
  - Microsoft 365
  - Distribution
- Roles
  - Built-in Administrative Roles
  - User-defined roles

The guidance in this article applies to the Blazor WebAssembly AAD deployment scenarios described in the following topics:

- Standalone with Microsoft Accounts
- Standalone with AAD
- Hosted with AAD

Microsoft Graph API permission

A Microsoft Graph API call is required for any app user with more than five built-in AAD Administrator role and security group memberships.

To permit Graph API calls, give the standalone or client app of a hosted Blazor solution any of the following Graph API permissions in the Azure portal:

- `Directory.Read.All`
- `Directory.ReadWrite.All`
- `Directory.AccessAsUser.All`

`Directory.Read.All` is the least-privileged permission and is the permission used for the example described in this article.

User-defined groups and built-in Administrative Roles

To configure the app in the Azure portal to provide a `groups` membership claim, see the following Azure articles. Assign users to user-defined AAD groups and built-in Administrative Roles.

- Roles using Azure AD security groups
- `groupMembershipClaims` attribute

The following examples assume that a user is assigned to the AAD built-in *Billing Administrator* role.

The single `groups` claim sent by AAD presents the user's groups and roles as Object IDs (GUIDs) in a JSON array.

The app must convert the JSON array of groups and roles into individual `group` claims that the app can build policies against.

When the number of assigned built-in Azure Administrative Roles and user-defined groups exceeds five, AAD sends a `hasgroups` claim with a `true` value instead of sending a `groups` claim. Any app that may have more than five roles and groups assigned to its users must make a separate Graph API call to obtain a user's roles and groups. The example implementation provided in this article addresses this scenario. For more information, see the `groups` and `hasgroups` claims information in Microsoft identity platform access tokens: Payload claims article.

Extend RemoteUserAccount to include array properties for groups and roles. Assign an empty array to each property so that checking for `null` isn't required when these properties are used in `foreach` loops later.

`CustomUserAccount.cs`:

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class CustomUserAccount : RemoteUserAccount
{
    [JsonPropertyName("groups")]
    public string[] Groups { get; set; } = new string[] { };

    [JsonPropertyName("roles")]
    public string[] Roles { get; set; } = new string[] { };
}
```

In the standalone app or the client app of a hosted Blazor solution, create a custom AuthorizationMessageHandler class. Use the correct scope (permission) for Graph API calls that obtain role and group information.

`GraphAPIAuthorizationMessageHandler.cs`:

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class GraphAPIAuthorizationMessageHandler : AuthorizationMessageHandler
{
    public GraphAPIAuthorizationMessageHandler(IAccessTokenProvider provider,
        NavigationManager navigationManager)
        : base(provider, navigationManager)
    {
        ConfigureHandler(
            authorizedUrls: new[] { "https://graph.microsoft.com" },
            scopes: new[] { "https://graph.microsoft.com/Directory.Read.All" });
    }
}
```

In `Program.Main` (`Program.cs`), add the AuthorizationMessageHandler implementation service and add a named HttpClient for making Graph API requests. The following example names the client `GraphAPI`:

```
builder.Services.AddScoped<GraphAPIAuthorizationMessageHandler>();

builder.Services.AddHttpClient("GraphAPI",
        client => client.BaseAddress = new Uri("https://graph.microsoft.com"))
    .AddHttpMessageHandler<GraphAPIAuthorizationMessageHandler>();
```

Create AAD directory objects classes to receive the Open Data Protocol (OData) roles and groups from a Graph API call. The OData arrives in JSON format, and a call to ReadFromJsonAsync populates an instance of the `DirectoryObjects` class.

`DirectoryObjects.cs`:

```
using System.Collections.Generic;
using System.Text.Json.Serialization;

public class DirectoryObjects
{
    [JsonPropertyName("@odata.context")]
    public string Context { get; set; }

    [JsonPropertyName("value")]
    public List<Value> Values { get; set; }
}

public class Value
{
    [JsonPropertyName("@odata.type")]
    public string Type { get; set; }

    [JsonPropertyName("id")]
    public string Id { get; set; }
}
```

Create a custom user factory to process roles and groups claims. The following example implementation also handles the `roles` claim array, which is covered in the User-defined roles section. If the `hasgroups` claim is present, the named HttpClient is used to make an authorized request to Graph API to obtain the user's roles and groups. This implementation uses the Microsoft Identity Platform v1.0 endpoint `https://graph.microsoft.com/v1.0/me/memberOf` (API documentation). The guidance in this topic will be updated for Identity v2.0 when the MSAL packages are upgraded for v2.0.

`CustomAccountFactory.cs`:

```
using System.Linq;
using System.Net.Http;
using System.Net.Http.Json;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;
using Microsoft.Extensions.Logging;

public class CustomUserFactory
    : AccountClaimsPrincipalFactory<CustomUserAccount>
{
    private readonly ILogger<CustomUserFactory> logger;
    private readonly IHttpClientFactory clientFactory;

    public CustomUserFactory(IAccessTokenProviderAccessor accessor,
        IHttpClientFactory clientFactory,
        ILogger<CustomUserFactory> logger)
        : base(accessor)
    {
        this.clientFactory = clientFactory;
        this.logger = logger;
    }

    public async override ValueTask<ClaimsPrincipal> CreateUserAsync(
        CustomUserAccount account,
        RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity.IsAuthenticated)
        {
```

```
            var userIdentity = (ClaimsIdentity)initialUser.Identity;

            foreach (var role in account.Roles)
            {
                userIdentity.AddClaim(new Claim("role", role));
            }

            if (userIdentity.HasClaim(c => c.Type == "hasgroups"))
            {
                try
                {
                    var client = clientFactory.CreateClient("GraphAPI");

                    var response = await client.GetAsync("v1.0/me/memberOf");

                    if (response.IsSuccessStatusCode)
                    {
                        var userObjects = await response.Content
                            .ReadFromJsonAsync<DirectoryObjects>();

                        foreach (var obj in userObjects?.Values)
                        {
                            userIdentity.AddClaim(new Claim("group", obj.Id));
                        }

                        var claim = userIdentity.Claims.FirstOrDefault(
                            c => c.Type == "hasgroups");

                        userIdentity.RemoveClaim(claim);
                    }
                    else
                    {
                        logger.LogError("Graph API request failure: {REASON}",
                            response.ReasonPhrase);
                    }
                }
                catch (AccessTokenNotAvailableException exception)
                {
                    logger.LogError("Graph API access token failure: {MESSAGE}",
                        exception.Message);
                }
            }
            else
            {
                foreach (var group in account.Groups)
                {
                    userIdentity.AddClaim(new Claim("group", group));
                }
            }
        }

        return initialUser;
    }
}
```

There's no need to provide code to remove the original `groups` claim, if present, because it's automatically removed by the framework.

NOTE

The approach in this example:

- Adds a custom AuthorizationMessageHandler class to attach access tokens to outgoing requests.
- Adds a named HttpClient for making web API requests to a secure, external web API endpoint.
- Uses the named HttpClient to make authorized requests.

General coverage for this approach is found in the ASP.NET Core Blazor WebAssembly additional security scenarios article.

Register the factory in `Program.Main` (`Program.cs`) of the standalone app or client app of a hosted Blazor solution. Consent to the `Directory.Read.All` permission scope as an additional scope for the app:

```
builder.Services.AddMsalAuthentication<RemoteAuthenticationState,
    CustomUserAccount>(options =>
{
    builder.Configuration.Bind("AzureAd",
        options.ProviderOptions.Authentication);
    options.ProviderOptions.DefaultAccessTokenScopes.Add("...");

    options.ProviderOptions.AdditionalScopesToConsent.Add(
        "https://graph.microsoft.com/Directory.Read.All");
})
.AddAccountClaimsPrincipalFactory<RemoteAuthenticationState, CustomUserAccount,
    CustomUserFactory>();
```

Create a policy for each group or role in `Program.Main`. The following example creates a policy for the AAD built-in *Billing Administrator* role:

```
builder.Services.AddAuthorizationCore(options =>
{
    options.AddPolicy("BillingAdministrator", policy =>
        policy.RequireClaim("group", "69ff516a-b57d-4697-a429-9de4af7b5609"));
});
```

For the complete list of AAD role Object IDs, see the AAD Adminstrative Role Group IDs section.

In the following examples, the app uses the preceding policy to authorize the user.

The `AuthorizeView` component works with the policy:

```
<AuthorizeView Policy="BillingAdministrator">
    <Authorized>
        <p>
            The user is in the 'Billing Administrator' AAD Administrative Role
            and can see this content.
        </p>
    </Authorized>
    <NotAuthorized>
        <p>
            The user is NOT in the 'Billing Administrator' role and sees this
            content.
        </p>
    </NotAuthorized>
</AuthorizeView>
```

Access to an entire component can be based on the policy using `[Authorize]` attribute directive (AuthorizeAttribute):

```
@page "/"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize(Policy = "BillingAdministrator")]
```

If the user isn't logged in, they're redirected to the AAD sign-in page and then back to the component after they sign in.

A policy check can also be performed in code with procedural logic:

```
@page "/checkpolicy"
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<h1>Check Policy</h1>

<p>This component checks a policy in code.</p>

<button @onclick="CheckPolicy">Check 'BillingAdministrator' policy</button>

<p>Policy Message: @policyMessage</p>

@code {
    private string policyMessage = "Check hasn't been made yet.";

    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private async void CheckPolicy()
    {
        var user = (await authenticationStateTask).User;

        if ((await AuthorizationService.AuthorizeAsync(user,
            "BillingAdministrator")).Succeeded)
        {
            policyMessage = "Yes! The 'BillingAdministrator' policy is met.";
        }
        else
        {
            policyMessage = "No! 'BillingAdministrator' policy is NOT met.";
        }
    }
}
```

User-defined roles

An AAD-registered app can also be configured to use user-defined roles.

To configure the app in the Azure portal to provide a `roles` membership claim, see How to: Add app roles in your application and receive them in the token in the Azure documentation.

The following example assumes that an app is configured with two roles:

- `admin`
- `developer`

> **NOTE**
>
> Although you can't assign roles to security groups without an Azure AD Premium account, you can assign users to roles and receive a `roles` claim for users with a standard Azure account. The guidance in this section doesn't require an Azure AD Premium account.
>
> Multiple roles are assigned in the Azure portal by ***re-adding a user*** for each additional role assignment.

The single `roles` claim sent by AAD presents the user-defined roles as the `appRoles`'s `value`s in a JSON array. The app must convert the JSON array of roles into individual `role` claims.

The `CustomUserFactory` shown in the User-defined groups and AAD built-in Administrative Roles section is set up to act on a `roles` claim with a JSON array value. Add and register the `CustomUserFactory` in the standalone app or client app of a hosted Blazor solution as shown in the User-defined groups and AAD built-in Administrative Roles section. There's no need to provide code to remove the original `roles` claim because it's automatically removed by the framework.

In `Program.Main` of the standalone app or client app of a hosted Blazor solution, specify the claim named "`role`" as the role claim:

```
builder.Services.AddMsalAuthentication(options =>
{
    ...

    options.UserOptions.RoleClaim = "role";
});
```

Component authorization approaches are functional at this point. Any of the authorization mechanisms in components can use the `admin` role to authorize the user:

- `AuthorizeView` component (Example: `<AuthorizeView Roles="admin">`)
- `[Authorize]` attribute directive (AuthorizeAttribute) (Example: `@attribute [Authorize(Roles = "admin")]`)
- Procedural logic (Example: `if (user.IsInRole("admin")) { ... }`)

  Multiple role tests are supported:

  ```
  if (user.IsInRole("admin") && user.IsInRole("developer"))
  {
      ...
  }
  ```

AAD Adminstrative Role Group IDs

The Object IDs presented in the following table are used to create policies for `group` claims. Policies permit an app to authorize users for various activities in an app. For more information, see the User-defined groups and AAD built-in Administrative Roles section.

AAD ADMINISTRATIVE ROLE	OBJECT ID
Application administrator	fa11557b-4f15-4ddd-85d5-313c7cd74047
Application developer	68adcbb8-9504-44f6-89f2-5cd48dc74a2c

AAD ADMINISTRATIVE ROLE	OBJECT ID
Authentication administrator	02d110a1-96b1-419e-af87-746461b60ed7
Azure DevOps administrator	a5311ace-ca41-44cd-b833-8d22caa0b34f
Azure Information Protection administrator	18632dce-f9b5-4f01-abb5-37051f06860e
B2C IEF Keyset administrator	0c2e87e5-94f9-4adb-ae8c-bcafe11bd368
B2C IEF Policy administrator	bfcab36c-10c6-4b13-b63c-4d8b62c0c44e
B2C user flow administrator	baa531b7-8cf0-44ad-8f98-eded88dae827
B2C user flow attribute administrator	dd0baca0-a535-48c1-b871-8431abe16452
Billing administrator	69ff516a-b57d-4697-a429-9de4af7b5609
Cloud application administrator	250b5fe3-b553-458d-9a53-b782c13c34bf
Cloud device administrator	26cd4b44-2636-4ddb-bdfa-27feae66f86d
Compliance administrator	9d6e1dd0-c9f8-45f8-b558-b134f700116c
Compliance data administrator	4c0ca3a2-231e-416c-9411-4abe57d5cb9d
Conditional Access administrator	8f71a611-137d-49af-87ad-e97f1fd5da76
Customer LockBox access approver	c18d54a8-b13e-4954-a1a4-7deaf2e4f184
Desktop Analytics administrator	c62c4ac5-e4c6-4096-8a2f-1ee3cbaaae15
Directory readers	e1fc84a6-7762-4b9b-8e29-518b4adbc23b
Dynamics 365 administrator	f20a9cfa-9fdf-49a8-a977-1afe446a1d6e
Exchange administrator	b2ec2cc0-d5c9-4864-ad9b-38dd9dba2652
External Identity Provider administrator	febfaeb4-e478-407a-b4b3-f4d9716618a2
Global administrator	a45ba61b-44db-462c-924b-3b2719152588
Global reader	f6903b21-6aba-4124-b44c-76671796b9d5
Groups administrator	158b3e5a-d89d-460b-92b5-3b34985f0197
Guest inviter	4c730a1d-cc22-44af-8f9f-4eec635c7502
Helpdesk administrator	108678c8-6628-44e1-8d01-caf598a6a5f5
Intune administrator	79950741-23fa-4189-b2cb-46640601c497

AAD ADMINISTRATIVE ROLE	OBJECT ID
Kaizala administrator	d6322af2-48e7-42e0-8c68-0bbe31af3412
License administrator	3355458a-e423-44bf-8b98-4ac5e572cea5
Message center privacy reader	6395db95-9fb8-42b9-b1ed-30a2405eee6f
Message center reader	fd5d37b8-4e24-434b-9e63-70ed3b759a16
Office apps administrator	5f3870cd-b042-4f93-86d7-c9d77c664dc7
Password administrator	466e48b7-5d66-4ae5-8911-1a118de74941
Power BI administrator	984e83b8-8337-4255-91a1-acb663175ab4
Power platform administrator	76d6f95e-9a15-4d7d-8d21-00de00faf9fd
Privileged authentication administrator	0829f731-b46d-419f-9742-aeb122367d11
Privileged role administrator	f20a725a-d1c8-4107-83ea-1171c97d00c7
Reports reader	54635450-e8ed-4f2d-9632-07db2517b4de
Search administrator	c770a2f1-c9ba-4e60-9176-9f52b1eb1a31
Search editor	6a6858c6-5f0d-44ac-87c7-0190fbedd271
Security administrator	20fa50e3-6531-44d8-bd39-b251420568ad
Security operator	43aae017-8e51-4188-91ab-e6debd572800
Security reader	45035cd3-fd97-4250-8197-3a53d3562d9b
Service support administrator	2c92cf45-c914-48f8-9bf9-fc14b28818ab
SharePoint administrator	e1c32229-875e-461d-ae24-3cb99116e86c
Skype for Business administrator	0a8cee12-e21d-43ef-abd9-f1ea85710e30
Teams Communications Administrator	2393e455-6e13-4743-9f52-63fcec2b6a9c
Teams Communications Support Engineer	802dd94e-d717-46f6-af98-b9167071e9fc
Teams Communications Specialist	ef547281-cf46-4cc6-bcaa-f5eac3f030c9
Teams Service Administrator	8846a0be-197b-443a-b13c-11192691fa24
User administrator	1f6eed58-7dd3-460b-a298-666f975427a1

Additional resources

- Claims-based authorization in ASP.NET Core
- ASP.NET Core Blazor authentication and authorization

Secure ASP.NET Core Blazor Server apps

9/22/2020 • 2 minutes to read • Edit Online

By Luke Latham

Blazor Server apps are configured for security in the same manner as ASP.NET Core apps. For more information, see the articles under Overview of ASP.NET Core Security. Topics under this overview apply specifically to Blazor Server.

Blazor Server project template

The Blazor Server project template can be configured for authentication when the project is created.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

Follow the Visual Studio guidance in Tooling for ASP.NET Core Blazor to create a new Blazor Server project with an authentication mechanism.

After choosing the **Blazor Server App** template in the **Create a new ASP.NET Core Web Application** dialog, select **Change** under **Authentication**.

A dialog opens to offer the same set of authentication mechanisms available for other ASP.NET Core projects:

- **No Authentication**
- **Individual User Accounts**: User accounts can be stored:
  - Within the app using ASP.NET Core's Identity system.
  - With Azure AD B2C.
- **Work or School Accounts**
- **Windows Authentication**

Scaffold Identity

Scaffold Identity into a Blazor Server project:

- Without existing authorization.
- With authorization.

Threat mitigation guidance for ASP.NET Core Blazor Server

9/22/2020 • 22 minutes to read • Edit Online

By Javier Calvarro Nelson

Blazor Server apps adopt a *stateful* data processing model, where the server and client maintain a long-lived relationship. The persistent state is maintained by a circuit, which can span connections that are also potentially long-lived.

When a user visits a Blazor Server site, the server creates a circuit in the server's memory. The circuit indicates to the browser what content to render and responds to events, such as when the user selects a button in the UI. To perform these actions, a circuit invokes JavaScript functions in the user's browser and .NET methods on the server. This two-way JavaScript-based interaction is referred to as JavaScript interop (JS interop).

Because JS interop occurs over the Internet and the client uses a remote browser, Blazor Server apps share most web app security concerns. This topic describes common threats to Blazor Server apps and provides threat mitigation guidance focused on Internet-facing apps.

In constrained environments, such as inside corporate networks or intranets, some of the mitigation guidance either:

- Doesn't apply in the constrained environment.
- Isn't worth the cost to implement because the security risk is low in a constrained environment.

Blazor and shared state

Blazor server apps live in server memory. That means that there are multiple apps hosted within the same process. For each app session, Blazor starts a circuit with its own DI container scope. That means that scoped services are unique per Blazor session.

> **WARNING**
> We don't recommend apps on the same server share state using singleton services unless extreme care is taken, as this can introduce security vulnerabilities, such as leaking user state across circuits.

You can use stateful singleton services in Blazor apps if they are specifically designed for it. For example, it's ok to use a memory cache as a singleton because it requires a key to access a given entry, assuming users don't have control of what cache keys are used.

Additionally, again for security reasons, you must not use IHttpContextAccessor within Blazor apps. Blazor apps run outside of the context of the ASP.NET Core pipeline. The HttpContext isn't guaranteed to be available within the IHttpContextAccessor, nor is it guaranteed to be holding the context that started the Blazor app.

The recommended way to pass request state to the Blazor app is through parameters to the root component in the initial rendering of the app:

- Define a class with all the data you want to pass to the Blazor app.
- Populate that data from the Razor page using the HttpContext available at that time.
- Pass the data to the Blazor app as a parameter to the root component (App).

- Define a parameter in the root component to hold the data being passed to the app.
- Use the user-specific data within the app; or alternatively, copy that data into a scoped service within OnInitializedAsync so that it can be used across the app.

For more information and example code, see ASP.NET Core Blazor Server additional security scenarios.

Resource exhaustion

Resource exhaustion can occur when a client interacts with the server and causes the server to consume excessive resources. Excessive resource consumption primarily affects:

- CPU
- Memory
- Client connections

Denial of service (DoS) attacks usually seek to exhaust an app or server's resources. However, resource exhaustion isn't necessarily the result of an attack on the system. For example, finite resources can be exhausted due to high user demand. DoS is covered further in the Denial of service (DoS) attacks section.

Resources external to the Blazor framework, such as databases and file handles (used to read and write files), may also experience resource exhaustion. For more information, see ASP.NET Core Performance Best Practices.

CPU

CPU exhaustion can occur when one or more clients force the server to perform intensive CPU work.

For example, consider a Blazor Server app that calculates a *Fibonnacci number*. A Fibonnacci number is produced from a Fibonnacci sequence, where each number in the sequence is the sum of the two preceding numbers. The amount of work required to reach the answer depends on the length of the sequence and the size of the initial value. If the app doesn't place limits on a client's request, the CPU-intensive calculations may dominate the CPU's time and diminish the performance of other tasks. Excessive resource consumption is a security concern impacting availability.

CPU exhaustion is a concern for all public-facing apps. In regular web apps, requests and connections time out as a safeguard, but Blazor Server apps don't provide the same safeguards. Blazor Server apps must include appropriate checks and limits before performing potentially CPU-intensive work.

Memory

Memory exhaustion can occur when one or more clients force the server to consume a large amount of memory.

For example, consider a Blazor-server side app with a component that accepts and displays a list of items. If the Blazor app doesn't place limits on the number of items allowed or the number of items rendered back to the client, the memory-intensive processing and rendering may dominate the server's memory to the point where performance of the server suffers. The server may crash or slow to the point that it appears to have crashed.

Consider the following scenario for maintaining and displaying a list of items that pertain to a potential memory exhaustion scenario on the server:

- The items in a `List<MyItem>` property or field use the server's memory. If the app allows the list of items to grow unbounded, there's a risk of the server running out of memory. Running out of memory causes the current session to end (crash) and all of the concurrent sessions in that server instance receive an out-of-memory exception. To prevent this scenario from occurring, the app must use a data structure that imposes an item limit on concurrent users.
- If a paging scheme isn't used for rendering, the server uses additional memory for objects that aren't visible in the UI. Without a limit on the number of items, memory demands may exhaust the available server memory. To prevent this scenario, use one of the following approaches:
  - Use paginated lists when rendering.

- Only display the first 100 to 1,000 items and require the user to enter search criteria to find items beyond the items displayed.
- For a more advanced rendering scenario, implement lists or grids that support *virtualization*. Using virtualization, lists only render a subset of items currently visible to the user. When the user interacts with the scrollbar in the UI, the component renders only those items required for display. The items that aren't currently required for display can be held in secondary storage, which is the ideal approach. Undisplayed items can also be held in memory, which is less ideal.

Blazor Server apps offer a similar programming model to other UI frameworks for stateful apps, such as WPF, Windows Forms, or Blazor WebAssembly. The main difference is that in several of the UI frameworks the memory consumed by the app belongs to the client and only affects that individual client. For example, a Blazor WebAssembly app runs entirely on the client and only uses client memory resources. In the Blazor Server scenario, the memory consumed by the app belongs to the server and is shared among clients on the server instance.

Server-side memory demands are a consideration for all Blazor Server apps. However, most web apps are stateless, and the memory used while processing a request is released when the response is returned. As a general recommendation, don't permit clients to allocate an unbound amount of memory as in any other server-side app that persists client connections. The memory consumed by a Blazor Server app persists for a longer time than a single request.

> **NOTE**
> During development, a profiler can be used or a trace captured to assess memory demands of clients. A profiler or trace won't capture the memory allocated to a specific client. To capture the memory use of a specific client during development, capture a dump and examine the memory demand of all the objects rooted at a user's circuit.

Client connections

Connection exhaustion can occur when one or more clients open too many concurrent connections to the server, preventing other clients from establishing new connections.

Blazor clients establish a single connection per session and keep the connection open for as long as the browser window is open. The demands on the server of maintaining all of the connections isn't specific to Blazor apps. Given the persistent nature of the connections and the stateful nature of Blazor Server apps, connection exhaustion is a greater risk to availability of the app.

By default, there's no limit on the number of connections per user for a Blazor Server app. If the app requires a connection limit, take one or more of the following approaches:

- Require authentication, which naturally limits the ability of unauthorized users to connect to the app. For this scenario to be effective, users must be prevented from provisioning new users at will.
- Limit the number of connections per user. Limiting connections can be accomplished via the following approaches. Exercise care to allow legitimate users to access the app (for example, when a connection limit is established based on the client's IP address).
  - At the app level:
    - Endpoint routing extensibility.
    - Require authentication to connect to the app and keep track of the active sessions per user.
    - Reject new sessions upon reaching a limit.
    - Proxy WebSocket connections to an app through the use of a proxy, such as the Azure SignalR Service that multiplexes connections from clients to an app. This provides an app with greater connection capacity than a single client can establish, preventing a client from exhausting the connections to the server.
  - At the server level: Use a proxy/gateway in front of the app. For example, Azure Front Door enables you

to define, manage, and monitor the global routing of web traffic to an app.

Denial of service (DoS) attacks

Denial of service (DoS) attacks involve a client causing the server to exhaust one or more of its resources making the app unavailable. Blazor Server apps include some default limits and rely on other ASP.NET Core and SignalR limits to protect against DoS attacks that are set on CircuitOptions.

BLAZOR SERVER APP LIMIT	DESCRIPTION	DEFAULT
DisconnectedCircuitMaxRetained	Maximum number of disconnected circuits that a given server holds in memory at a time.	100
DisconnectedCircuitRetentionPeriod	Maximum amount of time a disconnected circuit is held in memory before being torn down.	3 minutes
JSInteropDefaultCallTimeout	Maximum amount of time the server waits before timing out an asynchronous JavaScript function invocation.	1 minute
MaxBufferedUnacknowledgedRenderBatches	Maximum number of unacknowledged render batches the server keeps in memory per circuit at a given time to support robust reconnection. After reaching the limit, the server stops producing new render batches until one or more batches have been acknowledged by the client.	10

Set the maximum message size of a single incoming hub message with HubConnectionContextOptions.

SIGNALR AND ASP.NET CORE LIMIT	DESCRIPTION	DEFAULT
HubConnectionContextOptions.MaximumReceiveMessageSize	Message size for an individual message.	32 KB

Interactions with the browser (client)

A client interacts with the server through JS interop event dispatching and render completion. JS interop communication goes both ways between JavaScript and .NET:

- Browser events are dispatched from the client to the server in an asynchronous fashion.
- The server responds asynchronously rerendering the UI as necessary.

JavaScript functions invoked from .NET

For calls from .NET methods to JavaScript:

- All invocations have a configurable timeout after which they fail, returning a OperationCanceledException to the caller.
  - There's a default timeout for the calls (CircuitOptions.JSInteropDefaultCallTimeout) of one minute. To configure this limit, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.
  - A cancellation token can be provided to control the cancellation on a per-call basis. Rely on the default call timeout where possible and time-bound any call to the client if a cancellation token is provided.

- The result of a JavaScript call can't be trusted. The Blazor app client running in the browser searches for the JavaScript function to invoke. The function is invoked, and either the result or an error is produced. A malicious client can attempt to:
  - Cause an issue in the app by returning an error from the JavaScript function.
  - Induce an unintended behavior on the server by returning an unexpected result from the JavaScript function.

Take the following precautions to guard against the preceding scenarios:

- Wrap JS interop calls within `try-catch` statements to account for errors that might occur during the invocations. For more information, see Handle errors in ASP.NET Core Blazor apps.
- Validate data returned from JS interop invocations, including error messages, before taking any action.

.NET methods invoked from the browser

Don't trust calls from JavaScript to .NET methods. When a .NET method is exposed to JavaScript, consider how the .NET method is invoked:

- Treat any .NET method exposed to JavaScript as you would a public endpoint to the app.
  - Validate input.
    - Ensure that values are within expected ranges.
    - Ensure that the user has permission to perform the action requested.
  - Don't allocate an excessive quantity of resources as part of the .NET method invocation. For example, perform checks and place limits on CPU and memory use.
  - Take into account that static and instance methods can be exposed to JavaScript clients. Avoid sharing state across sessions unless the design calls for sharing state with appropriate constraints.
    - For instance methods exposed through `DotNetReference` objects that are originally created through dependency injection (DI), the objects should be registered as scoped objects. This applies to any DI service that the Blazor Server app uses.
    - For static methods, avoid establishing state that can't be scoped to the client unless the app is explicitly sharing state by-design across all users on a server instance.
  - Avoid passing user-supplied data in parameters to JavaScript calls. If passing data in parameters is absolutely required, ensure that the JavaScript code handles passing the data without introducing Cross-site scripting (XSS) vulnerabilities. For example, don't write user-supplied data to the Document Object Model (DOM) by setting the `innerHTML` property of an element. Consider using Content Security Policy (CSP) to disable `eval` and other unsafe JavaScript primitives.
- Avoid implementing custom dispatching of .NET invocations on top of the framework's dispatching implementation. Exposing .NET methods to the browser is an advanced scenario, not recommended for general Blazor development.

Events

Events provide an entry point to a Blazor Server app. The same rules for safeguarding endpoints in web apps apply to event handling in Blazor Server apps. A malicious client can send any data it wishes to send as the payload for an event.

For example:

- A change event for a `<select>` could send a value that isn't within the options that the app presented to the client.
- An `<input>` could send any text data to the server, bypassing client-side validation.

The app must validate the data for any event that the app handles. The Blazor framework forms components perform basic validations. If the app uses custom forms components, custom code must be written to validate event data as appropriate.

Blazor Server events are asynchronous, so multiple events can be dispatched to the server before the app has time to react by producing a new render. This has some security implications to consider. Limiting client actions in the app must be performed inside event handlers and not depend on the current rendered view state.

Consider a counter component that should allow a user to increment a counter a maximum of three times. The button to increment the counter is conditionally based on the value of `count`:

```
<p>Count: @count<p>

@if (count < 3)
{
    <button @onclick="IncrementCount" value="Increment count" />
}

@code
{
    private int count = 0;

    private void IncrementCount()
    {
        count++;
    }
}
```

A client can dispatch one or more increment events before the framework produces a new render of this component. The result is that the `count` can be incremented *over three times* by the user because the button isn't removed by the UI quickly enough. The correct way to achieve the limit of three `count` increments is shown in the following example:

```
<p>Count: @count<p>

@if (count < 3)
{
    <button @onclick="IncrementCount" value="Increment count" />
}

@code
{
    private int count = 0;

    private void IncrementCount()
    {
        if (count < 3)
        {
            count++;
        }
    }
}
```

By adding the `if (count < 3) { ... }` check inside the handler, the decision to increment `count` is based on the current app state. The decision isn't based on the state of the UI as it was in the previous example, which might be temporarily stale.

Guard against multiple dispatches

If an event callback invokes a long running operation asynchronously, such as fetching data from an external service or database, consider using a guard. The guard can prevent the user from queueing up multiple operations while the operation is in progress with visual feedback. The following component code sets `isLoading` to `true` while `GetForecastAsync` obtains data from the server. While `isLoading` is `true`, the button is disabled in the UI:

```
@page "/fetchdata"
@using BlazorServerSample.Data
@inject WeatherForecastService ForecastService

<button disabled="@isLoading" @onclick="UpdateForecasts">Update</button>

@code {
    private bool isLoading;
    private WeatherForecast[] forecasts;

    private async Task UpdateForecasts()
    {
        if (!isLoading)
        {
            isLoading = true;
            forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
            isLoading = false;
        }
    }
}
```

The guard pattern demonstrated in the preceding example works if the background operation is executed asynchronously with the `async` - `await` pattern.

Cancel early and avoid use-after-dispose

In addition to using a guard as described in the Guard against multiple dispatches section, consider using a CancellationToken to cancel long-running operations when the component is disposed. This approach has the added benefit of avoiding *use-after-dispose* in components:

```
@implements IDisposable

...

@code {
    private readonly CancellationTokenSource TokenSource =
        new CancellationTokenSource();

    private async Task UpdateForecasts()
    {
        ...

        forecasts = await ForecastService.GetForecastAsync(DateTime.Now,
            TokenSource.Token);

        if (TokenSource.Token.IsCancellationRequested)
        {
           return;
        }

        ...
    }

    public void Dispose()
    {
        TokenSource.Cancel();
    }
}
```

Avoid events that produce large amounts of data

Some DOM events, such as `oninput` or `onscroll`, can produce a large amount of data. Avoid using these events in Blazor server apps.

Additional security guidance

The guidance for securing ASP.NET Core apps apply to Blazor Server apps and are covered in the following sections:

- Logging and sensitive data
- Protect information in transit with HTTPS
- Cross-site scripting (XSS)
- Cross-origin protection
- Click-jacking
- Open redirects

Logging and sensitive data

JS interop interactions between the client and server are recorded in the server's logs with ILogger instances. Blazor avoids logging sensitive information, such as actual events or JS interop inputs and outputs.

When an error occurs on the server, the framework notifies the client and tears down the session. By default, the client receives a generic error message that can be seen in the browser's developer tools.

The client-side error doesn't include the callstack and doesn't provide detail on the cause of the error, but server logs do contain such information. For development purposes, sensitive error information can be made available to the client by enabling detailed errors.

Enable detailed errors in JavaScript with:

- CircuitOptions.DetailedErrors.
- The `DetailedErrors` configuration key set to `true`, which can be set in the app settings file (`appsettings.json`). The key can also be set using the `ASPNETCORE_DETAILEDERRORS` environment variable with a value of `true`.

> **WARNING**
> Exposing error information to clients on the Internet is a security risk that should always be avoided.

Protect information in transit with HTTPS

Blazor Server uses SignalR for communication between the client and the server. Blazor Server normally uses the transport that SignalR negotiates, which is typically WebSockets.

Blazor Server doesn't ensure the integrity and confidentiality of the data sent between the server and the client. Always use HTTPS.

Cross-site scripting (XSS)

Cross-site scripting (XSS) allows an unauthorized party to execute arbitrary logic in the context of the browser. A compromised app could potentially run arbitrary code on the client. The vulnerability could be used to potentially perform a number of malicious actions against the server:

- Dispatch fake/invalid events to the server.
- Dispatch fail/invalid render completions.
- Avoid dispatching render completions.
- Dispatch interop calls from JavaScript to .NET.
- Modify the response of interop calls from .NET to JavaScript.
- Avoid dispatching .NET to JS interop results.

The Blazor Server framework takes steps to protect against some of the preceding threats:

- Stops producing new UI updates if the client isn't acknowledging render batches. Configured with

CircuitOptions.MaxBufferedUnacknowledgedRenderBatches.

- Times out any .NET to JavaScript call after one minute without receiving a response from the client. Configured with CircuitOptions.JSInteropDefaultCallTimeout.
- Performs basic validation on all input coming from the browser during JS interop:
  - .NET references are valid and of the type expected by the .NET method.
  - The data isn't malformed.
  - The correct number of arguments for the method are present in the payload.
  - The arguments or result can be deserialized correctly before invoking the method.
- Performs basic validation in all input coming from the browser from dispatched events:
  - The event has a valid type.
  - The data for the event can be deserialized.
  - There's an event handler associated with the event.

In addition to the safeguards that the framework implements, the app must be coded by the developer to safeguard against threats and take appropriate actions:

- Always validate data when handling events.
- Take appropriate action upon receiving invalid data:
  - Ignore the data and return. This allows the app to continue processing requests.
  - If the app determines that the input is illegitimate and couldn't be produced by legitimate client, throw an exception. Throwing an exception tears down the circuit and ends the session.
- Don't trust the error message provided by render batch completions included in the logs. The error is provided by the client and can't generally be trusted, as the client might be compromised.
- Don't trust the input on JS interop calls in either direction between JavaScript and .NET methods.
- The app is responsible for validating that the content of arguments and results are valid, even if the arguments or results are correctly deserialized.

For a XSS vulnerability to exist, the app must incorporate user input in the rendered page. Blazor Server components execute a compile-time step where the markup in a `.razor` file is transformed into procedural C# logic. At runtime, the C# logic builds a *render tree* describing the elements, text, and child components. This is applied to the browser's DOM via a sequence of JavaScript instructions (or is serialized to HTML in the case of prerendering):

- User input rendered via normal Razor syntax (for example, `@someStringValue`) doesn't expose a XSS vulnerability because the Razor syntax is added to the DOM via commands that can only write text. Even if the value includes HTML markup, the value is displayed as static text. When prerendering, the output is HTML-encoded, which also displays the content as static text.
- Script tags aren't allowed and shouldn't be included in the app's component render tree. If a script tag is included in a component's markup, a compile-time error is generated.
- Component authors can author components in C# without using Razor. The component author is responsible for using the correct APIs when emitting output. For example, use `builder.AddContent(0, someUserSuppliedString)` and *not* `builder.AddMarkupContent(0, someUserSuppliedString)`, as the latter could create a XSS vulnerability.

As part of protecting against XSS attacks, consider implementing XSS mitigations, such as Content Security Policy (CSP).

For more information, see Prevent Cross-Site Scripting (XSS) in ASP.NET Core.

Cross-origin protection

Cross-origin attacks involve a client from a different origin performing an action against the server. The malicious action is typically a GET request or a form POST (Cross-Site Request Forgery, CSRF), but opening a malicious

WebSocket is also possible. Blazor Server apps offer the same guarantees that any other SignalR app using the hub protocol offer:

- Blazor Server apps can be accessed cross-origin unless additional measures are taken to prevent it. To disable cross-origin access, either disable CORS in the endpoint by adding the CORS middleware to the pipeline and adding the DisableCorsAttribute to the Blazor endpoint metadata or limit the set of allowed origins by configuring SignalR for cross-origin resource sharing.
- If CORS is enabled, extra steps might be required to protect the app depending on the CORS configuration. If CORS is globally enabled, CORS can be disabled for the Blazor Server hub by adding the DisableCorsAttribute metadata to the endpoint metadata after calling MapBlazorHub on the endpoint route builder.

For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Click-jacking

Click-jacking involves rendering a site as an `<iframe>` inside a site from a different origin in order to trick the user into performing actions on the site under attack.

To protect an app from rendering inside of an `<iframe>`, use Content Security Policy (CSP) and the `X-Frame-Options` header. For more information, see MDN web docs: X-Frame-Options.

Open redirects

When a Blazor Server app session starts, the server performs basic validation of the URLs sent as part of starting the session. The framework checks that the base URL is a parent of the current URL before establishing the circuit. No additional checks are performed by the framework.

When a user selects a link on the client, the URL for the link is sent to the server, which determines what action to take. For example, the app may perform a client-side navigation or indicate to the browser to go to the new location.

Components can also trigger navigation requests programatically through the use of NavigationManager. In such scenarios, the app might perform a client-side navigation or indicate to the browser to go to the new location.

Components must:

- Avoid using user input as part of the navigation call arguments.
- Validate arguments to ensure that the target is allowed by the app.

Otherwise, a malicious user can force the browser to go to an attacker-controlled site. In this scenario, the attacker tricks the app into using some user input as part of the invocation of the NavigationManager.NavigateTo method.

This advice also applies when rendering links as part of the app:

- If possible, use relative links.
- Validate that absolute link destinations are valid before including them in a page.

For more information, see Prevent open redirect attacks in ASP.NET Core.

Security checklist

The following list of security considerations isn't comprehensive:

- Validate arguments from events.
- Validate inputs and results from JS interop calls.
- Avoid using (or validate beforehand) user input for .NET to JS interop calls.
- Prevent the client from allocating an unbound amount of memory.
  - Data within the component.

    - `DotNetObject` references returned to the client.
- Guard against multiple dispatches.
- Cancel long-running operations when the component is disposed.
- Avoid events that produce large amounts of data.
- Avoid using user input as part of calls to NavigationManager.NavigateTo and validate user input for URLs against a set of allowed origins first if unavoidable.
- Don't make authorization decisions based on the state of the UI but only from component state.
- Consider using Content Security Policy (CSP) to protect against XSS attacks.
- Consider using CSP and X-Frame-Options to protect against click-jacking.
- Ensure CORS settings are appropriate when enabling CORS or explicitly disable CORS for Blazor apps.
- Test to ensure that the server-side limits for the Blazor app provide an acceptable user experience without unacceptable levels of risk.

ASP.NET Core Blazor Server additional security scenarios

9/22/2020 • 3 minutes to read • Edit Online

By Javier Calvarro Nelson

Pass tokens to a Blazor Server app

Tokens available outside of the Razor components in a Blazor Server app can be passed to components with the approach described in this section. For sample code, including a complete `Startup.ConfigureServices` example, see the Passing tokens to a server-side Blazor application.

Authenticate the Blazor Server app as you would with a regular Razor Pages or MVC app. Provision and save the tokens to the authentication cookie. For example:

```
using Microsoft.AspNetCore.Authentication.OpenIdConnect;

...

services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme, options =>
{
    options.ResponseType = "code";
    options.SaveTokens = true;

    options.Scope.Add("offline_access");
    options.Scope.Add("{SCOPE}");
    options.Resource = "{RESOURCE}";
});
```

Define a class to pass in the initial app state with the access and refresh tokens:

```
public class InitialApplicationState
{
    public string AccessToken { get; set; }
    public string RefreshToken { get; set; }
}
```

Define a **scoped** token provider service that can be used within the Blazor app to resolve the tokens from dependency injection (DI):

```
public class TokenProvider
{
    public string AccessToken { get; set; }
    public string RefreshToken { get; set; }
}
```

In `Startup.ConfigureServices`, add services for:

- `IHttpClientFactory`
- `TokenProvider`

```
services.AddHttpClient();
services.AddScoped<TokenProvider>();
```

In the `_Host.cshtml` file, create and instance of `InitialApplicationState` and pass it as a parameter to the app:

```
@using Microsoft.AspNetCore.Authentication

...

@{
    var tokens = new InitialApplicationState
    {
        AccessToken = await HttpContext.GetTokenAsync("access_token"),
        RefreshToken = await HttpContext.GetTokenAsync("refresh_token")
    };
}

<app>
    <component type="typeof(App)" param-InitialState="tokens"
        render-mode="ServerPrerendered" />
</app>
```

In the `App` component (`App.razor`), resolve the service and initialize it with the data from the parameter:

```
@inject TokenProvider TokenProvider

...

@code {
    [Parameter]
    public InitialApplicationState InitialState { get; set; }

    protected override Task OnInitializedAsync()
    {
        TokenProvider.AccessToken = InitialState.AccessToken;
        TokenProvider.RefreshToken = InitialState.RefreshToken;

        return base.OnInitializedAsync();
    }
}
```

Add a package reference to the app for the Microsoft.AspNet.WebApi.Client NuGet package.

In the service that makes a secure API request, inject the token provider and retrieve the token to call the API:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class WeatherForecastService
{
    private readonly TokenProvider store;

    public WeatherForecastService(IHttpClientFactory clientFactory,
        TokenProvider tokenProvider)
    {
        Client = clientFactory.CreateClient();
        store = tokenProvider;
    }

    public HttpClient Client { get; }

    public async Task<WeatherForecast[]> GetForecastAsync(DateTime startDate)
    {
        var token = store.AccessToken;
        var request = new HttpRequestMessage(HttpMethod.Get,
            "https://localhost:5003/WeatherForecast");
        request.Headers.Add("Authorization", $"Bearer {token}");
        var response = await Client.SendAsync(request);
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsAsync<WeatherForecast[]>();
    }
}
```

Set the authentication scheme

For an app that uses more than one Authentication Middleware and thus has more than one authentication scheme, the scheme that Blazor uses can be explicitly set in the endpoint configuration of `Startup.Configure`. The following example sets the Azure Active Directory scheme:

```
endpoints.MapBlazorHub().RequireAuthorization(
    new AuthorizeAttribute
    {
        AuthenticationSchemes = AzureADDefaults.AuthenticationScheme
    });
```

Use OpenID Connect (OIDC) v2.0 endpoints

The authentication library and Blazor templates use OpenID Connect (OIDC) v1.0 endpoints. To use a v2.0 endpoint, configure the OpenIdConnectOptions.Authority option in the OpenIdConnectOptions:

```
services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme,
    options =>
    {
        options.Authority += "/v2.0";
    }
```

Alternatively, the setting can be made in the app settings (`appsettings.json`) file:

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/common/oauth2/v2.0/",
    ...
  }
}
```

If tacking on a segment to the authority isn't appropriate for the app's OIDC provider, such as with non-AAD providers, set the Authority property directly. Either set the property in OpenIdConnectOptions or in the app settings file with the Authority key.

Code changes

- The list of claims in the ID token changes for v2.0 endpoints. For more information, see Why update to Microsoft identity platform (v2.0)? in the Azure documentation.
- Since resources are specified in scope URIs for v2.0 endpoints, remove the OpenIdConnectOptions.Resource property setting in OpenIdConnectOptions:

  ```
  services.Configure<OpenIdConnectOptions>(AzureADDefaults.OpenIdScheme, options =>
      {
          ...
          options.Resource = "...";    // REMOVE THIS LINE
          ...
      }
      ```

  For more information, see [Scopes, not resources](/azure/active-directory/azuread-dev/azure-ad-endpoint-comparison#scopes-not-resources) in the Azure documentation.

App ID URI

- When using v2.0 endpoints, APIs define an *App ID URI*, which is meant to represent a unique identifier for the API.
- All scopes include the App ID URI as a prefix, and v2.0 endpoints emit access tokens with the App ID URI as the audience.
- When using V2.0 endpoints, the client ID configured in the Server API changes from the API Application ID (Client ID) to the App ID URI.

`appsettings.json`:

```
{
  "AzureAd": {
    ...
    "ClientId": "https://{TENANT}.onmicrosoft.com/{APP NAME}"
    ...
  }
}
```

You can find the App ID URI to use in the OIDC provider app registration description.

Enforce a Content Security Policy for ASP.NET Core Blazor

9/22/2020 • 5 minutes to read • Edit Online

By Javier Calvarro Nelson and Luke Latham

Cross-Site Scripting (XSS) is a security vulnerability where an attacker places one or more malicious client-side scripts into an app's rendered content. A Content Security Policy (CSP) helps protect against XSS attacks by informing the browser of valid:

- Sources for loaded content, including scripts, stylesheets, and images.
- Actions taken by a page, specifying permitted URL targets of forms.
- Plugins that can be loaded.

To apply a CSP to an app, the developer specifies several CSP content security *directives* in one or more `Content-Security-Policy` headers or `<meta>` tags.

Policies are evaluated by the browser while a page is loading. The browser inspects the page's sources and determines if they meet the requirements of the content security directives. When policy directives aren't met for a resource, the browser doesn't load the resource. For example, consider a policy that doesn't allow third-party scripts. When a page contains a `<script>` tag with a third-party origin in the `src` attribute, the browser prevents the script from loading.

CSP is supported in most modern desktop and mobile browsers, including Chrome, Edge, Firefox, Opera, and Safari. CSP is recommended for Blazor apps.

Policy directives

Minimally, specify the following directives and sources for Blazor apps. Add additional directives and sources as needed. The following directives are used in the Apply the policy section of this article, where example security policies for Blazor WebAssembly and Blazor Server are provided:

- base-uri: Restricts the URLs for a page's `<base>` tag. Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
- block-all-mixed-content: Prevents loading mixed HTTP and HTTPS content.
- default-src: Indicates a fallback for source directives that aren't explicitly specified by the policy. Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
- img-src: Indicates valid sources for images.
  - Specify `data:` to permit loading images from `data:` URLs.
  - Specify `https:` to permit loading images from HTTPS endpoints.
- object-src: Indicates valid sources for the `<object>`, `<embed>`, and `<applet>` tags. Specify `none` to prevent all URL sources.
- script-src: Indicates valid sources for scripts.
  - Specify the `https://stackpath.bootstrapcdn.com/` host source for Bootstrap scripts.
  - Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
  - In a Blazor WebAssembly app:
    - Specify the following hashes to permit the required Blazor WebAssembly inline scripts to load:
      - `sha256-v8ZC9OgMhcnEQ/Me77/R9TlJfzOBqrMTW8e1KuqLaqc=`

        - `sha256-If//FtbPc03afjLezvWHnC3Nbu4fDM04IIzkPaf3pH0=`
        - `sha256-v8v3RKRPmN4odZ1CWM5gw80QKPCCWMcpNeOmimNL2AA=`
    - Specify `unsafe-eval` to use `eval()` and methods for creating code from strings.
  - In a Blazor Server app, specify the `sha256-34WLX60Tw3aG6hylk0plKbZZFXCuepeQ6Hu7OqRf8PI=` hash for the inline script that performs fallback detection for stylesheets.
- style-src: Indicates valid sources for stylesheets.
  - Specify the `https://stackpath.bootstrapcdn.com/` host source for Bootstrap stylesheets.
  - Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
  - Specify `unsafe-inline` to allow the use of inline styles. The inline declaration is required for the UI in Blazor Server apps for reconnecting the client and server after the initial request. In a future release, inline styling might be removed so that `unsafe-inline` is no longer required.
- upgrade-insecure-requests: Indicates that content URLs from insecure (HTTP) sources should be acquired securely over HTTPS.

The preceding directives are supported by all browsers except Microsoft Internet Explorer.

To obtain SHA hashes for additional inline scripts:

- Apply the CSP shown in the Apply the policy section.
- Access the browser's developer tools console while running the app locally. The browser calculates and displays hashes for blocked scripts when a CSP header or `meta` tag is present.
- Copy the hashes provided by the browser to the `script-src` sources. Use single quotes around each hash.

For a Content Security Policy Level 2 browser support matrix, see Can I use: Content Security Policy Level 2.

Apply the policy

Use a `<meta>` tag to apply the policy:

- Set the value of the `http-equiv` attribute to `Content-Security-Policy`.
- Place the directives in the `content` attribute value. Separate directives with a semicolon (`;`).
- Always place the `meta` tag in the `<head>` content.

The following sections show example policies for Blazor WebAssembly and Blazor Server. These examples are versioned with this article for each release of Blazor. To use a version appropriate for your release, select the document version with the **Version** drop down selector on this webpage.

Blazor WebAssembly

In the `<head>` content of the `wwwroot/index.html` host page, apply the directives described in the Policy directives section:

```
<meta http-equiv="Content-Security-Policy"
      content="base-uri 'self';
               block-all-mixed-content;
               default-src 'self';
               img-src data: https:;
               object-src 'none';
               script-src https://stackpath.bootstrapcdn.com/
                          'self'
                          'sha256-v8ZC9OgMhcnEQ/Me77/R9TlJfzOBqrMTW8e1KuqLaqc='
                          'sha256-If//FtbPc03afjLezvWHnC3Nbu4fDM04IIzkPaf3pH0='
                          'sha256-v8v3RKRPmN4odZ1CWM5gw80QKPCCWMcpNeOmimNL2AA='
                          'unsafe-eval';
               style-src https://stackpath.bootstrapcdn.com/
                         'self'
                         'unsafe-inline';
               upgrade-insecure-requests;">
```

Blazor Server

In the `<head>` content of the `Pages/_Host.cshtml` host page, apply the directives described in the Policy directives section:

```
<meta http-equiv="Content-Security-Policy"
      content="base-uri 'self';
               block-all-mixed-content;
               default-src 'self';
               img-src data: https:;
               object-src 'none';
               script-src https://stackpath.bootstrapcdn.com/
                          'self'
                          'sha256-34WLX60Tw3aG6hylk0plKbZZFXCuepeQ6Hu7OqRf8PI=';
               style-src https://stackpath.bootstrapcdn.com/
                         'self'
                         'unsafe-inline';
               upgrade-insecure-requests;">
```

Meta tag limitations

A `<meta>` tag policy doesn't support the following directives:

- frame-ancestors
- report-to
- report-uri
- sandbox

To support the preceding directives, use a header named `Content-Security-Policy`. The directive string is the header's value.

Test a policy and receive violation reports

Testing helps confirm that third-party scripts aren't inadvertently blocked when building an initial policy.

To test a policy over a period of time without enforcing the policy directives, set the `<meta>` tag's `http-equiv` attribute or header name of a header-based policy to `Content-Security-Policy-Report-Only`. Failure reports are sent as JSON documents to a specified URL. For more information, see MDN web docs: Content-Security-Policy-Report-Only.

For reporting on violations while a policy is active, see the following articles:

- report-to
- report-uri

Although `report-uri` is no longer recommended for use, both directives should be used until `report-to` is supported by all of the major browsers. Don't exclusively use `report-uri` because support for `report-uri` is subject to being dropped *at any time* from browsers. Remove support for `report-uri` in your policies when `report-to` is fully supported. To track adoption of `report-to`, see Can I use: report-to.

Test and update an app's policy every release.

Troubleshoot

- Errors appear in the browser's developer tools console. Browsers provide information about:
  - Elements that don't comply with the policy.
  - How to modify the policy to allow for a blocked item.
- A policy is only completely effective when the client's browser supports all of the included directives. For a current browser support matrix, see Can I use: Content-Security-Policy.

Additional resources

- MDN web docs: Content-Security-Policy
- Content Security Policy Level 2
- Google CSP Evaluator

ASP.NET Core Blazor state management

9/22/2020 • 20 minutes to read • Edit Online

By Steve Sanderson and Luke Latham

User state created in a Blazor WebAssembly app is held in the browser's memory.

Examples of user state held in browser memory include:

- The hierarchy of component instances and their most recent render output in the rendered UI.
- The values of fields and properties in component instances.
- Data held in dependency injection (DI) service instances.
- Values set through JavaScript interop calls.

When a user closes and re-opens their browser or reloads the page, user state held in the browser's memory is lost.

Persist state across browser sessions

Generally, maintain state across browser sessions where users are actively creating data, not simply reading data that already exists.

To preserve state across browser sessions, the app must persist the data to some other storage location than the browser's memory. State persistence isn't automatic. You must take steps when developing the app to implement stateful data persistence.

Data persistence is typically only required for high-value state that users expended effort to create. In the following examples, persisting state either saves time or aids in commercial activities:

- Multi-step web forms: It's time-consuming for a user to re-enter data for several completed steps of a multi-step web form if their state is lost. A user loses state in this scenario if they navigate away from the form and return later.
- Shopping carts: Any commercially important component of an app that represents potential revenue can be maintained. A user who loses their state, and thus their shopping cart, may purchase fewer products or services when they return to the site later.

An app can only persist *app state*. UIs can't be persisted, such as component instances and their render trees. Components and render trees aren't generally serializable. To persist UI state, such as the expanded nodes of a tree view control, the app must use custom code to model the behavior of the UI state as serializable app state.

Where to persist state

Three common locations exist for persisting state:

- Server-side storage
- URL
- Browser storage

Server-side storage

For permanent data persistence that spans multiple users and devices, the app can use independent server-side storage accessed via a web API. Options include:

- Blob storage

- Key-value storage
- Relational database
- Table storage

After data is saved, the user's state is retained and available in any new browser session.

Because Blazor WebAssembly apps run entirely in the user's browser, they require additional measures to access secure external systems, such as storage services and databases. Blazor WebAssembly apps are secured in the same manner as Single Page Applications (SPAs). Typically, an app authenticates a user via OAuth/OpenID Connect (OIDC) and then interacts with storage services and databases through web API calls to a server-side app. The server-side app mediates the transfer of data between the Blazor WebAssembly app and the storage service or database. The Blazor WebAssembly app maintains an ephemeral connection to the server-side app, while the server-side app has a persistent connection to storage.

For more information, see the following resources:

- Call a web API from ASP.NET Core Blazor WebAssembly
- Secure ASP.NET Core Blazor WebAssembly
- Blazor *Security and Identity* articles

For more information on Azure data storage options, see the following:

- Azure Databases
- Azure Storage Documentation

URL

For transient data representing navigation state, model the data as a part of the URL. Examples of user state modeled in the URL include:

- The ID of a viewed entity.
- The current page number in a paged grid.

The contents of the browser's address bar are retained if the user manually reloads the page.

For information on defining URL patterns with the `@page` directive, see ASP.NET Core Blazor routing.

Browser storage

For transient data that the user is actively creating, a commonly used storage location is the browser's `localStorage` and `sessionStorage` collections:

- `localStorage` is scoped to the browser's window. If the user reloads the page or closes and re-opens the browser, the state persists. If the user opens multiple browser tabs, the state is shared across the tabs. Data persists in `localStorage` until explicitly cleared.
- `sessionStorage` is scoped to the browser tab. If the user reloads the tab, the state persists. If the user closes the tab or the browser, the state is lost. If the user opens multiple browser tabs, each tab has its own independent version of the data.

> **NOTE**
>
> `localStorage` and `sessionStorage` can be used in Blazor WebAssembly apps but only by writing custom code or using a third-party package.

Generally, `sessionStorage` is safer to use. `sessionStorage` avoids the risk that a user opens multiple tabs and encounters the following:

- Bugs in state storage across tabs.

- Confusing behavior when a tab overwrites the state of other tabs.

`localStorage` is the better choice if the app must persist state across closing and re-opening the browser.

> **WARNING**
> Users may view or tamper with the data stored in `localStorage` and `sessionStorage`.

Additional resources

- Save app state before an authentication operation
- Call a web API from ASP.NET Core Blazor WebAssembly
- Secure ASP.NET Core Blazor WebAssembly

Blazor Server is a stateful app framework. Most of the time, the app maintains a connection to the server. The user's state is held in the server's memory in a *circuit*.

Examples of user state held in a circuit include:

- The hierarchy of component instances and their most recent render output in the rendered UI.
- The values of fields and properties in component instances.
- Data held in dependency injection (DI) service instances that are scoped to the circuit.

User state might also be found in JavaScript variables in the browser's memory set via JavaScript interop calls.

If a user experiences a temporary network connection loss, Blazor attempts to reconnect the user to their original circuit with their original state. However, reconnecting a user to their original circuit in the server's memory isn't always possible:

- The server can't retain a disconnected circuit forever. The server must release a disconnected circuit after a timeout or when the server is under memory pressure.
- In multi-server, load-balanced deployment environments, individual servers may fail or be automatically removed when no longer required to handle the overall volume of requests. The original server processing requests for a user may become unavailable when the user attempts to reconnect.
- The user might close and re-open their browser or reload the page, which removes any state held in the browser's memory. For example, JavaScript variable values set through JavaScript interop calls are lost.

When a user can't be reconnected to their original circuit, the user receives a new circuit with an empty state. This is equivalent to closing and re-opening a desktop app.

Persist state across circuits

Generally, maintain state across circuits where users are actively creating data, not simply reading data that already exists.

To preserve state across circuits, the app must persist the data to some other storage location than the server's memory. State persistence isn't automatic. You must take steps when developing the app to implement stateful data persistence.

Data persistence is typically only required for high-value state that users expended effort to create. In the following examples, persisting state either saves time or aids in commercial activities:

- Multi-step web forms: It's time-consuming for a user to re-enter data for several completed steps of a multi-step web form if their state is lost. A user loses state in this scenario if they navigate away from the form and return later.

- Shopping carts: Any commercially important component of an app that represents potential revenue can be maintained. A user who loses their state, and thus their shopping cart, may purchase fewer products or services when they return to the site later.

An app can only persist *app state*. UIs can't be persisted, such as component instances and their render trees. Components and render trees aren't generally serializable. To persist UI state, such as the expanded nodes of a tree view control, the app must use custom code to model the behavior of the UI state as serializable app state.

Where to persist state

Three common locations exist for persisting state:

- Server-side storage
- URL
- Browser storage

Server-side storage

For permanent data persistence that spans multiple users and devices, the app can use server-side storage. Options include:

- Blob storage
- Key-value storage
- Relational database
- Table storage

After data is saved, the user's state is retained and available in any new circuit.

For more information on Azure data storage options, see the following:

- Azure Databases
- Azure Storage Documentation

URL

For transient data representing navigation state, model the data as a part of the URL. Examples of user state modeled in the URL include:

- The ID of a viewed entity.
- The current page number in a paged grid.

The contents of the browser's address bar are retained:

- If the user manually reloads the page.
- If the web server becomes unavailable, and the user is forced to reload the page in order to connect to a different server.

For information on defining URL patterns with the `@page` directive, see ASP.NET Core Blazor routing.

Browser storage

For transient data that the user is actively creating, a commonly used storage location is the browser's `localStorage` and `sessionStorage` collections:

- `localStorage` is scoped to the browser's window. If the user reloads the page or closes and re-opens the browser, the state persists. If the user opens multiple browser tabs, the state is shared across the tabs. Data persists in `localStorage` until explicitly cleared.
- `sessionStorage` is scoped to the browser tab. If the user reloads the tab, the state persists. If the user closes the tab or the browser, the state is lost. If the user opens multiple browser tabs, each tab has its own independent

version of the data.

Generally, `sessionStorage` is safer to use. `sessionStorage` avoids the risk that a user opens multiple tabs and encounters the following:

- Bugs in state storage across tabs.
- Confusing behavior when a tab overwrites the state of other tabs.

`localStorage` is the better choice if the app must persist state across closing and re-opening the browser.

Caveats for using browser storage:

- Similar to the use of a server-side database, loading and saving data are asynchronous.
- Unlike a server-side database, storage isn't available during prerendering because the requested page doesn't exist in the browser during the prerendering stage.
- Storage of a few kilobytes of data is reasonable to persist for Blazor Server apps. Beyond a few kilobytes, you must consider the performance implications because the data is loaded and saved across the network.
- Users may view or tamper with the data. ASP.NET Core Data Protection can mitigate the risk. For example, ASP.NET Core Protected Browser Storage uses ASP.NET Core Data Protection.

Third-party NuGet packages provide APIs for working with `localStorage` and `sessionStorage`. It's worth considering choosing a package that transparently uses ASP.NET Core Data Protection. Data Protection encrypts stored data and reduces the potential risk of tampering with stored data. If JSON-serialized data is stored in plain text, users can see the data using browser developer tools and also modify the stored data. Securing data isn't always a problem because the data might be trivial in nature. For example, reading or modifying the stored color of a UI element isn't a significant security risk to the user or the organization. Avoid allowing users to inspect or tamper with *sensitive data*.

ASP.NET Core Protected Browser Storage

ASP.NET Core Protected Browser Storage leverages ASP.NET Core Data Protection for `localStorage` and `sessionStorage`.

> **NOTE**
> Protected Browser Storage relies on ASP.NET Core Data Protection and is only supported for Blazor Server apps.

Configuration

1. Add a package reference to `Microsoft.AspNetCore.Components.Web.Extensions`.
2. In `Startup.ConfigureServices`, call `AddProtectedBrowserStorage` to add `localStorage` and `sessionStorage` services to the service collection:

   ```
   services.AddProtectedBrowserStorage();
   ```

Save and load data within a component

In any component that requires loading or saving data to browser storage, use the `@inject` directive to inject an instance of either of the following:

- `ProtectedLocalStorage`
- `ProtectedSessionStorage`

The choice depends on which browser storage location you wish to use. In the following example, `sessionStorage` is used:

```
@using Microsoft.AspNetCore.Components.Web.Extensions
@inject ProtectedSessionStorage ProtectedSessionStore
```

The `@using` directive can be placed in the app's `_Imports.razor` file instead of in the component. Use of the `_Imports.razor` file makes the namespace available to larger segments of the app or the whole app.

To persist the `currentCount` value in the `Counter` component of an app based on the Blazor Server project template, modify the `IncrementCount` method to use `ProtectedSessionStore.SetAsync`:

```
private async Task IncrementCount()
{
    currentCount++;
    await ProtectedSessionStore.SetAsync("count", currentCount);
}
```

In larger, more realistic apps, storage of individual fields is an unlikely scenario. Apps are more likely to store entire model objects that include complex state. `ProtectedSessionStore` automatically serializes and deserializes JSON data to store complex state objects.

In the preceding code example, the `currentCount` data is stored as `sessionStorage['count']` in the user's browser. The data isn't stored in plain text but rather is protected using ASP.NET Core Data Protection. The encrypted data can be inspected if `sessionStorage['count']` is evaluated in the browser's developer console.

To recover the `currentCount` data if the user returns to the `Counter` component later, including if the user is on a new circuit, use `ProtectedSessionStore.GetAsync`:

```
protected override async Task OnInitializedAsync()
{
    var result = await ProtectedSessionStore.GetAsync<int>("count");
    currentCount = result.Success ? result.Value : 0;
}
```

If the component's parameters include navigation state, call `ProtectedSessionStore.GetAsync` and assign a non-`null` result in OnParametersSetAsync, not OnInitializedAsync. OnInitializedAsync is only called once when the component is first instantiated. OnInitializedAsync isn't called again later if the user navigates to a different URL while remaining on the same page. For more information, see ASP.NET Core Blazor lifecycle.

> **WARNING**
>
> The examples in this section only work if the server doesn't have prerendering enabled. With prerendering enabled, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.
>
> Either disable prerendering or add additional code to work with prerendering. To learn more about writing code that works with prerendering, see the Handle prerendering section.

Handle the loading state

Since browser storage is accessed asynchronously over a network connection, there's always a period of time before the data is loaded and available to a component. For the best results, render a loading-state message while loading is in progress instead of displaying blank or default data.

One approach is to track whether the data is `null`, which means that the data is still loading. In the default `Counter` component, the count is held in an `int`. Make `currentCount` nullable by adding a question mark (`?`) to the type (`int`):

```
private int? currentCount;
```

Instead of unconditionally displaying the count and `Increment` button, display these elements only if the data is loaded by checking HasValue:

```
@if (currentCount.HasValue)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}
```

Handle prerendering

During prerendering:

- An interactive connection to the user's browser doesn't exist.
- The browser doesn't yet have a page in which it can run JavaScript code.

`localStorage` or `sessionStorage` aren't available during prerendering. If the component attempts to interact with storage, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.

One way to resolve the error is to disable prerendering. This is usually the best choice if the app makes heavy use of browser-based storage. Prerendering adds complexity and doesn't benefit the app because the app can't prerender any useful content until `localStorage` or `sessionStorage` are available.

To disable prerendering, open the `Pages/_Host.cshtml` file and change the `render-mode` attribute of the Component Tag Helper to Server:

```
<component type="typeof(App)" render-mode="Server" />
```

Prerendering might be useful for other pages that don't use `localStorage` or `sessionStorage`. To retain prerendering, defer the loading operation until the browser is connected to the circuit. The following is an example for storing a counter value:

```
@using Microsoft.AspNetCore.Components.Web.Extensions
@inject ProtectedLocalStorage ProtectedLocalStore

@if (isConnected)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}

@code {
    private int currentCount;
    private bool isConnected;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            isConnected = true;
            await LoadStateAsync();
            StateHasChanged();
        }
    }

    private async Task LoadStateAsync()
    {
        var result = await ProtectedLocalStore.GetAsync<int>("count");
        currentCount = result.Success ? result.Value : 0;
    }

    private async Task IncrementCount()
    {
        currentCount++;
        await ProtectedLocalStore.SetAsync("count", currentCount);
    }
}
```

Factor out the state preservation to a common location

If many components rely on browser-based storage, re-implementing state provider code many times creates code duplication. One option for avoiding code duplication is to create a *state provider parent component* that encapsulates the state provider logic. Child components can work with persisted data without regard to the state persistence mechanism.

In the following example of a `CounterStateProvider` component, counter data is persisted to `sessionStorage`:

```
@using Microsoft.AspNetCore.Components.Web.Extensions
@inject ProtectedSessionStorage ProtectedSessionStore

@if (isLoaded)
{
    <CascadingValue Value="@this">
        @ChildContent
    </CascadingValue>
}
else
{
    <p>Loading...</p>
}

@code {
    private bool isLoaded;

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    public int CurrentCount { get; set; }

    protected override async Task OnInitializedAsync()
    {
        var result = await ProtectedSessionStore.GetAsync<int>("count");
        currentCount = result.Success ? result.Value : 0;
        isLoaded = true;
    }

    public async Task SaveChangesAsync()
    {
        await ProtectedSessionStore.SetAsync("count", CurrentCount);
    }
}
```

The `CounterStateProvider` component handles the loading phase by not rendering its child content until loading is complete.

To use the `CounterStateProvider` component, wrap an instance of the component around any other component that requires access to the counter state. To make the state accessible to all components in an app, wrap the `CounterStateProvider` component around the Router in the `App` component (`App.razor`):

```
<CounterStateProvider>
    <Router AppAssembly="typeof(Startup).Assembly">
        ...
    </Router>
</CounterStateProvider>
```

Wrapped components receive and can modify the persisted counter state. The following `Counter` component implements the pattern:

```
@page "/counter"

<p>Current count: <strong>@CounterStateProvider.CurrentCount</strong></p>
<button @onclick="IncrementCount">Increment</button>

@code {
    [CascadingParameter]
    private CounterStateProvider CounterStateProvider { get; set; }

    private async Task IncrementCount()
    {
        CounterStateProvider.CurrentCount++;
        await CounterStateProvider.SaveChangesAsync();
    }
}
```

The preceding component isn't required to interact with `ProtectedBrowserStorage`, nor does it deal with a "loading" phase.

To deal with prerendering as described earlier, `CounterStateProvider` can be amended so that all of the components that consume the counter data automatically work with prerendering. For more information, see the Handle prerendering section.

In general, the *state provider parent component* pattern is recommended:

- To consume state across many components.
- If there's just one top-level state object to persist.

To persist many different state objects and consume different subsets of objects in different places, it's better to avoid persisting state globally.

Protected Browser Storage experimental NuGet package

ASP.NET Core Protected Browser Storage leverages ASP.NET Core Data Protection for `localStorage` and `sessionStorage`.

> **WARNING**
>
> `Microsoft.AspNetCore.ProtectedBrowserStorage` is an unsupported, experimental package unsuitable for production use.
>
> The package is only available for use in ASP.NET Core 3.1 Blazor Server apps.

Configuration

1. Add a package reference to `Microsoft.AspNetCore.ProtectedBrowserStorage`.
2. In the `Pages/_Host.cshtml` file, add the following script inside the closing `</body>` tag:

   ```
   <script src="_content/Microsoft.AspNetCore.ProtectedBrowserStorage/protectedBrowserStorage.js"></script>
   ```

3. In `Startup.ConfigureServices`, call `AddProtectedBrowserStorage` to add `localStorage` and `sessionStorage` services to the service collection:

   ```
   services.AddProtectedBrowserStorage();
   ```

Save and load data within a component

In any component that requires loading or saving data to browser storage, use the `@inject` directive to inject an instance of either of the following:

- `ProtectedLocalStorage`
- `ProtectedSessionStorage`

The choice depends on which browser storage location you wish to use. In the following example, `sessionStorage` is used:

```
@using Microsoft.AspNetCore.ProtectedBrowserStorage
@inject ProtectedSessionStorage ProtectedSessionStore
```

The `@using` statement can be placed into an `_Imports.razor` file instead of in the component. Use of the `_Imports.razor` file makes the namespace available to larger segments of the app or the whole app.

To persist the `currentCount` value in the `Counter` component of an app based on the Blazor Server project template, modify the `IncrementCount` method to use `ProtectedSessionStore.SetAsync`:

```
private async Task IncrementCount()
{
    currentCount++;
    await ProtectedSessionStore.SetAsync("count", currentCount);
}
```

In larger, more realistic apps, storage of individual fields is an unlikely scenario. Apps are more likely to store entire model objects that include complex state. `ProtectedSessionStore` automatically serializes and deserializes JSON data.

In the preceding code example, the `currentCount` data is stored as `sessionStorage['count']` in the user's browser. The data isn't stored in plain text but rather is protected using ASP.NET Core Data Protection. The encrypted data can be inspected if `sessionStorage['count']` is evaluated in the browser's developer console.

To recover the `currentCount` data if the user returns to the `Counter` component later, including if they're on an entirely new circuit, use `ProtectedSessionStore.GetAsync`:

```
protected override async Task OnInitializedAsync()
{
    currentCount = await ProtectedSessionStore.GetAsync<int>("count");
}
```

If the component's parameters include navigation state, call `ProtectedSessionStore.GetAsync` and assign the result in OnParametersSetAsync, not OnInitializedAsync. OnInitializedAsync is only called once when the component is first instantiated. OnInitializedAsync isn't called again later if the user navigates to a different URL while remaining on the same page. For more information, see ASP.NET Core Blazor lifecycle.

> **WARNING**
>
> The examples in this section only work if the server doesn't have prerendering enabled. With prerendering enabled, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.
>
> Either disable prerendering or add additional code to work with prerendering. To learn more about writing code that works with prerendering, see the Handle prerendering section.

Handle the loading state

Since browser storage is accessed asynchronously over a network connection, there's always a period of time

before the data is loaded and available to a component. For the best results, render a loading-state message while loading is in progress instead of displaying blank or default data.

One approach is to track whether the data is `null`, which means that the data is still loading. In the default `Counter` component, the count is held in an `int`. Make `currentCount` nullable by adding a question mark (`?`) to the type (`int`):

```
private int? currentCount;
```

Instead of unconditionally displaying the count and `Increment` button, choose to display these elements only if the data is loaded:

```
@if (currentCount.HasValue)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}
```

Handle prerendering

During prerendering:

- An interactive connection to the user's browser doesn't exist.
- The browser doesn't yet have a page in which it can run JavaScript code.

`localStorage` or `sessionStorage` aren't available during prerendering. If the component attempts to interact with storage, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.

One way to resolve the error is to disable prerendering. This is usually the best choice if the app makes heavy use of browser-based storage. Prerendering adds complexity and doesn't benefit the app because the app can't prerender any useful content until `localStorage` or `sessionStorage` are available.

To disable prerendering, open the `Pages/_Host.cshtml` file and change the `render-mode` attribute of the Component Tag Helper to Server:

```
<component type="typeof(App)" render-mode="Server" />
```

Prerendering might be useful for other pages that don't use `localStorage` or `sessionStorage`. To retain prerendering, defer the loading operation until the browser is connected to the circuit. The following is an example for storing a counter value:

```
@using Microsoft.AspNetCore.ProtectedBrowserStorage
@inject ProtectedLocalStorage ProtectedLocalStore

@if (isConnected)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}

@code {
    private int? currentCount;
    private bool isConnected = false;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            isConnected = true;
            await LoadStateAsync();
            StateHasChanged();
        }
    }

    private async Task LoadStateAsync()
    {
        currentCount = await ProtectedLocalStore.GetAsync<int>("count");
    }

    private async Task IncrementCount()
    {
        currentCount++;
        await ProtectedLocalStore.SetAsync("count", currentCount);
    }
}
```

Factor out the state preservation to a common location

If many components rely on browser-based storage, re-implementing state provider code many times creates code duplication. One option for avoiding code duplication is to create a *state provider parent component* that encapsulates the state provider logic. Child components can work with persisted data without regard to the state persistence mechanism.

In the following example of a `CounterStateProvider` component, counter data is persisted to `sessionStorage`:

```
@using Microsoft.AspNetCore.ProtectedBrowserStorage
@inject ProtectedSessionStorage ProtectedSessionStore

@if (isLoaded)
{
    <CascadingValue Value="@this">
        @ChildContent
    </CascadingValue>
}
else
{
    <p>Loading...</p>
}

@code {
    private bool isLoaded;

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    public int CurrentCount { get; set; }

    protected override async Task OnInitializedAsync()
    {
        CurrentCount = await ProtectedSessionStore.GetAsync<int>("count");
        isLoaded = true;
    }

    public async Task SaveChangesAsync()
    {
        await ProtectedSessionStore.SetAsync("count", CurrentCount);
    }
}
```

The `CounterStateProvider` component handles the loading phase by not rendering its child content until loading is complete.

To use the `CounterStateProvider` component, wrap an instance of the component around any other component that requires access to the counter state. To make the state accessible to all components in an app, wrap the `CounterStateProvider` component around the Router in the `App` component (`App.razor`):

```
<CounterStateProvider>
    <Router AppAssembly="typeof(Startup).Assembly">
        ...
    </Router>
</CounterStateProvider>
```

Wrapped components receive and can modify the persisted counter state. The following `Counter` component implements the pattern:

```
@page "/counter"

<p>Current count: <strong>@CounterStateProvider.CurrentCount</strong></p>
<button @onclick="IncrementCount">Increment</button>

@code {
    [CascadingParameter]
    private CounterStateProvider CounterStateProvider { get; set; }

    private async Task IncrementCount()
    {
        CounterStateProvider.CurrentCount++;
        await CounterStateProvider.SaveChangesAsync();
    }
}
```

The preceding component isn't required to interact with `ProtectedBrowserStorage`, nor does it deal with a "loading" phase.

To deal with prerendering as described earlier, `CounterStateProvider` can be amended so that all of the components that consume the counter data automatically work with prerendering. For more information, see the Handle prerendering section.

In general, *state provider parent component* pattern is recommended:

- To consume state across many components.
- If there's just one top-level state object to persist.

To persist many different state objects and consume different subsets of objects in different places, it's better to avoid persisting state globally.

Debug ASP.NET Core Blazor WebAssembly

9/22/2020 • 10 minutes to read • Edit Online

Daniel Roth

Blazor WebAssembly apps can be debugged using the browser dev tools in Chromium-based browsers (Edge/Chrome). You can also debug your app using the following integrated development environments (IDEs):

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

Available scenarios include:

- Set and remove breakpoints.
- Run the app with debugging support in IDEs.
- Single-step through the code.
- Resume code execution with a keyboard shortcut in IDEs.
- In the *Locals* window, observe the values of local variables.
- See the call stack, including call chains between JavaScript and .NET.

For now, you *can't*:

- Break on unhandled exceptions.
- Hit breakpoints during app startup before the debug proxy is running. This includes breakpoints in `Program.Main` (`Program.cs`) and breakpoints in the `OnInitialized{Async}` methods of components that are loaded by the first page requested from the app.

Prerequisites

Debugging requires either of the following browsers:

- Google Chrome (version 70 or later) (default)
- Microsoft Edge (version 80 or later)

Visual Studio for Mac requires version 8.8 (build 1532) or later:

1. Install the latest release of Visual Studio for Mac by selecting the **Download Visual Studio for Mac** button at Microsoft: Visual Studio for Mac.
2. Select the *Preview* channel from within Visual Studio. For more information, see Install a preview version of Visual Studio for Mac.

> **NOTE**
> Apple Safari on macOS isn't currently supported.

Enable debugging

To enable debugging for an existing Blazor WebAssembly app, update the `launchSettings.json` file in the startup project to include the following `inspectUri` property in each launch profile:

```
"inspectUri": "{wsProtocol}://{url.hostname}:{url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}"
```

Once updated, the `launchSettings.json` file should look similar to the following example:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:50454",
      "sslPort": 44399
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "inspectUri": "{wsProtocol}://{url.hostname}:{url.port}/_framework/debug/ws-proxy?browser=
{browserInspectUri}",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "BlazorApp1.Server": {
      "commandName": "Project",
      "launchBrowser": true,
      "inspectUri": "{wsProtocol}://{url.hostname}:{url.port}/_framework/debug/ws-proxy?browser=
{browserInspectUri}",
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The `inspectUri` property:

- Enables the IDE to detect that the app is a Blazor WebAssembly app.
- Instructs the script debugging infrastructure to connect to the browser through Blazor's debugging proxy.

The placeholder values for the WebSockets protocol (`wsProtocol`), host (`url.hostname`), port (`url.port`), and inspector URI on the launched browser (`browserInspectUri`) are provided by the framework.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

To debug a Blazor WebAssembly app in Visual Studio:

1. Create a new ASP.NET Core hosted Blazor WebAssembly app.
2. Press F5 to run the app in the debugger.

   > **NOTE**
   >
   > **Start Without Debugging** (`Ctrl+F5`) isn't supported. When the app is run in Debug configuration, debugging overhead always results in a small performance reduction.

3. In the *Client* app, set a breakpoint on the `currentCount++;` line in `Pages/Counter.razor`.

4. In the browser, navigate to `Counter` page and select the **Click me** button to hit the breakpoint.
5. In Visual Studio, inspect the value of the `currentCount` field in the **Locals** window.
6. Press F5 to continue execution.

While debugging a Blazor WebAssembly app, you can also debug server code:

1. Set a breakpoint in the `Pages/FetchData.razor` page in OnInitializedAsync.
2. Set a breakpoint in the `WeatherForecastController` in the `Get` action method.
3. Browse to the `Fetch Data` page to hit the first breakpoint in the `FetchData` component just before it issues an HTTP request to the server.
4. Press F5 to continue execution and then hit the breakpoint on the server in the `WeatherForecastController`.
5. Press F5 again to let execution continue and see the weather forecast table rendered in the browser.

> **NOTE**
>
> Breakpoints are **not** hit during app startup before the debug proxy is running. This includes breakpoints in `Program.Main` (`Program.cs`) and breakpoints in the `OnInitialized{Async}` methods of components that are loaded by the first page requested from the app.

Debug in the browser

The guidance in this section applies to Google Chrome and Microsoft Edge running on Windows.

1. Run a Debug build of the app in the Development environment.
2. Launch a browser and navigate to the app's URL (for example, `https://localhost:5001`).
3. In the browser, attempt to commence remote debugging by pressing `Shift+Alt+d`.

   The browser must be running with remote debugging enabled, which isn't the default. If remote debugging is disabled, an **Unable to find debuggable browser tab** error page is rendered with instructions for launching the browser with the debugging port open. Follow the instructions for your browser, which opens a new browser window. Close the previous browser window.

1. Once the browser is running with remote debugging enabled, the debugging keyboard shortcut in the previous step opens a new debugger tab.
2. After a moment, the **Sources** tab shows a list of the app's .NET assemblies within the `file://` node.
3. In component code (`.razor` files) and C# code files (`.cs`), breakpoints that you set are hit when code executes. After a breakpoint is hit, single-step (`F10`) through the code or resume (`F8`) code execution normally.

Blazor provides a debugging proxy that implements the Chrome DevTools Protocol and augments the protocol with .NET-specific information. When debugging keyboard shortcut is pressed, Blazor points the Chrome DevTools at the proxy. The proxy connects to the browser window you're seeking to debug (hence the need to enable remote debugging).

Browser source maps

Browser source maps allow the browser to map compiled files back to their original source files and are commonly used for client-side debugging. However, Blazor doesn't currently map C# directly to JavaScript/WASM. Instead, Blazor does IL interpretation within the browser, so source maps aren't relevant.

Troubleshoot

If you're running into errors, the following tips may help:

- In the **Debugger** tab, open the developer tools in your browser. In the console, execute `localStorage.clear()` to remove any breakpoints.
- Confirm that you've installed and trusted the ASP.NET Core HTTPS development certificate. For more information, see Enforce HTTPS in ASP.NET Core.
- Visual Studio requires the **Enable JavaScript debugging for ASP.NET (Chrome, Edge and IE)** option in **Tools** > **Options** > **Debugging** > **General**. This is the default setting for Visual Studio. If debugging isn't working, confirm that the option is selected.

Breakpoints in `OnInitialized{Async}` not hit

The Blazor framework's debugging proxy takes a short time to launch, so breakpoints in the `OnInitialized{Async}` lifecycle method might not be hit. We recommend adding a delay at the start of the method body to give the debug proxy some time to launch before the breakpoint is hit. You can include the delay based on an `if` compiler directive to ensure that the delay isn't present for a release build of the app.

OnInitialized:

```
protected override void OnInitialized()
{
#if DEBUG
    Thread.Sleep(10000)
#endif

    ...
}
```

OnInitializedAsync:

```
protected override async Task OnInitializedAsync()
{
#if DEBUG
    await Task.Delay(10000)
#endif

    ...
}
```

Visual Studio (Windows) timeout

If Visual Studio throws an exception that the debug adapter failed to launch mentioning that the timeout was reached, you can adjust the timeout with a Registry setting:

```
VsRegEdit.exe set "<VSInstallFolder>" HKCU JSDebugger\Options\Debugging "BlazorTimeoutInMilliseconds" dword
{TIMEOUT}
```

The `{TIMEOUT}` placeholder in the preceding command is in milliseconds. For example, one minute is assigned as `60000`.

Lazy load assemblies in ASP.NET Core Blazor WebAssembly

9/22/2020 • 5 minutes to read • Edit Online

By Safia Abdalla and Luke Latham

Blazor WebAssembly app startup performance can be improved by deferring the loading of some application assemblies until they are required, which is called *lazy loading*. For example, assemblies that are only used to render a single component can be set up to load only if the user navigates to that component. After loading, the assemblies are cached client-side and are available for all future navigations.

Blazor's lazy loading feature allows you to mark app assemblies for lazy loading, which loads the assemblies during runtime when the user navigates to a particular route. The feature consists of changes to the project file and changes to the application's router.

> **NOTE**
> Assembly lazy loading doesn't benefit Blazor Server apps because assemblies aren't downloaded to the client in a Blazor Server app.

Project file

Mark assemblies for lazy loading in the app's project file (`.csproj`) using the `BlazorWebAssemblyLazyLoad` item. Use the assembly name without the `.dll` extension. The Blazor framework prevents the assemblies specified by this item group from loading at app launch. The following example marks a large custom assembly (`GrantImaharaRobotControls.dll`) for lazy loading. If an assembly that's marked for lazy loading has dependencies, they must also be marked for lazy loading in the project file.

```
<ItemGroup>
  <BlazorWebAssemblyLazyLoad Include="GrantImaharaRobotControls.dll" />
</ItemGroup>
```

`Router` component

Blazor's `Router` component designates which assemblies Blazor searches for routable components. The `Router` component is also responsible for rendering the component for the route where the user navigates. The `Router` component supports an `OnNavigateAsync` feature that can be used in conjunction with lazy loading.

In the app's `Router` component (`App.razor`):

- Add an `OnNavigateAsync` callback. The `OnNavigateAsync` handler is invoked when the user:
  - Visits a route for the first time by navigating to it directly from their browser.
  - Navigates to a new route using a link or a NavigationManager.NavigateTo invocation.
- If lazy-loaded assemblies contain routable components, add a List<Assembly> (for example, named `lazyLoadedAssemblies`) to the component. The assemblies are passed back to the AdditionalAssemblies collection in case the assemblies contain routable components. The framework searches the assemblies for routes and updates the route collection if any new routes are found.

```
@using System.Reflection

<Router AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="@lazyLoadedAssemblies" OnNavigateAsync="@OnNavigateAsync">
    ...
</Router>

@code {
    private List<Assembly> lazyLoadedAssemblies = new List<Assembly>();

    private async Task OnNavigateAsync(NavigationContext args)
    {
    }
}
```

If the `OnNavigateAsync` callback throws an unhandled exception, the Blazor error UI is invoked.

Assembly load logic in `OnNavigateAsync`

`OnNavigateAsync` has a `NavigationContext` parameter that provides information about the current asynchronous navigation event, including the target path (`Path`) and the cancellation token (`CancellationToken`):

- The `Path` property is the user's destination path relative to the app's base path, such as `/robot`.
- The `CancellationToken` can be used to observe the cancellation of the asynchronous task. `OnNavigateAsync` automatically cancels the currently running navigation task when the user navigates to a different page.

Inside `OnNavigateAsync`, implement logic to determine the assemblies to load. Options include:

- Conditional checks inside the `OnNavigateAsync` method.
- A lookup table that maps routes to assembly names, either injected into the component or implemented within the `@code` block.

`LazyAssemblyLoader` is a framework-provided singleton service for loading assemblies. Inject `LazyAssemblyLoader` into the `Router` component:

```
...
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@inject LazyAssemblyLoader assemblyLoader

...
```

The `LazyAssemblyLoader` provides the `LoadAssembliesAsync` method that:

- Uses JS interop to fetch assemblies via a network call.
- Loads assemblies into the runtime executing on WebAssembly in the browser.

The framework's lazy loading implementation supports lazy loading with prerendering in a hosted Blazor solution. During prerendering, all assemblies, including those marked for lazy loading, are assumed to be loaded. Manually register `LazyAssemblyLoader` in the *Server* project's `Startup.ConfigureServices` method (`Startup.cs`):

```
services.AddSingleton<LazyAssemblyLoader>();
```

User interaction with `<Navigating>` content

While loading assemblies, which can take several seconds, the `Router` component can indicate to the user that a page transition is occurring:

- Add an `@using` directive for the Microsoft.AspNetCore.Components.Routing namespace.

- Add a `<Navigating>` tag to the component with markup to display during page transition events.

```
...
@using Microsoft.AspNetCore.Components.Routing
...

<Router ...>
    <Navigating>
        <div style="...">
            <p>Loading the requested page&hellip;</p>
        </div>
    </Navigating>
</Router>

...
```

Handle cancellations in `OnNavigateAsync`

The `NavigationContext` object passed to the `OnNavigateAsync` callback contains a `CancellationToken` that's set when a new navigation event occurs. The `OnNavigateAsync` callback must throw when this cancellation token is set to avoid continuing to run the `OnNavigateAsync` callback on a outdated navigation.

If a user navigates to Route A and then immediately to Route B, the app shouldn't continue running the `OnNavigateAsync` callback for Route A:

```
@inject HttpClient Http
@inject ProductCatalog Products

<Router AppAssembly="@typeof(Program).Assembly"
    OnNavigateAsync="@OnNavigateAsync">
    ...
</Router>

@code {
    private async Task OnNavigateAsync(NavigationContext context)
    {
        if (context.Path == "/about")
        {
            var stats = new Stats = { Page = "/about" };
            await Http.PostAsJsonAsync("api/visited", stats, context.CancellationToken);
        }
        else if (context.Path == "/store")
        {
            var productIds = [345, 789, 135, 689];

            foreach (var productId in productIds)
            {
                context.CancellationToken.ThrowIfCancellationRequested();
                Products.Prefetch(productId);
            }
        }
    }
}
```

> **NOTE**
>
> Not throwing if the cancellation token in `NavigationContext` is canceled can result in unintended behavior, such as rendering a component from a previous navigation.

`OnNavigateAsync` events and renamed assembly files

The resource loader relies on the assembly names that are defined in the `blazor.boot.json` file. If assemblies are renamed, the assembly names used in `OnNavigateAsync` methods and the assembly names in the `blazor.boot.json` file are out of sync.

To rectify this:

- Check to see if the app is running in the Production environment when determining which assembly names to use.
- Store the renamed assembly names in a separate file and read from that file to determine what assembly name to use in the `LazyLoadAssemblyService` and `OnNavigateAsync` methods.

Complete example

The following complete `Router` component demonstrates loading the `GrantImaharaRobotControls.dll` assembly when the user navigates to `/robot`. During page transitions, a styled message is displayed to the user.

```
@using System.Reflection
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@inject LazyAssemblyLoader assemblyLoader

<Router AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="@lazyLoadedAssemblies" OnNavigateAsync="@OnNavigateAsync">
    <Navigating>
        <div style="padding:20px;background-color:blue;color:white">
            <p>Loading the requested page&hellip;</p>
        </div>
    </Navigating>
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>

@code {
    private List<Assembly> lazyLoadedAssemblies = new List<Assembly>();

    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
        {
            if (args.Path.EndsWith("/robot"))
            {
                var assemblies = await assemblyLoader.LoadAssembliesAsync(
                    new List<string>() { "GrantImaharaRobotControls.dll" });
                lazyLoadedAssemblies.AddRange(assemblies);
            }
        }
        catch (Exception ex)
        {
            ...
        }
    }
}
```

Troubleshoot

- If unexpected rendering occurs (for example, a component from a previous navigation is rendered), confirm that the code throws if the cancellation token is set.

- If assemblies are still loaded at application start, check that the assembly is marked as lazy loaded in the project file.

Additional resources

- ASP.NET Core Blazor WebAssembly performance best practices

ASP.NET Core Blazor WebAssembly performance best practices

9/22/2020 • 5 minutes to read • Edit Online

By Pranav Krishnamoorthy

This article provides guidelines for ASP.NET Core Blazor WebAssembly performance best practices.

Avoid unnecessary component renders

Blazor's diffing algorithm avoids rerendering a component when the algorithm perceives that the component hasn't changed. Override ComponentBase.ShouldRender for fine-grained control over component rendering.

If authoring a UI-only component that never changes after the initial render, configure ShouldRender to return `false`:

```
@code {
    protected override bool ShouldRender() => false;
}
```

Most apps don't require fine-grained control, but ShouldRender can be used to selectively render a component responding to a UI event. Using ShouldRender might also be important in scenarios where a large number of components are rendered. Consider a grid, where use of EventCallback in one component in one cell of the grid calls StateHasChanged on the grid. Calling StateHasChanged causes a re-render of every child component. If only a small number of cells require rerendering, use ShouldRender to avoid the performance penalty of unnecessary renders.

In the following example:

- ShouldRender is overridden and set to the value of the ShouldRender field, which is initially `false` when the component loads.
- When the button is selected, ShouldRender is set to `true`, which forces the component to rerender with the updated `currentCount`.
- Immediately after rerendering, OnAfterRender sets the value of ShouldRender back to `false` to prevent further rerendering until the next time the button is selected.

```
<p>Current count: @currentCount</p>

<button @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;
    private bool shouldRender;

    protected override bool ShouldRender() => shouldRender;

    protected override void OnAfterRender(bool first)
    {
        shouldRender = false;
    }

    private void IncrementCount()
    {
        currentCount++;
        shouldRender = true;
    }
}
```

For more information, see ASP.NET Core Blazor lifecycle.

Virtualize re-usable fragments

Components offer a convenient approach to produce re-usable fragments of code and markup. In general, we recommend authoring individual components that best align with the app's requirements. One caveat is that each additional child component contributes to the total time it takes to render a parent component. For most apps, the additional overhead is negligible. Apps that produce a large number of components should consider using strategies to reduce processing overhead, such as limiting the number of rendered components.

For more information, see ASP.NET Core Blazor component virtualization.

Avoid JavaScript interop to marshal data

In Blazor WebAssembly, a JavaScript (JS) interop call must traverse the WebAssembly-JS boundary. Serializing and deserializing content across the two contexts creates processing overhead for the app. Frequent JS interop calls often adversely affects performance. To reduce the marshalling of data across the boundary, determine if the app can consolidate many small payloads into a single large payload to avoid the high volume of context switching between WebAssembly and JS.

Use System.Text.Json

Blazor's JS interop implementation relies on System.Text.Json, which is a high-performance JSON serialization library with low memory allocation. Using System.Text.Json doesn't result in additional app payload size over adding one or more alternate JSON libraries.

For migration guidance, see How to migrate from `Newtonsoft.Json` to `System.Text.Json`.

Use synchronous and unmarshalled JS interop APIs where appropriate

Blazor WebAssembly offers two additional versions of IJSRuntime over the single version available to Blazor Server apps:

- IJSInProcessRuntime allows invoking JS interop calls synchronously, which has less overhead than the asynchronous versions:

```
@inject IJSRuntime JS

@code {
    protected override void OnInitialized()
    {
        var jsInProcess = (IJSInProcessRuntime)JS;

        var value = jsInProcess.Invoke<string>("jsInteropCall");
    }
}
```

- WebAssemblyJSRuntime permits unmarshalled JS interop calls:

```
function jsInteropCall() {
  return BINDING.js_to_mono_obj("Hello world");
}
```

```
@inject IJSRuntime JS

@code {
    protected override void OnInitialized()
    {
        var jsInProcess = (WebAssemblyJSRuntime)JS;

        var value = jsInProcess.InvokeUnmarshalled<string>("jsInteropCall");
    }
}
```

> **WARNING**
> While using WebAssemblyJSRuntime has the least overhead of the JS interop approaches, the JavaScript APIs required to interact with these APIs are currently undocumented and subject to breaking changes in future releases.

Reduce app size

Intermediate Language (IL) trimming

Trimming unused assemblies from a Blazor WebAssembly app reduces the app's size by removing unused code in the app's binaries. By default, the Trimmer is executed when publishing an application. To benefit from trimming, publish the app for deployment using the `dotnet publish` command with the -c|--configuration option set to `Release`:

Intermediate Language (IL) linking

Linking a Blazor WebAssembly app reduces the app's size by trimming unused code in the app's binaries. By default, the Intermediate Language (IL) Linker is only enabled when building in `Release` configuration. To benefit from this, publish the app for deployment using the `dotnet publish` command with the -c|--configuration option set to `Release`:

```
dotnet publish -c Release
```

Lazy load assemblies

Load assemblies at runtime when the assemblies are required by a route. For more information, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Compression

When a Blazor WebAssembly app is published, the output is statically compressed during publish to reduce the app's size and remove the overhead for runtime compression. Blazor relies on the server to perform content negotation and serve statically-compressed files.

After an app is deployed, verify that the app serves compressed files. Inspect the Network tab in a browser's Developer Tools and verify that the files are served with `Content-Encoding: br` or `Content-Encoding: gz`. If the host isn't serving compressed files, follow the instructions in Host and deploy ASP.NET Core Blazor WebAssembly.

Disable unused features

Blazor WebAssembly's runtime includes the following .NET features that can be disabled if the app doesn't require them for a smaller payload size:

- A data file is included to make timezone information correct. If the app doesn't require this feature, consider disabling it by setting the `BlazorEnableTimeZoneSupport` MSBuild property in the app's project file to `false`:

  ```
  <PropertyGroup>
    <BlazorEnableTimeZoneSupport>false</BlazorEnableTimeZoneSupport>
  </PropertyGroup>
  ```

- By default, Blazor WebAssembly carries globalization resources required to display values, such as dates and currency, in the user's culture. If the app doesn't require localization, you may configure the app to support the invariant culture, which is based on the `en-US` culture:

  ```
  <PropertyGroup>
    <InvariantGlobalization>true</InvariantGlobalization>
  </PropertyGroup>
  ```

- Collation information is included to make APIs such as StringComparison.InvariantCultureIgnoreCase work correctly. If you're certain that the app doesn't require the collation data, consider disabling it by setting the `BlazorWebAssemblyPreserveCollationData` MSBuild property in the app's project file to `false`:

  ```
  <PropertyGroup>
    <BlazorWebAssemblyPreserveCollationData>false</BlazorWebAssemblyPreserveCollationData>
  </PropertyGroup>
  ```

Test components in ASP.NET Core Blazor

9/22/2020 • 5 minutes to read • Edit Online

Egil Hansen

Testing is an important aspect of building stable and maintainable software.

To test a Blazor component, the *Component Under Test* (CUT) is:

- Rendered with relevant input for the test.
- Depending on the type of test performed, possibly subject to interaction or modification. For example, event handlers can be triggered, such as an `onclick` event for a button.
- Inspected for expected values.

Test approaches

Two common approaches for testing Blazor components are end-to-end (E2E) testing and unit testing:

- **Unit testing**: Unit tests are written with a unit testing library that provides:
  - Component rendering.
  - Inspection of component output and state.
  - Triggering of event handlers and life cycle methods.
  - Assertions that component behavior is correct.

  bUnit is an example of a library that enables Razor component unit testing.
- **E2E testing**: A test runner runs a Blazor app containing the CUT and automates a browser instance. The testing tool inspects and interacts with the CUT through the browser. Selenium is an example of an E2E testing framework that can be used with Blazor apps.

In unit testing, only the Blazor component (Razor/C#) is involved. External dependencies, such as services and JS interop, must be mocked. In E2E testing, the Blazor component and all of it's auxiliary infrastructure are part of the test, including CSS, JS, and the DOM and browser APIs.

Test scope describes how extensive the tests are. Test scope typically has an influence on the speed of the tests. Unit tests run on a subset of the app's subsystems and usually execute in milliseconds. E2E tests, which test a broad group of the app's subsystems, can take several seconds to complete.

Unit testing also provides access to the instance of the CUT, allowing for inspection and verification of the component's internal state. This normally isn't possible in E2E testing.

With regard to the component's environment, E2E tests must make sure that the expected environmental state has been reached before verification starts. Otherwise, the result is unpredictable. In unit testing, the rendering of the CUT and the life cycle of the test are more integrated, which improves test stability.

E2E testing involves launching multiple processes, network and disk I/O, and other subsystem activity that often lead to poor test reliability. Unit tests are typically insulated from these sorts of issues.

The following table summarizes the difference between the two testing approaches.

CAPABILITY	UNIT TESTING	E2E TESTING

CAPABILITY	UNIT TESTING	E2E TESTING
Test scope	Blazor component (Razor/C#) only	Blazor component (Razor/C#) with CSS/JS
Test execution time	Milliseconds	Seconds
Access to the component instance	Yes	No
Sensitive to the environment	No	Yes
Reliability	More reliable	Less reliable

Choose the most appropriate test approach

Consider the scenario when choosing the type of testing to perform. Some considerations are described in the following table.

SCENARIO	SUGGESTED APPROACH	REMARKS
Component without JS interop logic	Unit testing	When there's no dependency on JS interop in a Blazor component, the component can be tested without access to JS or the DOM API. In this scenario, there are no disadvantages to choosing unit testing.
Component with simple JS interop logic	Unit testing	It's common for components to query the DOM or trigger animations through JS interop. Unit testing is usually preferred in this scenario, since it's straightforward to mock the JS interaction through the IJSRuntime interface.
Component that depends on complex JS code	Unit testing and separate JS testing	If a component uses JS interop to call a large or complex JS library but the interaction between the Blazor component and JS library is simple, then the best approach is likely to treat the component and JS library or code as two separate parts and test each individually. Test the Blazor component with a unit testing library, and test the JS with a JS testing library.
Component with logic that depends on JS manipulation of the browser DOM	E2E testing	When a component's functionality is dependent on JS and its manipulation of the DOM, verify both the JS and Blazor code together in an E2E test. This is the approach that the Blazor framework developers have taken with Blazor's browser rendering logic, which has tightly-coupled C# and JS code. The C# and JS code must work together to correctly render Blazor components in a browser.

SCENARIO	SUGGESTED APPROACH	REMARKS
Component that depends on 3rd party component library with hard-to-mock dependencies	E2E testing	When a component's functionality is dependent on a 3rd party component library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component.

Test components with bUnit

There's no official Microsoft testing framework for Blazor, but the community-driven project bUnit provides a convenient way to unit test Blazor components.

> **NOTE**
> bUnit is a third-party testing library and isn't supported or maintained by Microsoft.

bUnit works with general-purpose testing frameworks, such as MSTest, NUnit, and xUnit. These testing frameworks make bUnit tests look and feel like regular unit tests. bUnit tests integrated with a general-purpose testing framework are ordinarily executed with:

- Visual Studio's Test Explorer.
- `dotnet test` CLI command in a command shell.
- An automated DevOps testing pipeline.

> **NOTE**
> Test concepts and test implementations across different test frameworks are similar but not identical. Refer to the test framework's documentation for guidance.

The following demonstrates the structure of a bUnit test on the `Counter` component in an app based on a Blazor project template. The `Counter` component displays and increments a counter based on the user selecting a button in the page:

```
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

The following bUnit test verifies that the CUT's counter is incremented correctly when the button is selected:

```
[Fact]
public void CounterShouldIncrementWhenSelected()
{
    // Arrange
    using var ctx = new TestContext();
    var cut = ctx.RenderComponent<Counter>();
    var paraElm = cut.Find("p");

    // Act
    cut.Find("button").Click();
    var paraElmText = paraElm.TextContent;

    // Assert
    paraElmText.MarkupMatches("Current count: 1");
}
```

The following actions take place at each step of the test:

- *Arrange*: The `Counter` component is rendered using bUnit's `TestContext`. The CUT's paragraph element (`<p>`) is found and assigned to `paraElm`.
- *Act*: The button's element (`<button>`) is located and then selected by calling `Click`, which should increment the counter and update the content of the paragraph tag (`<p>`). The paragraph element text content is obtained by calling `TextContent`.
- *Assert*: `MarkupMatches` is called on the text content to verify that it matches the expected string, which is `Current count: 1`.

> **NOTE**
>
> The `MarkupMatches` assert method differs from a regular string comparison assertion (for example, `Assert.Equal("Current count: 1", paraElmText);`) `MarkupMatches` performs a semantic comparison of the input and expected HTML markup. A semantic comparison is aware of HTML semantics, meaning things like insignificant whitespace is ignored. This results in more stable tests. For more information, see Customizing the Semantic HTML Comparison.

Additional resources

- Getting Started with bUnit: bUnit instructions include guidance on creating a test project, referencing testing framework packages, and building and running tests.

Build Progressive Web Applications with ASP.NET Core Blazor WebAssembly

9/22/2020 • 16 minutes to read • Edit Online

By Steve Sanderson

A Progressive Web Application (PWA) is usually a Single Page Application (SPA) that uses modern browser APIs and capabilities to behave like a desktop app. Blazor WebAssembly is a standards-based client-side web app platform, so it can use any browser API, including PWA APIs required for the following capabilities:

- Working offline and loading instantly, independent of network speed.
- Running in its own app window, not just a browser window.
- Being launched from the host's operating system start menu, dock, or home screen.
- Receiving push notifications from a backend server, even while the user isn't using the app.
- Automatically updating in the background.

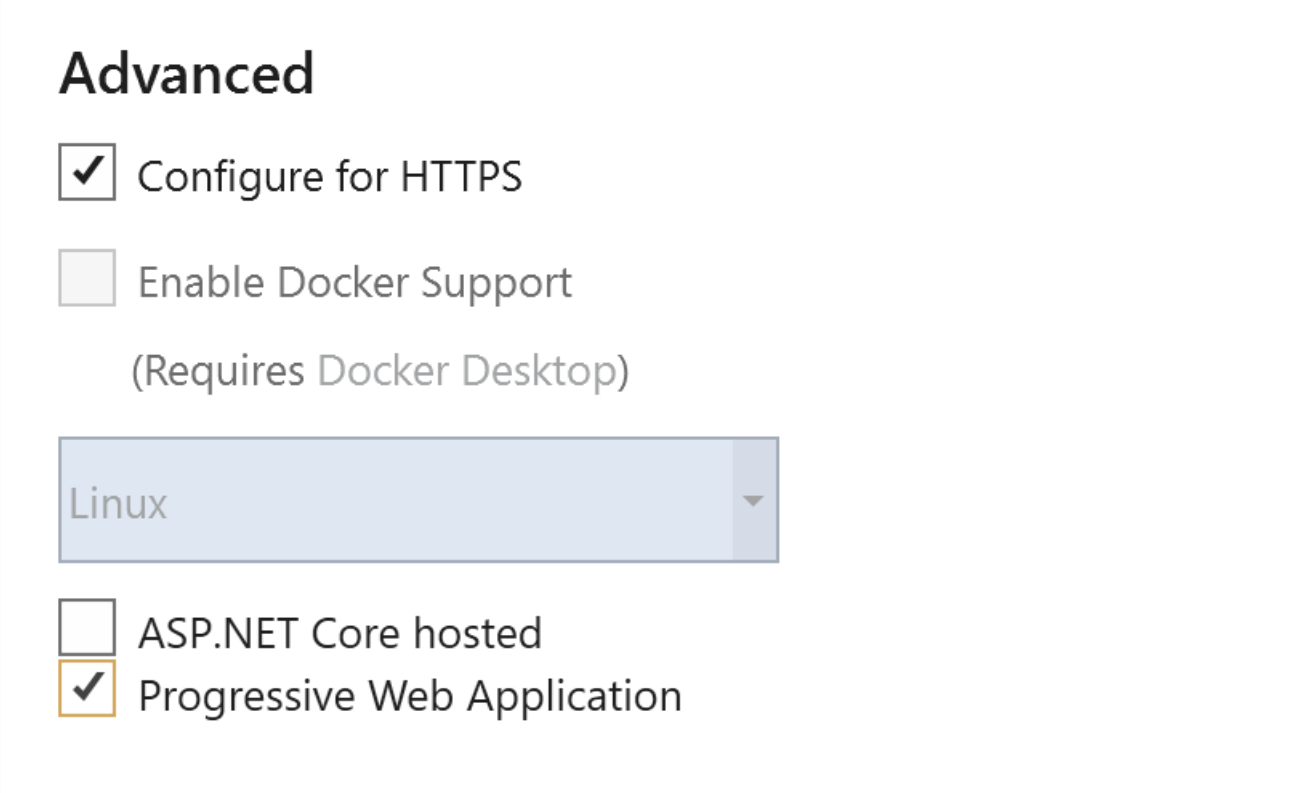
The word *progressive* is used to describe such apps because:

- A user might first discover and use the app within their web browser like any other SPA.
- Later, the user progresses to installing it in their OS and enabling push notifications.

Create a project from the PWA template

- Visual Studio
- Visual Studio Code / .NET Core CLI

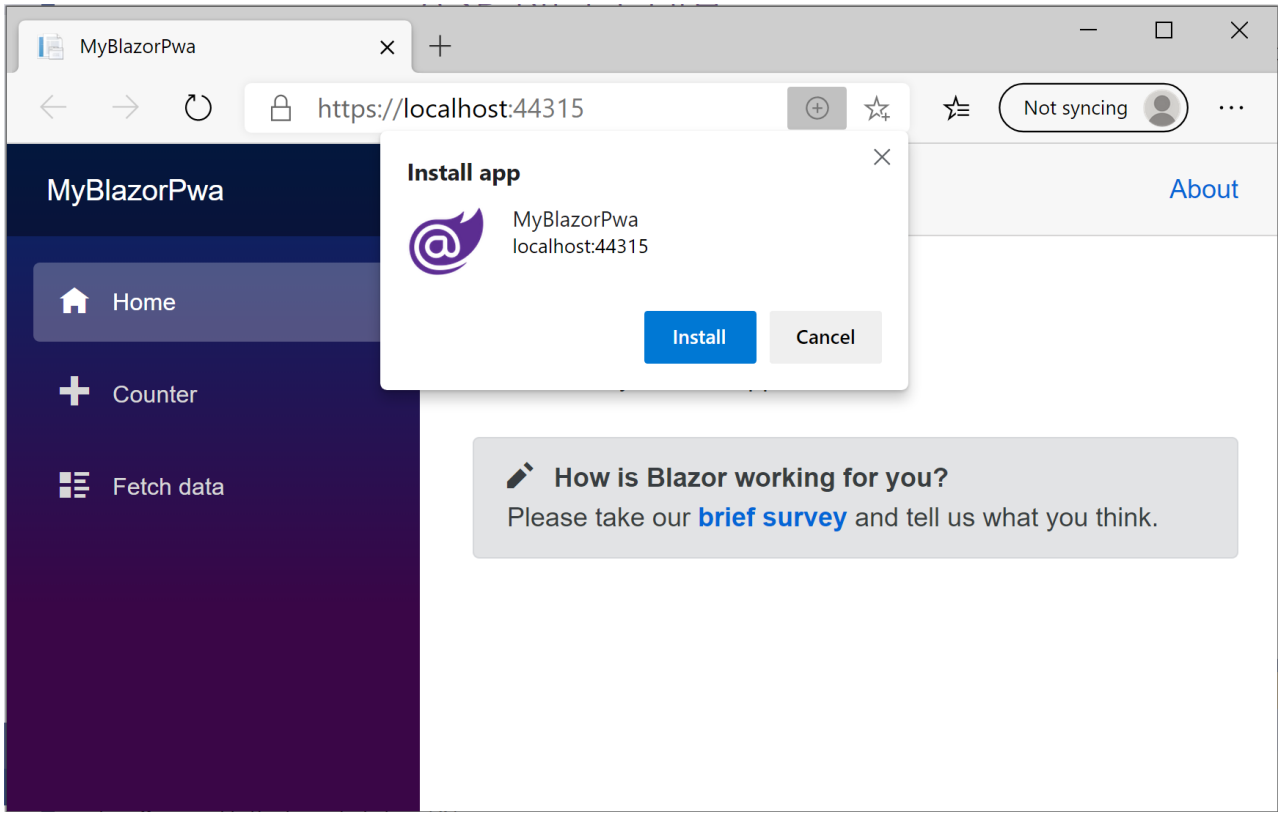
When creating a new **Blazor WebAssembly App** in the **Create a New Project** dialog, select the **Progressive Web Application** check box:



Optionally, PWA can be configured for an app created from the ASP.NET Core Hosted template. The PWA scenario is independent of the hosting model.

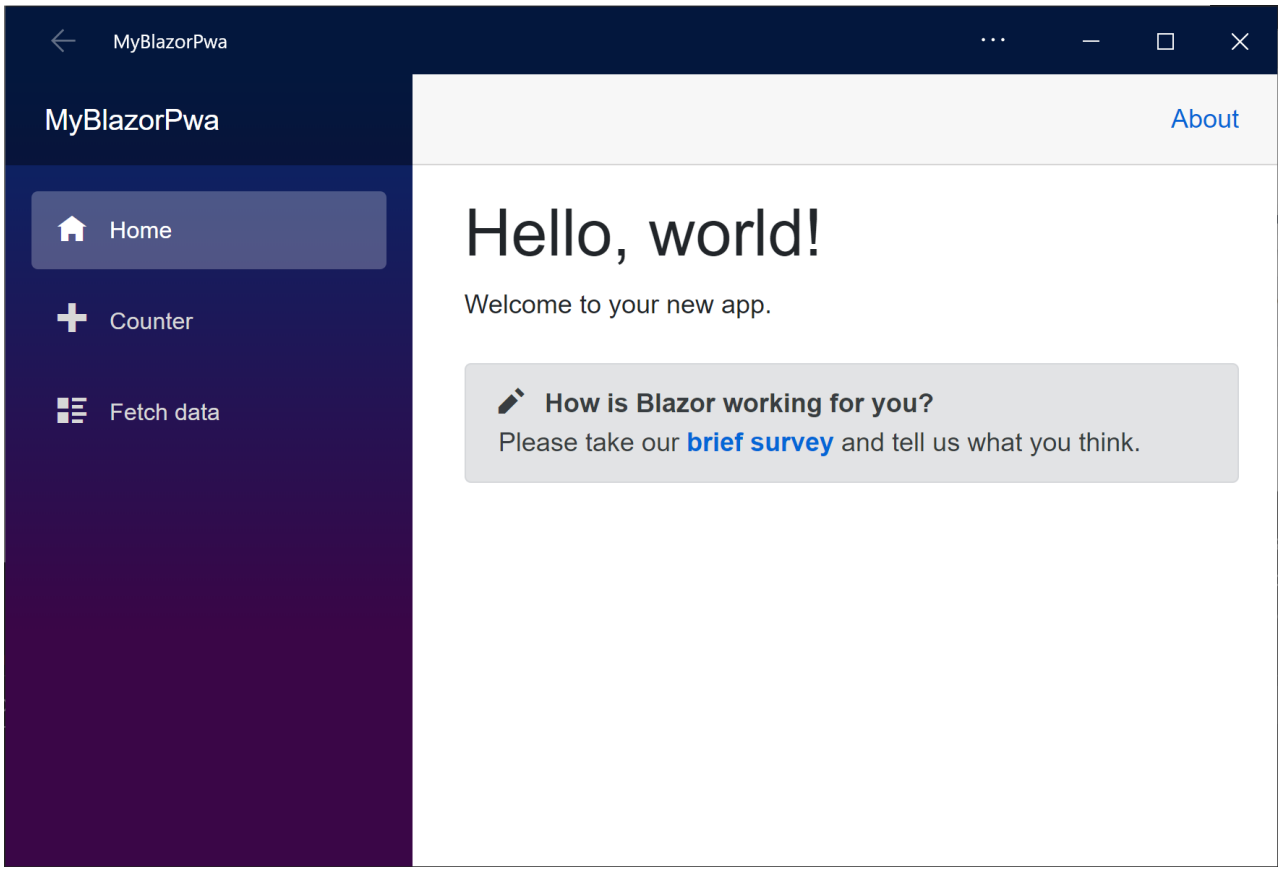
Installation and app manifest

When visiting an app created using the PWA template, users have the option of installing the app into their OS's start menu, dock, or home screen. The way this option is presented depends on the user's browser. When using desktop Chromium-based browsers, such as Edge or Chrome, an **Add** button appears within the URL bar. After the user selects the **Add** button, they receive a confirmation dialog:



On iOS, visitors can install the PWA using Safari's **Share** button and its **Add to Homescreen** option. On Chrome for Android, users should select the **Menu** button in the upper-right corner, followed by **Add to Home screen**.

Once installed, the app appears in its own window without an address bar:



To customize the window's title, color scheme, icon, or other details, see the `manifest.json` file in the project's `wwwroot` directory. The schema of this file is defined by web standards. For more information, see MDN web docs: Web App Manifest.

Offline support

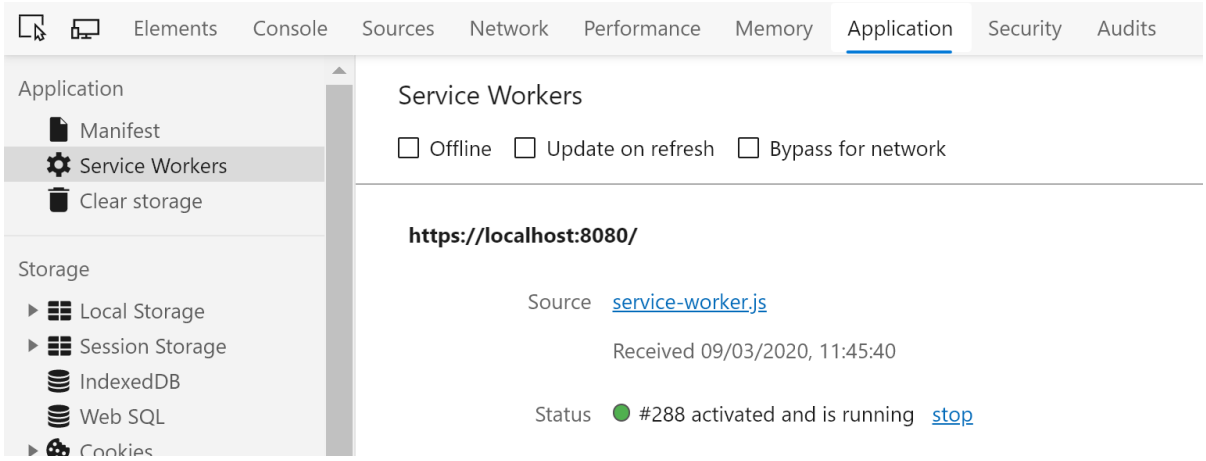
By default, apps created using the PWA template option have support for running offline. A user must first visit the app while they're online. The browser automatically downloads and caches all of the resources required to operate offline.

> **IMPORTANT**
> Development support would interfere with the usual development cycle of making changes and testing them. Therefore, offline support is only enabled for *published* apps.

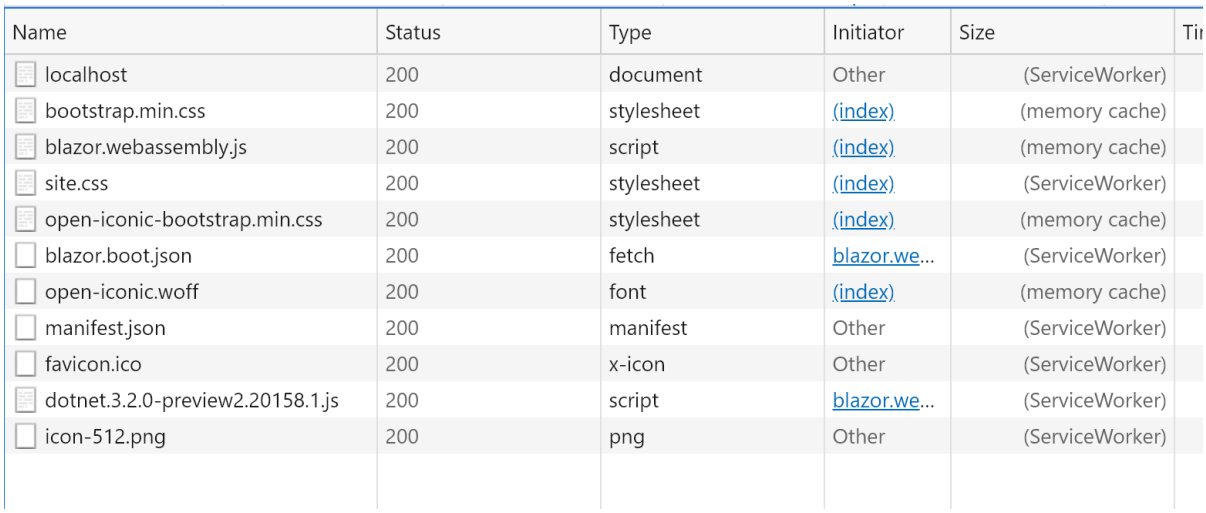
> **WARNING**
> If you intend to distribute an offline-enabled PWA, there are several important warnings and caveats. These scenarios are inherent to offline PWAs and not specific to Blazor. Be sure to read and understand these caveats before making assumptions about how your offline-enabled app will work.

To see how offline support works:

1. Publish the app. For more information, see Host and deploy ASP.NET Core Blazor.
2. Deploy the app to a server that supports HTTPS, and access the app in a browser at its secure HTTPS address.
3. Open the browser's dev tools and verify that a *Service Worker* is registered for the host on the **Application** tab:



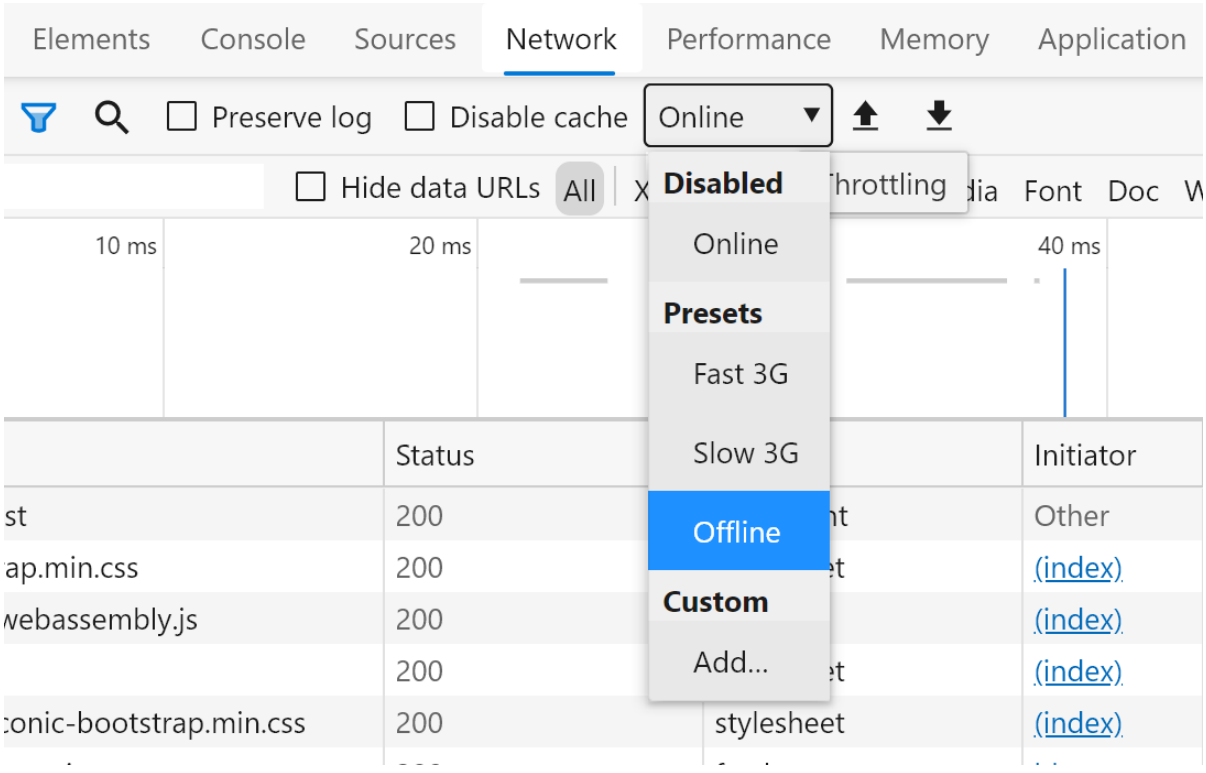
4. Reload the page and examine the **Network** tab. **Service Worker** or **memory cache** are listed as the sources for all of the page's assets:



Name	Status	Type	Initiator	Size	Ti
localhost	200	document	Other	(ServiceWorker)	
bootstrap.min.css	200	stylesheet	(index)	(memory cache)	
blazor.webassembly.js	200	script	(index)	(memory cache)	
site.css	200	stylesheet	(index)	(ServiceWorker)	
open-iconic-bootstrap.min.css	200	stylesheet	(index)	(memory cache)	
blazor.boot.json	200	fetch	blazor.we...	(ServiceWorker)	
open-iconic.woff	200	font	(index)	(memory cache)	
manifest.json	200	manifest	Other	(ServiceWorker)	
favicon.ico	200	x-icon	Other	(ServiceWorker)	
dotnet.3.2.0-preview2.20158.1.js	200	script	blazor.we...	(ServiceWorker)	
icon-512.png	200	png	Other	(ServiceWorker)	

5. To verify that the browser isn't dependent on network access to load the app, either:

   - Shut down the web server and see how the app continues to function normally, which includes page reloads. Likewise, the app continues to function normally when there's a slow network connection.
   - Instruct the browser to simulate offline mode in the **Network** tab:



Offline support using a service worker is a web standard, not specific to Blazor. For more information on service workers, see MDN web docs: Service Worker API. To learn more about common usage patterns for service workers, see Google Web: The Service Worker Lifecycle.

Blazor's PWA template produces two service worker files:

- `wwwroot/service-worker.js`, which is used during development.
- `wwwroot/service-worker.published.js`, which is used after the app is published.

To share logic between the two service worker files, consider the following approach:

- Add a third JavaScript file to hold the common logic.
- Use `self.importScripts` to load the common logic into both service worker files.

Cache-first fetch strategy

The built-in `service-worker.published.js` service worker resolves requests using a *cache-first* strategy. This means that the service worker prefers to return cached content, regardless of whether the user has network access or newer content is available on the server.

The cache-first strategy is valuable because:

- **It ensures reliability.** Network access isn't a boolean state. A user isn't simply online or offline:
  - The user's device may assume it's online, but the network might be so slow as to be impractical to wait for.
  - The network might return invalid results for certain URLs, such as when there's a captive WIFI portal that's currently blocking or redirecting certain requests.

  This is why the browser's `navigator.onLine` API isn't reliable and shouldn't be depended upon.
- **It ensures correctness.** When building a cache of offline resources, the service worker uses content hashing to guarantee it has fetched a complete and self-consistent snapshot of resources at a single instant in time. This cache is then used as an atomic unit. There's no point asking the network for newer resources, since the only versions required are the ones already cached. Anything else risks inconsistency and incompatibility (for example, trying to use versions of .NET assemblies that weren't compiled together).

Background updates

As a mental model, you can think of an offline-first PWA as behaving like a mobile app that can be installed. The app starts up immediately regardless of network connectivity, but the installed app logic comes from a point-in-time snapshot that might not be the latest version.

The Blazor PWA template produces apps that automatically try to update themselves in the background whenever the user visits and has a working network connection. The way this works is as follows:

- During compilation, the project generates a *service worker assets manifest*. By default, this is called `service-worker-assets.js`. The manifest lists all the static resources that the app requires to function offline, such as .NET assemblies, JavaScript files, and CSS, including their content hashes. The resource list is loaded by the service worker so that it knows which resources to cache.
- Each time the user visits the app, the browser re-requests `service-worker.js` and `service-worker-assets.js` in the background. The files are compared byte-for-byte with the existing installed service worker. If the server returns changed content for either of these files, the service worker attempts to install a new version of itself.
- When installing a new version of itself, the service worker creates a new, separate cache for offline resources and starts populating the cache with resources listed in `service-worker-assets.js`. This logic is implemented in the `onInstall` function inside `service-worker.published.js`.
- The process completes successfully when all of the resources are loaded without error and all content hashes match. If successful, the new service worker enters a *waiting for activation* state. As soon as the user closes the

app (no remaining app tabs or windows), the new service worker becomes *active* and is used for subsequent app visits. The old service worker and its cache are deleted.

- If the process doesn't complete successfully, the new service worker instance is discarded. The update process is attempted again on the user's next visit, when hopefully the client has a better network connection that can complete the requests.

Customize this process by editing the service worker logic. None of the preceding behavior is specific to Blazor but is merely the default experience provided by the PWA template option. For more information, see MDN web docs: Service Worker API.

How requests are resolved

As described in the Cache-first fetch strategy section, the default service worker uses a *cache-first* strategy, meaning that it tries to serve cached content when available. If there is no content cached for a certain URL, for example when requesting data from a backend API, the service worker falls back on a regular network request. The network request succeeds if the server is reachable. This logic is implemented inside `onFetch` function within `service-worker.published.js`.

If the app's Razor components rely on requesting data from backend APIs and you want to provide a friendly user experience for failed requests due to network unavailability, implement logic within the app's components. For example, use `try/catch` around HttpClient requests.

Support server-rendered pages

Consider what happens when the user first navigates to a URL such as `/counter` or any other deep link in the app. In these cases, you don't want to return content cached as `/counter`, but instead need the browser to load the content cached as `/index.html` to start up your Blazor WebAssembly app. These initial requests are known as *navigation* requests, as opposed to:

- `subresource` requests for images, stylesheets, or other files.
- `fetch/XHR` requests for API data.

The default service worker contains special-case logic for navigation requests. The service worker resolves the requests by returning the cached content for `/index.html`, regardless of the requested URL. This logic is implemented in the `onFetch` function inside `service-worker.published.js`.

If your app has certain URLs that must return server-rendered HTML, and not serve `/index.html` from the cache, then you need to edit the logic in your service worker. If all URLs containing `/Identity/` need to be handled as regular online-only requests to the server, then modify `service-worker.published.js` `onFetch` logic. Locate the following code:

```
const shouldServeIndexHtml = event.request.mode === 'navigate';
```

Change the code to the following:

```
const shouldServeIndexHtml = event.request.mode === 'navigate'
    && !event.request.url.includes('/Identity/');
```

If you don't do this, then regardless of network connectivity, the service worker intercepts requests for such URLs and resolves them using `/index.html`.

Control asset caching

If your project defines the `ServiceWorkerAssetsManifest` MSBuild property, Blazor's build tooling generates a service worker assets manifest with the specified name. The default PWA template produces a project file containing the following property:

```
<ServiceWorkerAssetsManifest>service-worker-assets.js</ServiceWorkerAssetsManifest>
```

The file is placed in the `wwwroot` output directory, so the browser can retrieve this file by requesting `/service-worker-assets.js`. To see the contents of this file, open `/bin/Debug/{TARGET FRAMEWORK}/wwwroot/service-worker-assets.js` in a text editor. However, don't edit the file, as it's regenerated on each build.

By default, this manifest lists:

- Any Blazor-managed resources, such as .NET assemblies and the .NET WebAssembly runtime files required to function offline.
- All resources for publishing to the app's `wwwroot` directory, such as images, stylesheets, and JavaScript files, including static web assets supplied by external projects and NuGet packages.

You can control which of these resources are fetched and cached by the service worker by editing the logic in `onInstall` in `service-worker.published.js`. By default, the service worker fetches and caches files matching typical web filename extensions such as `.html`, `.css`, `.js`, and `.wasm`, plus file types specific to Blazor WebAssembly (`.dll`, `.pdb`).

To include additional resources that aren't present in the app's `wwwroot` directory, define extra MSBuild `ItemGroup` entries, as shown in the following example:

```
<ItemGroup>
  <ServiceWorkerAssetsManifestItem Include="MyDirectory\AnotherFile.json"
    RelativePath="MyDirectory\AnotherFile.json" AssetUrl="files/AnotherFile.json" />
</ItemGroup>
```

The `AssetUrl` metadata specifies the base-relative URL that the browser should use when fetching the resource to cache. This can be independent of its original source file name on disk.

> **IMPORTANT**
> Adding a `ServiceWorkerAssetsManifestItem` doesn't cause the file to be published in the app's `wwwroot` directory. The publish output must be controlled separately. The `ServiceWorkerAssetsManifestItem` only causes an additional entry to appear in the service worker assets manifest.

Push notifications

Like any other PWA, a Blazor WebAssembly PWA can receive push notifications from a backend server. The server can send push notifications at any time, even when the user isn't actively using the app. For example, push notifications can be sent when a different user performs a relevant action.

The mechanism for sending a push notification is entirely independent of Blazor WebAssembly, since it's implemented by the backend server which can use any technology. If you want to send push notifications from an ASP.NET Core server, consider using a technique similar to the approach taken in the Blazing Pizza workshop.

The mechanism for receiving and displaying a push notification on the client is also independent of Blazor WebAssembly, since it's implemented in the service worker JavaScript file. For an example, see the approach used in the Blazing Pizza workshop.

Caveats for offline PWAs

Not all apps should attempt to support offline use. Offline support adds significant complexity, while not always

being relevant for the use cases required.

Offline support is usually relevant only:

- If the primary data store is local to the browser. For example, the approach is relevant in an app with a UI for an IoT device that stores data in `localStorage` or IndexedDB.
- If the app performs a significant amount of work to fetch and cache the backend API data relevant to each user so that they can navigate through the data offline. If the app must support editing, a system for tracking changes and synchronizing data with the backend must be built.
- If the goal is to guarantee that the app loads immediately regardless of network conditions. Implement a suitable user experience around backend API requests to show the progress of requests and behave gracefully when requests fail due to network unavailability.

Additionally, offline-capable PWAs must deal with a range of additional complications. Developers should carefully familiarize themselves with the caveats in the following sections.

Offline support only when published

During development you typically want to see each change reflected immediately in the browser without going through a background update process. Therefore, Blazor's PWA template enables offline support only when published.

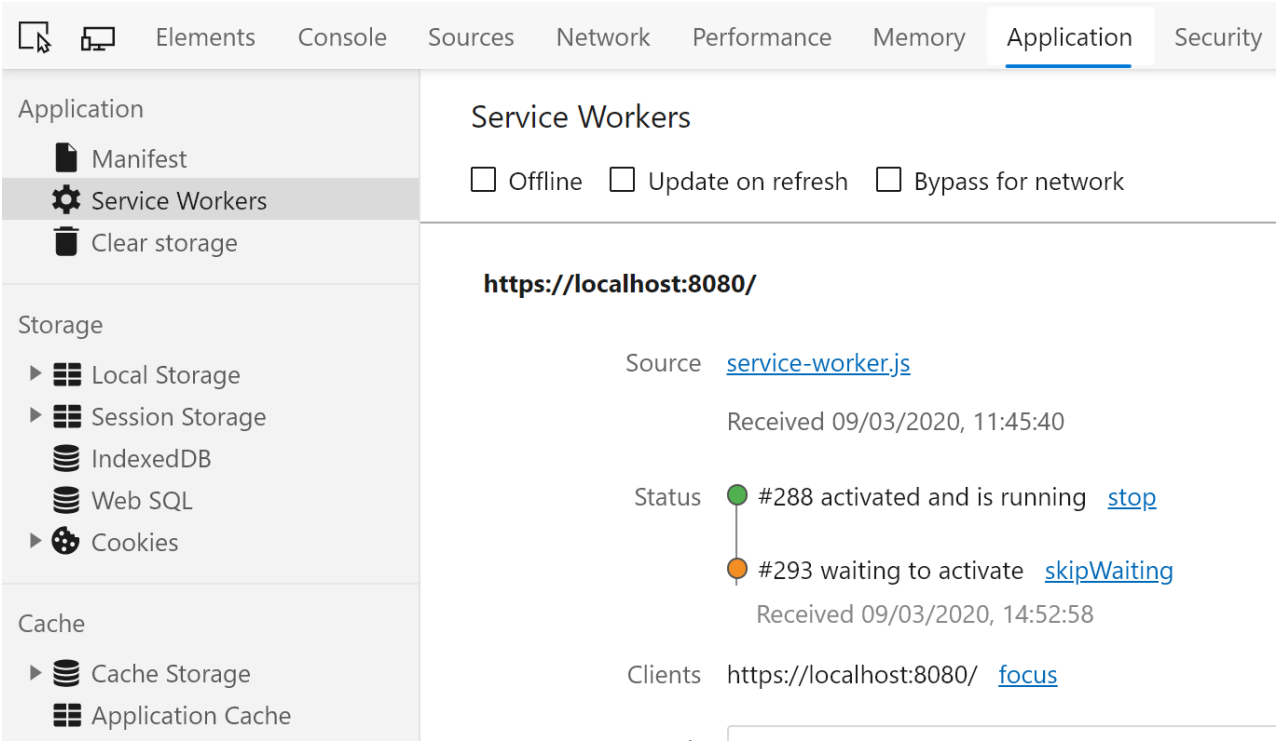
When building an offline-capable app, it's not enough to test the app in the Development environment. You must test the app in its published state to understand how it responds to different network conditions.

Update completion after user navigation away from app

Updates don't complete until the user has navigated away from the app in all tabs. As explained in the Background updates section, after you deploy an update to the app, the browser fetches the updated service worker files to begin the update process.

What surprises many developers is that, even when this update completes, it does **not** take effect until the user has navigated away in all tabs. It is **not** sufficient to refresh the tab displaying the app, even if it's the only tab displaying the app. Until your app is completely closed, the new service worker remains in the *waiting to activate* status. **This is not specific to Blazor, but rather is a standard web platform behavior.**

This commonly troubles developers who are trying to test updates to their service worker or offline cached resources. If you check in the browser's developer tools, you may see something like the following:



For as long as the list of "clients," which are tabs or windows displaying your app, is nonempty, the worker continues waiting. The reason service workers do this is to guarantee consistency. Consistency means that all resources are fetched from the same atomic cache.

When testing changes, you may find it convenient to click the "skipWaiting" link as shown in the preceding screenshot, then reload the page. You can automate this for all users by coding your service worker to skip the "waiting" phase and immediately activate on update. If you skip the waiting phase, you're giving up the guarantee that resources are always fetched consistently from the same cache instance.

Users may run any historical version of the app

Web developers habitually expect that users only run the latest deployed version of their web app, since that's normal within the traditional web distribution model. However, an offline-first PWA is more akin to a native mobile app, where users aren't necessarily running the latest version.

As explained in the Background updates section, after you deploy an update to your app, **each existing user continues to use a previous version for at least one further visit** because the update occurs in the background and isn't activated until the user thereafter navigates away. Plus, the previous version being used isn't necessarily the previous one you deployed. The previous version can be *any* historical version, depending on when the user last completed an update.

This can be an issue if the frontend and backend parts of your app require agreement about the schema for API requests. You must not deploy backward-incompatible API schema changes until you can be sure that all users have upgraded. Alternatively, block users from using incompatible older versions of the app. This scenario requirement is the same as for native mobile apps. If you deploy a breaking change in server APIs, the client app is broken for users who haven't yet updated.

If possible, don't deploy breaking changes to your backend APIs. If you must do so, consider using standard Service Worker APIs such as ServiceWorkerRegistration to determine whether the app is up-to-date, and if not, to prevent usage.

Interference with server-rendered pages

As described in the Support server-rendered pages section, if you want to bypass the service worker's behavior of returning `/index.html` contents for all navigation requests, edit the logic in your service worker.

All service worker asset manifest contents are cached by default

As described in the Control asset caching section, the file `service-worker-assets.js` is generated during build and lists all assets the service worker should fetch and cache.

Since this list by default includes everything emitted to `wwwroot`, including content supplied by external packages and projects, you must be careful not to put too much content there. If the `wwwroot` directory contains millions of images, the service worker tries to fetch and cache them all, consuming excessive bandwidth and most likely not completing successfully.

Implement arbitrary logic to control which subset of the manifest's contents should be fetched and cached by editing the `onInstall` function in `service-worker.published.js`.

Interaction with authentication

The PWA template can be used in conjunction with authentication. An offline-capable PWA can also support authentication when the user has initial network connectivity.

When a user doesn't have network connectivity, they can't authenticate or obtain access tokens. By default, attempting to visit the login page without network access results in a "network error" message. You must design a UI flow that allows the user perform useful tasks while offline without attempting to authenticate the user or obtain access tokens. Alternatively, you can design the app to gracefully fail when the network isn't available. If the app can't be designed to handle these scenarios, you might not want to enable offline support.

When an app that's designed for online and offline use is online again:

- The app might need to provision a new access token.
- The app must detect if a different user is signed into the service so that it can apply operations to the user's account that were made while they were offline.

To create an offline PWA app that interacts with authentication:

- Replace the AccountClaimsPrincipalFactory<TAccount> with a factory that stores the last signed-in user and uses the stored user when the app is offline.
- Queue operations while the app is offline and apply them when the app returns online.
- During sign out, clear the stored user.

The `CarChecker` sample app demonstrates the preceding approaches. See the following parts of the app:

- `OfflineAccountClaimsPrincipalFactory` (`Client/Data/OfflineAccountClaimsPrincipalFactory.cs`)
- `LocalVehiclesStore` (`Client/Data/LocalVehiclesStore.cs`)
- `LoginStatus` component (`Client/Shared/LoginStatus.razor`)

Additional resources

- SignalR cross-origin negotiation for authentication

Host and deploy ASP.NET Core Blazor

9/22/2020 • 4 minutes to read • Edit Online

By Luke Latham, Rainer Stropek, and Daniel Roth

Publish the app

Apps are published for deployment in Release configuration.

- Visual Studio
- Visual Studio for Mac
- .NET Core CLI

1. Select **Build** > **Publish {APPLICATION}** from the navigation bar.
2. Select the *publish target*. To publish locally, select **Folder**.
3. Accept the default location in the **Choose a folder** field or specify a different location. Select the `Publish` button.

Publishing the app triggers a restore of the project's dependencies and builds the project before creating the assets for deployment. As part of the build process, unused methods and assemblies are removed to reduce app download size and load times.

Publish locations:

- Blazor WebAssembly
  - Standalone: The app is published into the `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` folder. To deploy the app as a static site, copy the contents of the `wwwroot` folder to the static site host.
  - Hosted: The client Blazor WebAssembly app is published into the `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` folder of the server app, along with any other static web assets of the server app. Deploy the contents of the `publish` folder to the host.
- Blazor Server: The app is published into the `/bin/Release/{TARGET FRAMEWORK}/publish` folder. Deploy the contents of the `publish` folder to the host.

The assets in the folder are deployed to the web server. Deployment might be a manual or automated process depending on the development tools in use.

App base path

The *app base path* is the app's root URL path. Consider the following ASP.NET Core app and Blazor sub-app:

- The ASP.NET Core app is named `MyApp`:
  - The app physically resides at `d:/MyApp`.
  - Requests are received at `https://www.contoso.com/{MYAPP RESOURCE}`.
- A Blazor app named `CoolApp` is a sub-app of `MyApp`:
  - The sub-app physically resides at `d:/MyApp/CoolApp`.
  - Requests are received at `https://www.contoso.com/CoolApp/{COOLAPP RESOURCE}`.

Without specifying additional configuration for `CoolApp`, the sub-app in this scenario has no knowledge of where it resides on the server. For example, the app can't construct correct relative URLs to its resources without knowing that it resides at the relative URL path `/CoolApp/`.

To provide configuration for the Blazor app's base path of `https://www.contoso.com/CoolApp/`, the `<base>` tag's `href` attribute is set to the relative root path in the `Pages/_Host.cshtml` file (Blazor Server) or `wwwroot/index.html` file (Blazor WebAssembly):

```
<base href="/CoolApp/">
```

Blazor Server apps additionally set the server-side base path by calling UsePathBase in the app's request pipeline of `Startup.Configure`:

```
app.UsePathBase("/CoolApp");
```

By providing the relative URL path, a component that isn't in the root directory can construct URLs relative to the app's root path. Components at different levels of the directory structure can build links to other resources at locations throughout the app. The app base path is also used to intercept selected hyperlinks where the `href` target of the link is within the app base path URI space. The Blazor router handles the internal navigation.

In many hosting scenarios, the relative URL path to the app is the root of the app. In these cases, the app's relative URL base path is a forward slash (`<base href="/" />`), which is the default configuration for a Blazor app. In other hosting scenarios, such as GitHub Pages and IIS sub-apps, the app base path must be set to the server's relative URL path of the app.

To set the app's base path, update the `<base>` tag within the `<head>` tag elements of the `Pages/_Host.cshtml` file (Blazor Server) or `wwwroot/index.html` file (Blazor WebAssembly). Set the `href` attribute value to `/{RELATIVE URL PATH}/` (the trailing slash is required), where `{RELATIVE URL PATH}` is the app's full relative URL path.

For an Blazor WebAssembly app with a non-root relative URL path (for example, `<base href="/CoolApp/">`), the app fails to find its resources *when run locally*. To overcome this problem during local development and testing, you can supply a *path base* argument that matches the `href` value of the `<base>` tag at runtime. Don't include a trailing slash. To pass the path base argument when running the app locally, execute the `dotnet run` command from the app's directory with the `--pathbase` option:

```
dotnet run --pathbase=/{RELATIVE URL PATH (no trailing slash)}
```

For a Blazor WebAssembly app with a relative URL path of `/CoolApp/` (`<base href="/CoolApp/">`), the command is:

```
dotnet run --pathbase=/CoolApp
```

The Blazor WebAssembly app responds locally at `http://localhost:port/CoolApp`.

Blazor Server `MapFallbackToPage` configuration

Pass the following path to MapFallbackToPage in `Startup.Configure`:

```
endpoints.MapFallbackToPage("/{RELATIVE PATH}/{**path:nonfile}");
```

The placeholder `{RELATIVE PATH}` is the non-root path on the server. For example, `CoolApp` is the placeholder segment if the non-root URL to the app is `https://{HOST}:{PORT}/CoolApp/`):

```
endpoints.MapFallbackToPage("/CoolApp/{**path:nonfile}");
```

Host multiple Blazor WebAssembly apps

For more information on hosting multiple Blazor WebAssembly apps in a hosted Blazor solution, see Host and deploy ASP.NET Core Blazor WebAssembly.

Deployment

For deployment guidance, see the following topics:

- Host and deploy ASP.NET Core Blazor WebAssembly
- Host and deploy ASP.NET Core Blazor Server

Host and deploy ASP.NET Core Blazor WebAssembly

9/22/2020 • 21 minutes to read • Edit Online

By Luke Latham, Rainer Stropek, Daniel Roth, Ben Adams, and Safia Abdalla

With the Blazor WebAssembly hosting model:

- The Blazor app, its dependencies, and the .NET runtime are downloaded to the browser in parallel.
- The app is executed directly on the browser UI thread.

The following deployment strategies are supported:

- The Blazor app is served by an ASP.NET Core app. This strategy is covered in the Hosted deployment with ASP.NET Core section.
- The Blazor app is placed on a static hosting web server or service, where .NET isn't used to serve the Blazor app. This strategy is covered in the Standalone deployment section, which includes information on hosting a Blazor WebAssembly app as an IIS sub-app.

Compression

When a Blazor WebAssembly app is published, the output is statically compressed during publish to reduce the app's size and remove the overhead for runtime compression. The following compression algorithms are used:

- Brotli (highest level)
- Gzip

Blazor relies on the host to the serve the appropriate compressed files. When using an ASP.NET Core hosted project, the host project is capable of performing content negotiation and serving the statically-compressed files. When hosting a Blazor WebAssembly standalone app, additional work might be required to ensure that statically-compressed files are served:

- For IIS `web.config` compression configuration, see the IIS: Brotli and Gzip compression section.
- When hosting on static hosting solutions that don't support statically-compressed file content negotiation, such as GitHub Pages, consider configuring the app to fetch and decode Brotli compressed files:
  - Obtain the JavaScript Brotli decoder from the google/brotli GitHub repository. As of July 2020, the decoder file is named `decode.min.js` and found in the repository's `js` folder.
  - Update the app to use the decoder. Change the markup inside the closing `<body>` tag in `wwwroot/index.html` to the following:

```
<script src="decode.min.js"></script>
<script src="_framework/blazor.webassembly.js" autostart="false"></script>
<script>
  Blazor.start({
    loadBootResource: function (type, name, defaultUri, integrity) {
      if (type !== 'dotnetjs' && location.hostname !== 'localhost') {
        return (async function () {
          const response = await fetch(defaultUri + '.br', { cache: 'no-cache' });
          if (!response.ok) {
            throw new Error(response.statusText);
          }
          const originalResponseBuffer = await response.arrayBuffer();
          const originalResponseArray = new Int8Array(originalResponseBuffer);
          const decompressedResponseArray = BrotliDecode(originalResponseArray);
          const contentType = type ===
            'dotnetwasm' ? 'application/wasm' : 'application/octet-stream';
          return new Response(decompressedResponseArray,
            { headers: { 'content-type': contentType } });
        })();
      }
    }
  });
</script>
```

To disable compression, add the `BlazorEnableCompression` MSBuild property to the app's project file and set the value to `false`:

```
<PropertyGroup>
  <BlazorEnableCompression>false</BlazorEnableCompression>
</PropertyGroup>
```

The `BlazorEnableCompression` property can be passed to the `dotnet publish` command with the following syntax in a command shell:

```
dotnet publish -p:BlazorEnableCompression=false
```

Rewrite URLs for correct routing

Routing requests for page components in a Blazor WebAssembly app isn't as straightforward as routing requests in a Blazor Server, hosted app. Consider a Blazor WebAssembly app with two components:

- `Main.razor`: Loads at the root of the app and contains a link to the `About` component (`href="About"`).
- `About.razor`: `About` component.

When the app's default document is requested using the browser's address bar (for example, `https://www.contoso.com/`):

1. The browser makes a request.
2. The default page is returned, which is usually `index.html`.
3. `index.html` bootstraps the app.
4. Blazor's router loads, and the Razor `Main` component is rendered.

In the Main page, selecting the link to the `About` component works on the client because the Blazor router stops the browser from making a request on the Internet to `www.contoso.com` for `About` and serves the rendered `About` component itself. All of the requests for internal endpoints *within the Blazor WebAssembly app* work the same way: Requests don't trigger browser-based requests to server-hosted resources on the Internet. The router handles the

requests internally.

If a request is made using the browser's address bar for `www.contoso.com/About`, the request fails. No such resource exists on the app's Internet host, so a *404 - Not Found* response is returned.

Because browsers make requests to Internet-based hosts for client-side pages, web servers and hosting services must rewrite all requests for resources not physically on the server to the `index.html` page. When `index.html` is returned, the app's Blazor router takes over and responds with the correct resource.

When deploying to an IIS server, you can use the URL Rewrite Module with the app's published `web.config` file. For more information, see the IIS section.

Hosted deployment with ASP.NET Core

A *hosted deployment* serves the Blazor WebAssembly app to browsers from an ASP.NET Core app that runs on a web server.

The client Blazor WebAssembly app is published into the `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` folder of the server app, along with any other static web assets of the server app. The two apps are deployed together. A web server that is capable of hosting an ASP.NET Core app is required. For a hosted deployment, Visual Studio includes the **Blazor WebAssembly App** project template (`blazorwasm` template when using the `dotnet new` command) with the `Hosted` option selected (`-ho|--hosted` when using the `dotnet new` command).

For more information on ASP.NET Core app hosting and deployment, see Host and deploy ASP.NET Core.

For information on deploying to Azure App Service, see Publish an ASP.NET Core app to Azure with Visual Studio.

Hosted deployment with multiple Blazor WebAssembly apps

App configuration

To configure a hosted Blazor solution to serve multiple Blazor WebAssembly apps:

- Use an existing hosted Blazor solution or create a new solution from the Blazor Hosted project template.
- In the client app's project file, add a `<StaticWebAssetBasePath>` property to the `<PropertyGroup>` with a value of `FirstApp` to set the base path for the project's static assets:

  ```
  <PropertyGroup>
    ...
    <StaticWebAssetBasePath>FirstApp</StaticWebAssetBasePath>
  </PropertyGroup>
  ```

- Add a second client app to the solution:
  - Add a folder named `SecondClient` to the solution's folder.
  - Create a Blazor WebAssembly app named `SecondBlazorApp.Client` in the `SecondClient` folder from the Blazor WebAssembly project template.
  - In the app's project file:
    - Add a `<StaticWebAssetBasePath>` property to the `<PropertyGroup>` with a value of `SecondApp`:

      ```
      <PropertyGroup>
        ...
        <StaticWebAssetBasePath>SecondApp</StaticWebAssetBasePath>
      </PropertyGroup>
      ```

- Add a project reference to the `Shared` project:

  ```
  <ItemGroup>
    <ProjectReference Include="..\Shared\{SOLUTION NAME}.Shared.csproj" />
  </ItemGroup>
  ```

  The placeholder `{SOLUTION NAME}` is the solution's name.

- In the server app's project file, create a project reference for the added client app:

  ```
  <ItemGroup>
    ...
    <ProjectReference Include="..\SecondClient\SecondBlazorApp.Client.csproj" />
  </ItemGroup>
  ```

- In the server app's `Properties/launchSettings.json` file, configure the `applicationUrl` of the Kestrel profile (`{SOLUTION NAME}.Server`) to access the client apps at ports 5001 and 5002:

  ```
  "applicationUrl": "https://localhost:5001;https://localhost:5002",
  ```

- In the server app's `Startup.Configure` method (`Startup.cs`), remove the following lines, which appear after the call to UseHttpsRedirection:

  ```
  app.UseBlazorFrameworkFiles();
  app.UseStaticFiles();

  app.UseRouting();

  app.UseEndpoints(endpoints =>
  {
      endpoints.MapRazorPages();
      endpoints.MapControllers();
      endpoints.MapFallbackToFile("index.html");
  });
  ```

  Add middleware that maps requests to the client apps. The following example configures the middleware to run when:

  - The request port is either 5001 for the original client app or 5002 for the added client app.
  - The request host is either `firstapp.com` for the original client app or `secondapp.com` for the added client app.

  > **NOTE**
  >
  > The example shown in this section requires additional configuration for:
  >
  > - Accessing the apps at the example host domains, `firstapp.com` and `secondapp.com`.
  > - Certificates for the client apps to enable TLS security (HTTPS).
  >
  > The required configuration is beyond the scope of this article and depends on how the solution is hosted. For more information see the Host and deploy articles.

  Place the following code where the lines were removed earlier:

```
app.MapWhen(ctx => ctx.Request.Host.Port == 5001 ||
    ctx.Request.Host.Equals("firstapp.com"), first =>
{
    first.Use((ctx, nxt) =>
    {
        ctx.Request.Path = "/FirstApp" + ctx.Request.Path;
        return nxt();
    });

    first.UseBlazorFrameworkFiles("/FirstApp");
    first.UseStaticFiles();
    first.UseStaticFiles("/FirstApp");
    first.UseRouting();

    first.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapFallbackToFile("/FirstApp/{*path:nonfile}",
            "FirstApp/index.html");
    });
});

app.MapWhen(ctx => ctx.Request.Host.Port == 5002 ||
    ctx.Request.Host.Equals("secondapp.com"), second =>
{
    second.Use((ctx, nxt) =>
    {
        ctx.Request.Path = "/SecondApp" + ctx.Request.Path;
        return nxt();
    });

    second.UseBlazorFrameworkFiles("/SecondApp");
    second.UseStaticFiles();
    second.UseStaticFiles("/SecondApp");
    second.UseRouting();

    second.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapFallbackToFile("/SecondApp/{*path:nonfile}",
            "SecondApp/index.html");
    });
});
```

- In the server app's weather forecast controller (`Controllers/WeatherForecastController.cs`), replace the existing route (`[Route("[controller]")]`) to `WeatherForecastController` with the following routes:

  ```
  [Route("FirstApp/[controller]")]
  [Route("SecondApp/[controller]")]
  ```

  The middleware added to the server app's `Startup.Configure` method earlier modifies incoming requests to `/WeatherForecast` to either `/FirstApp/WeatherForecast` or `/SecondApp/WeatherForecast` depending on the port (5001/5002) or domain (`firstapp.com` / `secondapp.com`). The preceding controller routes are required in order to return weather data from the server app to the client apps.

Static assets and class libraries

Use the following approaches for static assets:

- When the asset is in the client app's `wwwroot` folder, provide their paths normally:

```
<img alt="..." src="/{ASSET FILE NAME}" />
```

- When the asset is in the `wwwroot` folder of a Razor Class Library (RCL), reference the static asset in the client app per the guidance in the RCL article:

  ```
  <img alt="..." src="_content/{LIBRARY NAME}/{ASSET FILE NAME}" />
  ```

Components provided to a client app by a class library are referenced normally. If any components require stylesheets or JavaScript files, use either of the following approaches to obtain the static assets:

- The client app's `wwwroot/index.html` file can link (`<link>`) to the static assets.
- The component can use the framework's `Link` component to obtain the static assets.

The preceding approaches are demonstrated in the following examples.

Components provided to a client app by a class library are referenced normally. If any components require stylesheets or JavaScript files, the client app's `wwwroot/index.html` file must include the correct static asset links. These approaches are demonstrated in the following examples.

Add the following `Jeep` component to one of the client apps. The `Jeep` component uses:

- An image from the client app's `wwwroot` folder (`jeep-cj.png`).
- An image from an added Razor component library (`JeepImage`) `wwwroot` folder (`jeep-yj.png`).
- The example component (`Component1`) is created automatically by the RCL project template when the `JeepImage` library is added to the solution.

```
@page "/Jeep"

<h1>1979 Jeep CJ-5&trade;</h1>

<p>
    <img alt="1979 Jeep CJ-5&trade;" src="/jeep-cj.png" />
</p>

<h1>1991 Jeep YJ&trade;</h1>

<p>
    <img alt="1991 Jeep YJ&trade;" src="_content/JeepImage/jeep-yj.png" />
</p>

<p>
    <em>Jeep CJ-5</em> and <em>Jeep YJ</em> are a trademarks of
    <a href="https://www.fcagroup.com">Fiat Chrysler Automobiles</a>.
</p>

<JeepImage.Component1 />
```

WARNING

Do **not** publish images of vehicles publicly unless you own the images. Otherwise, you risk copyright infringement.

The library's `jeep-yj.png` image can also be added to the library's `Component1` component (`Component1.razor`). To provide the `my-component` CSS class to the client app's page, link to the library's stylesheet using the framework's `Link` component:

```
<div class="my-component">
    <Link href="_content/JeepImage/styles.css" rel="stylesheet" />

    <h1>JeepImage.Component1</h1>

    <p>
        This Blazor component is defined in the <strong>JeepImage</strong> package.
    </p>

    <p>
        <img alt="1991 Jeep YJ&trade;" src="_content/JeepImage/jeep-yj.png" />
    </p>
</div>
```

An alternative to using the `Link` component is to load the stylesheet from the client app's `wwwroot/index.html` file. This approach makes the stylesheet available to all of the components in the client app:

```
<head>
    ...
    <link href="_content/JeepImage/styles.css" rel="stylesheet" />
</head>
```

The library's `jeep-yj.png` image can also be added to the library's `Component1` component (`Component1.razor`):

```
<div class="my-component">
    <h1>JeepImage.Component1</h1>

    <p>
        This Blazor component is defined in the <strong>JeepImage</strong> package.
    </p>

    <p>
        <img alt="1991 Jeep YJ&trade;" src="_content/JeepImage/jeep-yj.png" />
    </p>
</div>
```

The client app's `wwwroot/index.html` file requests the library's stylesheet with the following added `<link>` tag:

```
<head>
    ...
    <link href="_content/JeepImage/styles.css" rel="stylesheet" />
</head>
```

Add navigation to the `Jeep` component in the client app's `NavMenu` component (`Shared/NavMenu.razor`):

```
<li class="nav-item px-3">
    <NavLink class="nav-link" href="Jeep">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Jeep
    </NavLink>
</li>
```

For more information on RCLs, see:

- ASP.NET Core Razor components class libraries
- Reusable Razor UI in class libraries with ASP.NET Core

Standalone deployment

A *standalone deployment* serves the Blazor WebAssembly app as a set of static files that are requested directly by clients. Any static file server is able to serve the Blazor app.

Standalone deployment assets are published into the `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` folder.

Azure App Service

Blazor WebAssembly apps can be deployed to Azure App Services on Windows, which hosts the app on IIS.

Deploying a standalone Blazor WebAssembly app to Azure App Service for Linux isn't currently supported. A Linux server image to host the app isn't available at this time. Work is in progress to enable this scenario.

IIS

IIS is a capable static file server for Blazor apps. To configure IIS to host Blazor, see Build a Static Website on IIS.

Published assets are created in the `/bin/Release/{TARGET FRAMEWORK}/publish` folder. Host the contents of the `publish` folder on the web server or hosting service.

web.config

When a Blazor project is published, a `web.config` file is created with the following IIS configuration:

- MIME types are set for the following file extensions:
  - `.dll` : `application/octet-stream`
  - `.json` : `application/json`
  - `.wasm` : `application/wasm`
  - `.woff` : `application/font-woff`
  - `.woff2` : `application/font-woff`
- HTTP compression is enabled for the following MIME types:
  - `application/octet-stream`
  - `application/wasm`
- URL Rewrite Module rules are established:
  - Serve the sub-directory where the app's static assets reside (`wwwroot/{PATH REQUESTED}`).
  - Create SPA fallback routing so that requests for non-file assets are redirected to the app's default document in its static assets folder (`wwwroot/index.html`).

Use a custom web.config

To use a custom `web.config` file, place the custom `web.config` file at the root of the project folder and publish the project.

Install the URL Rewrite Module

The URL Rewrite Module is required to rewrite URLs. The module isn't installed by default, and it isn't available for install as a Web Server (IIS) role service feature. The module must be downloaded from the IIS website. Use the Web Platform Installer to install the module:

1. Locally, navigate to the URL Rewrite Module downloads page. For the English version, select **WebPI** to download the WebPI installer. For other languages, select the appropriate architecture for the server (x86/x64) to download the installer.
2. Copy the installer to the server. Run the installer. Select the **Install** button and accept the license terms. A server restart isn't required after the install completes.

Configure the website

Set the website's **Physical path** to the app's folder. The folder contains:

- The `web.config` file that IIS uses to configure the website, including the required redirect rules and file content types.
- The app's static asset folder.

Host as an IIS sub-app

If a standalone app is hosted as an IIS sub-app, perform either of the following:

- Disable the inherited ASP.NET Core Module handler.

  Remove the handler in the Blazor app's published `web.config` file by adding a `<handlers>` section to the file:

  ```
  <handlers>
    <remove name="aspNetCore" />
  </handlers>
  ```

- Disable inheritance of the root (parent) app's `<system.webServer>` section using a `<location>` element with `inheritInChildApplications` set to `false`:

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <location path="." inheritInChildApplications="false">
      <system.webServer>
        <handlers>
          <add name="aspNetCore" ... />
        </handlers>
        <aspNetCore ... />
      </system.webServer>
    </location>
  </configuration>
  ```

Removing the handler or disabling inheritance is performed in addition to configuring the app's base path. Set the app base path in the app's `index.html` file to the IIS alias used when configuring the sub-app in IIS.

Brotli and Gzip compression

IIS can be configured via `web.config` to serve Brotli or Gzip compressed Blazor assets. For an example configuration, see `web.config`.

Troubleshooting

If a *500 - Internal Server Error* is received and IIS Manager throws errors when attempting to access the website's configuration, confirm that the URL Rewrite Module is installed. When the module isn't installed, the `web.config` file can't be parsed by IIS. This prevents the IIS Manager from loading the website's configuration and the website from serving Blazor's static files.

For more information on troubleshooting deployments to IIS, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Azure Storage

Azure Storage static file hosting allows serverless Blazor app hosting. Custom domain names, the Azure Content Delivery Network (CDN), and HTTPS are supported.

When the blob service is enabled for static website hosting on a storage account:

- Set the **Index document name** to `index.html`.
- Set the **Error document path** to `index.html`. Razor components and other non-file endpoints don't reside at physical paths in the static content stored by the blob service. When a request for one of these resources is received that the Blazor router should handle, the *404 - Not Found* error generated by the blob service routes the request to the **Error document path**. The `index.html` blob is returned, and the Blazor router loads and processes the path.

If files aren't loaded at runtime due to inappropriate MIME types in the files' `Content-Type` headers, take either of the following actions:

- Configure your tooling to set the correct MIME types (`Content-Type` headers) when the files are deployed.
- Change the MIME types (`Content-Type` headers) for the files after the app is deployed.

  In Storage Explorer (Azure portal) for each file:

  1. Right-click the file and select **Properties**.
  2. Set the **ContentType** and select the **Save** button.

For more information, see Static website hosting in Azure Storage.

Nginx

The following `nginx.conf` file is simplified to show how to configure Nginx to send the `index.html` file whenever it can't find a corresponding file on disk.

```
events { }
http {
    server {
        listen 80;

        location / {
            root      /usr/share/nginx/html;
            try_files $uri $uri/ /index.html =404;
        }
    }
}
```

When setting the NGINX burst rate limit with `limit_req`, Blazor WebAssembly apps may require a large `burst` parameter value to accommodate the relatively large number of requests made by an app. Initially, set the value to at least 60:

```
http {
    server {
        ...

        location / {
            ...

            limit_req zone=one burst=60 nodelay;
        }
    }
}
```

Increase the value if browser developer tools or a network traffic tool indicates that requests are receiving a *503 - Service Unavailable* status code.

For more information on production Nginx web server configuration, see Creating NGINX Plus and NGINX Configuration Files.

Nginx in Docker

To host Blazor in Docker using Nginx, setup the Dockerfile to use the Alpine-based Nginx image. Update the Dockerfile to copy the `nginx.config` file into the container.

Add one line to the Dockerfile, as shown in the following example:

```
FROM nginx:alpine
COPY ./bin/Release/netstandard2.0/publish /usr/share/nginx/html/
COPY nginx.conf /etc/nginx/nginx.conf
```

Apache

To deploy a Blazor WebAssembly app to CentOS 7 or later:

1. Create the Apache configuration file. The following example is a simplified configuration file (`blazorapp.config`):

```
<VirtualHost *:80>
    ServerName www.example.com
    ServerAlias *.example.com

    DocumentRoot "/var/www/blazorapp"
    ErrorDocument 404 /index.html

    AddType application/wasm .wasm
    AddType application/octet-stream .dll

    <Directory "/var/www/blazorapp">
        Options -Indexes
        AllowOverride None
    </Directory>

    <IfModule mod_deflate.c>
        AddOutputFilterByType DEFLATE text/css
        AddOutputFilterByType DEFLATE application/javascript
        AddOutputFilterByType DEFLATE text/html
        AddOutputFilterByType DEFLATE application/octet-stream
        AddOutputFilterByType DEFLATE application/wasm
        <IfModule mod_setenvif.c>
        BrowserMatch ^Mozilla/4 gzip-only-text/html
        BrowserMatch ^Mozilla/4.0[678] no-gzip
        BrowserMatch bMSIE !no-gzip !gzip-only-text/html
    </IfModule>
    </IfModule>

    ErrorLog /var/log/httpd/blazorapp-error.log
    CustomLog /var/log/httpd/blazorapp-access.log common
</VirtualHost>
```

2. Place the Apache configuration file into the `/etc/httpd/conf.d/` directory, which is the default Apache configuration directory in CentOS 7.
3. Place the app's files into the `/var/www/blazorapp` directory (the location specified to `DocumentRoot` in the configuration file).
4. Restart the Apache service.

For more information, see `mod_mime` and `mod_deflate` .

GitHub Pages

To handle URL rewrites, add a `wwwroot/404.html` file with a script that handles redirecting the request to the `index.html` page. For an example, see the SteveSandersonMS/BlazorOnGitHubPages GitHub repository:

- `wwwroot/404.html`
- Live site)

When using a project site instead of an organization site, update the `<base>` tag in `wwwroot/index.html` . Set the `href` attribute value to the GitHub repository name with a trailing slash (for example, `/my-repository/`). In the SteveSandersonMS/BlazorOnGitHubPages GitHub repository, the base `href` is updated at publish by the `.github/workflows/main.yml` configuration file.

> **NOTE**
> The SteveSandersonMS/BlazorOnGitHubPages GitHub repository isn't owned, maintained, or supported by the .NET Foundation or Microsoft.

Host configuration values

Blazor WebAssembly apps can accept the following host configuration values as command-line arguments at runtime in the development environment.

Content root

The `--contentroot` argument sets the absolute path to the directory that contains the app's content files (content root). In the following examples, `/content-root-path` is the app's content root path.

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

```
dotnet run --contentroot=/content-root-path
```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when the app is run with the Visual Studio Debugger and from a command prompt with `dotnet run`.

```
"commandLineArgs": "--contentroot=/content-root-path"
```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

```
--contentroot=/content-root-path
```

Path base

The `--pathbase` argument sets the app base path for an app run locally with a non-root relative URL path (the `<base>` tag `href` is set to a path other than `/` for staging and production). In the following examples, `/relative-URL-path` is the app's path base. For more information, see App base path.

> **IMPORTANT**
> Unlike the path provided to `href` of the `<base>` tag, don't include a trailing slash (`/`) when passing the `--pathbase` argument value. If the app base path is provided in the `<base>` tag as `<base href="/CoolApp/">` (includes a trailing slash), pass the command-line argument value as `--pathbase=/CoolApp` (no trailing slash).

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

```
dotnet run --pathbase=/relative-URL-path
```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when running the app with the Visual Studio Debugger and from a command prompt with `dotnet run`.

```
"commandLineArgs": "--pathbase=/relative-URL-path"
```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the

argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

```
--pathbase=/relative-URL-path
```

URLs

The `--urls` argument sets the IP addresses or host addresses with ports and protocols to listen on for requests.

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

  ```
  dotnet run --urls=http://127.0.0.1:0
  ```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when running the app with the Visual Studio Debugger and from a command prompt with `dotnet run`.

  ```
  "commandLineArgs": "--urls=http://127.0.0.1:0"
  ```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

  ```
  --urls=http://127.0.0.1:0
  ```

Configure the Trimmer

Blazor performs Intermediate Language (IL) trimming on each Release build to remove unnecessary IL from the output assemblies. For more information, see Configure the Trimmer for ASP.NET Core Blazor.

Configure the Linker

Blazor performs Intermediate Language (IL) linking on each Release build to remove unnecessary IL from the output assemblies. For more information, see Configure the Linker for ASP.NET Core Blazor.

Custom boot resource loading

A Blazor WebAssembly app can be initialized with the `loadBootResource` function to override the built-in boot resource loading mechanism. Use `loadBootResource` for the following scenarios:

- Allow users to load static resources, such as timezone data or `dotnet.wasm` from a CDN.
- Load compressed assemblies using an HTTP request and decompress them on the client for hosts that don't support fetching compressed contents from the server.
- Alias resources to a different name by redirecting each `fetch` request to a new name.

`loadBootResource` parameters appear in the following table.

PARAMETER	DESCRIPTION
`type`	The type of the resource. Permissable types: `assembly`, `pdb`, `dotnetjs`, `dotnetwasm`, `timezonedata`
`name`	The name of the resource.

PARAMETER	DESCRIPTION
`defaultUri`	The relative or absolute URI of the resource.
`integrity`	The integrity string representing the expected content in the response.

`loadBootResource` returns any of the following to override the loading process:

- URI string. In the following example (`wwwroot/index.html`), the following files are served from a CDN at `https://my-awesome-cdn.com/`:
  - `dotnet.*.js`
  - `dotnet.wasm`
  - Timezone data

  ```
  ...

  <script src="_framework/blazor.webassembly.js" autostart="false"></script>
  <script>
    Blazor.start({
      loadBootResource: function (type, name, defaultUri, integrity) {
        console.log(`Loading: '${type}', '${name}', '${defaultUri}', '${integrity}'`);
        switch (type) {
          case 'dotnetjs':
          case 'dotnetwasm':
          case 'timezonedata':
            return `https://my-awesome-cdn.com/blazorwebassembly/3.2.0/${name}`;
        }
      }
    });
  </script>
  ```

- `Promise<Response>`. Pass the `integrity` parameter in a header to retain the default integrity-checking behavior.

  The following example (`wwwroot/index.html`) adds a custom HTTP header to the outbound requests and passes the `integrity` parameter through to the `fetch` call:

  ```
  <script src="_framework/blazor.webassembly.js" autostart="false"></script>
  <script>
    Blazor.start({
      loadBootResource: function (type, name, defaultUri, integrity) {
        return fetch(defaultUri, {
          cache: 'no-cache',
          integrity: integrity,
          headers: { 'MyCustomHeader': 'My custom value' }
        });
      }
    });
  </script>
  ```

- `null` / `undefined`, which results in the default loading behavior.

External sources must return the required CORS headers for browsers to allow the cross-origin resource loading. CDNs usually provide the required headers by default.

You only need to specify types for custom behaviors. Types not specified to `loadBootResource` are loaded by the framework per their default loading behaviors.

Change the filename extension of DLL files

In case you have a need to change the filename extensions of the app's published `.dll` files, follow the guidance in this section.

After publishing the app, use a shell script or DevOps build pipeline to rename `.dll` files to use a different file extension. Target the `.dll` files in the `wwwroot` directory of the app's published output (for example, `{CONTENT ROOT}/bin/Release/netstandard2.1/publish/wwwroot`).

In the following examples, `.dll` files are renamed to use the `.bin` file extension.

On Windows:

```
dir .\_framework\_bin | rename-item -NewName { $_.name -replace ".dll\b",".bin" }
((Get-Content .\_framework\blazor.boot.json -Raw) -replace '.dll"','.bin"') | Set-Content
.\_framework\blazor.boot.json
```

If service worker assets are also in use, add the following command:

```
((Get-Content .\service-worker-assets.js -Raw) -replace '.dll"','.bin"') | Set-Content .\service-worker-
assets.js
```

On Linux or macOS:

```
for f in _framework/_bin/*; do mv "$f" "`echo $f | sed -e 's/\.dll\b/.bin/g'`"; done
sed -i 's/\.dll"/.bin"/g' _framework/blazor.boot.json
```

If service worker assets are also in use, add the following command:

```
sed -i 's/\.dll"/.bin"/g' service-worker-assets.js
```

To use a different file extension than `.bin`, replace `.bin` in the preceding commands.

To address the compressed `blazor.boot.json.gz` and `blazor.boot.json.br` files, adopt either of the following approaches:

- Remove the compressed `blazor.boot.json.gz` and `blazor.boot.json.br` files. Compression is disabled with this approach.
- Recompress the updated `blazor.boot.json` file.

The preceding guidance also applies when service worker assets are in use. Remove or recompress `wwwroot/service-worker-assets.js.br` and `wwwroot/service-worker-assets.js.gz`. Otherwise, file integrity checks fail in the browser.

The following Windows example uses a PowerShell script placed at the root of the project.

`ChangeDLLExtensions.ps1:`:

```
param([string]$filepath,[string]$tfm)
dir $filepath\bin\Release\$tfm\wwwroot\_framework\_bin | rename-item -NewName { $_.name -replace
".dll\b",".bin" }
((Get-Content $filepath\bin\Release\$tfm\wwwroot\_framework\blazor.boot.json -Raw) -replace '.dll"','.bin"') |
Set-Content $filepath\bin\Release\$tfm\wwwroot\_framework\blazor.boot.json
Remove-Item $filepath\bin\Release\$tfm\wwwroot\_framework\blazor.boot.json.gz
```

If service worker assets are also in use, add the following command:

```
((Get-Content $filepath\bin\Release\$tfm\wwwroot\service-worker-assets.js -Raw) -replace '.dll"','.bin"') |
Set-Content $filepath\bin\Release\$tfm\wwwroot\service-worker-assets.js
```

In the project file, the script is run after publishing the app:

```
<Target Name="ChangeDLLFileExtensions" AfterTargets="Publish" Condition="'$(Configuration)'=='Release'">
  <Exec Command="powershell.exe -command &amp;quot;&amp;amp; { .\ChangeDLLExtensions.ps1 '$(SolutionDir)'
'$(TargetFramework)'}&amp;quot;" />
</Target>
```

> **NOTE**
>
> When renaming and lazy loading the same assemblies, see the guidance in Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

To provide feedback, visit aspnetcore/issues #5477.

Host and deploy Blazor Server

9/22/2020 • 5 minutes to read • Edit Online

By Luke Latham, Rainer Stropek, and Daniel Roth

Host configuration values

Blazor Server apps can accept Generic Host configuration values.

Deployment

Using the Blazor Server hosting model, Blazor is executed on the server from within an ASP.NET Core app. UI updates, event handling, and JavaScript calls are handled over a SignalR connection.

A web server capable of hosting an ASP.NET Core app is required. Visual Studio includes the **Blazor Server App** project template (`blazorserverside` template when using the `dotnet new` command).

Scalability

Plan a deployment to make the best use of the available infrastructure for a Blazor Server app. See the following resources to address Blazor Server app scalability:

- Fundamentals of Blazor Server apps
- Threat mitigation guidance for ASP.NET Core Blazor Server

Deployment server

When considering the scalability of a single server (scale up), the memory available to an app is likely the first resource that the app will exhaust as user demands increase. The available memory on the server affects the:

- Number of active circuits that a server can support.
- UI latency on the client.

For guidance on building secure and scalable Blazor server apps, see Threat mitigation guidance for ASP.NET Core Blazor Server.

Each circuit uses approximately 250 KB of memory for a minimal *Hello World*-style app. The size of a circuit depends on the app's code and the state maintenance requirements associated with each component. We recommend that you measure resource demands during development for your app and infrastructure, but the following baseline can be a starting point in planning your deployment target: If you expect your app to support 5,000 concurrent users, consider budgeting at least 1.3 GB of server memory to the app (or ~273 KB per user).

SignalR configuration

Blazor Server apps use ASP.NET Core SignalR to communicate with the browser. SignalR's hosting and scaling conditions apply to Blazor Server apps.

Blazor works best when using WebSockets as the SignalR transport due to lower latency, reliability, and security. Long Polling is used by SignalR when WebSockets isn't available or when the app is explicitly configured to use Long Polling. When deploying to Azure App Service, configure the app to use WebSockets in the Azure portal settings for the service. For details on configuring the app for Azure App Service, see the SignalR publishing guidelines.

Azure SignalR Service

We recommend using the Azure SignalR Service for Blazor Server apps. The service allows for scaling up a Blazor Server app to a large number of concurrent SignalR connections. In addition, the SignalR service's global reach and high-performance data centers significantly aid in reducing latency due to geography.

> **IMPORTANT**
>
> When WebSockets are disabled, Azure App Service simulates a real-time connection using HTTP long-polling. HTTP long-polling is noticeably slower than running with WebSockets enabled, which doesn't use polling to simulate a client-server connection.
>
> We recommend using WebSockets for Blazor Server apps deployed to Azure App Service. The Azure SignalR Service uses WebSockets by default. If the app doesn't use the Azure SignalR Service, see Publish an ASP.NET Core SignalR app to Azure App Service.
>
> For more information, see:
>
> - What is Azure SignalR Service?
> - Performance guide for Azure SignalR Service

To configure an app (and optionally provision) the Azure SignalR Service:

1. Enable the service to support *sticky sessions*, where clients are redirected back to the same server when prerendering. Set the `ServerStickyMode` option or configuration value to `Required`. Typically, an app creates the configuration using **one** of the following approaches:

   - `Startup.ConfigureServices`:

     ```
     services.AddSignalR().AddAzureSignalR(options =>
     {
         options.ServerStickyMode =
             Microsoft.Azure.SignalR.ServerStickyMode.Required;
     });
     ```

   - Configuration (use **one** of the following approaches):

     - `appsettings.json`:

       ```
       "Azure:SignalR:ServerStickyMode": "Required"
       ```

     - The app service's **Configuration** > **Application settings** in the Azure portal (**Name**: `Azure:SignalR:ServerStickyMode`, **Value**: `Required`).

2. Create an Azure Apps publish profile in Visual Studio for the Blazor Server app.
3. Add the **Azure SignalR Service** dependency to the profile. If the Azure subscription doesn't have a pre-existing Azure SignalR Service instance to assign to the app, select **Create a new Azure SignalR Service instance** to provision a new service instance.
4. Publish the app to Azure.

IIS

When using IIS, enable:

- WebSockets on IIS.
- Sticky sessions with Application Request Routing.

Kubernetes

Create an ingress definition with the following Kubernetes annotations for sticky sessions:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: <ingress-name>
  annotations:
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "affinity"
    nginx.ingress.kubernetes.io/session-cookie-expires: "14400"
    nginx.ingress.kubernetes.io/session-cookie-max-age: "14400"
```

Linux with Nginx

For SignalR WebSockets to function properly, confirm that the proxy's `Upgrade` and `Connection` headers are set to the following values and that `$connection_upgrade` is mapped to either:

- The Upgrade header value by default.
- `close` when the Upgrade header is missing or empty.

```
http {
    map $http_upgrade $connection_upgrade {
        default Upgrade;
        ''      close;
    }

    server {
        listen      80;
        server_name example.com *.example.com
        location / {
            proxy_pass         http://localhost:5000;
            proxy_http_version 1.1;
            proxy_set_header   Upgrade $http_upgrade;
            proxy_set_header   Connection $connection_upgrade;
            proxy_set_header   Host $host;
            proxy_cache_bypass $http_upgrade;
            proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header   X-Forwarded-Proto $scheme;
        }
    }
}
```

For more information, see the following articles:

- NGINX as a WebSocket Proxy
- WebSocket proxying
- Host ASP.NET Core on Linux with Nginx

Linux with Apache

To host a Blazor app behind Apache on Linux, configure `ProxyPass` for HTTP and WebSockets traffic.

In the following example:

- Kestrel server is running on the host machine.
- The app listens for traffic on port 5000.

```
ProxyRequests       On
ProxyPreserveHost   On
ProxyPassMatch      ^/_blazor/(.*) http://localhost:5000/_blazor/$1
ProxyPass           /_blazor ws://localhost:5000/_blazor
ProxyPass           / http://localhost:5000/
ProxyPassReverse    / http://localhost:5000/
```

Enable the following modules:

```
a2enmod   proxy
a2enmod   proxy_wstunnel
```

Check the browser console for WebSockets errors. Example errors:

- Firefox can't establish a connection to the server at ws://the-domain-name.tld/_blazor?id=XXX.
- Error: Failed to start the transport 'WebSockets': Error: There was an error with the transport.
- Error: Failed to start the transport 'LongPolling': TypeError: this.transport is undefined
- Error: Unable to connect to the server with any of the available transports. WebSockets failed
- Error: Cannot send data if the connection is not in the 'Connected' State.

For more information, see the Apache documentation.

Measure network latency

JS interop can be used to measure network latency, as the following example demonstrates:

```
@inject IJSRuntime JS

@if (latency is null)
{
    <span>Calculating...</span>
}
else
{
    <span>@(latency.Value.TotalMilliseconds)ms</span>
}

@code {
    private DateTime startTime;
    private TimeSpan? latency;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            startTime = DateTime.UtcNow;
            var _ = await JS.InvokeAsync<string>("toString");
            latency = DateTime.UtcNow - startTime;
            StateHasChanged();
        }
    }
}
```

For a reasonable UI experience, we recommend a sustained UI latency of 250ms or less.

Configure the Linker for ASP.NET Core Blazor

9/22/2020 • 2 minutes to read • Edit Online

By Luke Latham

Blazor WebAssembly performs Intermediate Language (IL) linking during a build to trim unnecessary IL from the app's output assemblies. The linker is disabled when building in Debug configuration. Apps must build in Release configuration to enable the linker. We recommend building in Release when deploying your Blazor WebAssembly apps.

Linking an app optimizes for size but may have detrimental effects. Apps that use reflection or related dynamic features may break when trimmed because the linker doesn't know about this dynamic behavior and can't determine in general which types are required for reflection at runtime. To trim such apps, the linker must be informed about any types required by reflection in the code and in packages or frameworks that the app depends on.

To ensure the trimmed app works correctly once deployed, it's important to test Release builds of the app frequently while developing.

Linking for Blazor apps can be configured using these MSBuild features:

- Configure linking globally with a MSBuild property.
- Control linking on a per-assembly basis with a configuration file.

Control linking with an MSBuild property

Linking is enabled when an app is built in `Release` configuration. To change this, configure the `BlazorWebAssemblyEnableLinking` MSBuild property in the project file:

```
<PropertyGroup>
  <BlazorWebAssemblyEnableLinking>false</BlazorWebAssemblyEnableLinking>
</PropertyGroup>
```

Control linking with a configuration file

Control linking on a per-assembly basis by providing an XML configuration file and specifying the file as a MSBuild item in the project file:

```
<ItemGroup>
  <BlazorLinkerDescriptor Include="LinkerConfig.xml" />
</ItemGroup>
```

`LinkerConfig.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
  This file specifies which parts of the BCL or Blazor packages must not be
  stripped by the IL Linker even if they aren't referenced by user code.
-->
<linker>
  <assembly fullname="mscorlib">
    <!--
      Preserve the methods in WasmRuntime because its methods are called by
      JavaScript client-side code to implement timers.
      Fixes: https://github.com/dotnet/blazor/issues/239
    -->
    <type fullname="System.Threading.WasmRuntime" />
  </assembly>
  <assembly fullname="System.Core">
    <!--
      System.Linq.Expressions* is required by Json.NET and any
      expression.Compile caller. The assembly isn't stripped.
    -->
    <type fullname="System.Linq.Expressions*" />
  </assembly>
  <!--
    In this example, the app's entry point assembly is listed. The assembly
    isn't stripped by the IL Linker.
  -->
  <assembly fullname="MyCoolBlazorApp" />
</linker>
```

For more information and examples, see Data Formats (mono/linker GitHub repository).

Add an XML linker configuration file to a library

To configure the linker for a specific library, add an XML linker configuration file into the library as an embedded resource. The embedded resource must have the same name as the assembly.

In the following example, the `LinkerConfig.xml` file is specified as an embedded resource that has the same name as the library's assembly:

```
<ItemGroup>
  <EmbeddedResource Include="LinkerConfig.xml">
    <LogicalName>$(MSBuildProjectName).xml</LogicalName>
  </EmbeddedResource>
</ItemGroup>
```

Configure the linker for internationalization

By default, Blazor's linker configuration for Blazor WebAssembly apps strips out internationalization information except for locales explicitly requested. Removing these assemblies minimizes the app's size.

To control which I18N assemblies are retained, set the `<BlazorWebAssemblyI18NAssemblies>` MSBuild property in the project file:

```
<PropertyGroup>
  <BlazorWebAssemblyI18NAssemblies>{all|none|REGION1,REGION2,...}</BlazorWebAssemblyI18NAssemblies>
</PropertyGroup>
```

REGION VALUE	MONO REGION ASSEMBLY
`all`	All assemblies included

REGION VALUE	MONO REGION ASSEMBLY
`cjk`	`I18N.CJK.dll`
`mideast`	`I18N.MidEast.dll`
`none` (default)	None
`other`	`I18N.Other.dll`
`rare`	`I18N.Rare.dll`
`west`	`I18N.West.dll`

Use a comma to separate multiple values (for example, `mideast,west`).

For more information, see I18N: Pnetlib Internationalization Framework Library (mono/mono GitHub repository).

Additional resources

- ASP.NET Core Blazor WebAssembly performance best practices

Configure the Trimmer for ASP.NET Core Blazor

9/22/2020 • 2 minutes to read • Edit Online

By Pranav Krishnamoorthy

Blazor WebAssembly performs Intermediate Language (IL) trimming to reduce the size of the published output.

Trimming an app optimizes for size but may have detrimental effects. Apps that use reflection or related dynamic features may break when trimmed because the trimmer doesn't know about dynamic behavior and can't determine in general which types are required for reflection at runtime. To trim such apps, the trimmer must be informed about any types required by reflection in the code and in packages or frameworks that the app depends on.

To ensure the trimmed app works correctly once deployed, it's important to test published output frequently while developing.

Trimming for .NET apps can be disabled by setting the `PublishTrimmed` MSBuild property to `false` in the app's project file:

```
<PropertyGroup>
  <PublishTrimmed>false</PublishTrimmed>
</PropertyGroup>
```

Additional options to configure the trimmer can be found at Trimming options.

Additional resources

- Trim self-contained deployments and executables
- ASP.NET Core Blazor WebAssembly performance best practices

ASP.NET Core Blazor Server with Entity Framework Core (EFCore)

9/22/2020 • 8 minutes to read • Edit Online

By: Jeremy Likness

Blazor Server is a stateful app framework. The app maintains an ongoing connection to the server, and the user's state is held in the server's memory in a *circuit*. One example of user state is data held in dependency injection (DI) service instances that are scoped to the circuit. The unique application model that Blazor Server provides requires a special approach to use Entity Framework Core.

> **NOTE**
> This article addresses EF Core in Blazor Server apps. Blazor WebAssembly apps run in a WebAssembly sandbox that prevents most direct database connections. Running EF Core in Blazor WebAssembly is beyond the scope of this article.

Sample app

The sample app was built as a reference for Blazor Server apps that use EF Core. The sample app includes a grid with sorting and filtering, delete, add, and update operations. The sample demonstrates use of EF Core to handle optimistic concurrency.

View or download sample code (how to download)

The sample uses a local SQLite database so that it can be used on any platform. The sample also configures database logging to show the SQL queries that are generated. This is configured in `appsettings.Development.json`:

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  }
}
```

The grid, add, and view components use the "context-per-operation" pattern, where a context is created for each operation. The edit component uses the "context-per-component" pattern, where a context is created for each component.

> **NOTE**
> Some of the code examples in this topic require namespaces and services that aren't shown. To inspect the fully working code, including the required `@using` and `@inject` directives for Razor examples, see the sample app.

Database access

EF Core relies on a DbContext as the means to configure database access and act as a *unit of work*. EF Core provides the AddDbContext extension for ASP.NET Core apps that registers the context as a *scoped* service by default. In Blazor Server apps, scoped service registrations can be problematic because the instance is shared across components within the user's circuit. DbContext isn't thread safe and isn't designed for concurrent use. The existing lifetimes are inappropriate for these reasons:

- **Singleton** shares state across all users of the app and leads to inappropriate concurrent use.
- **Scoped** (the default) poses a similar issue between components for the same user.
- **Transient** results in a new instance per request; but as components can be long-lived, this results in a longer-lived context than may be intended.

The following recommendations are designed to provide a consistent approach to using EF Core in Blazor Server apps.

- By default, consider using one context per operation. The context is designed for fast, low overhead instantiation:

  ```
  var using context = new MyContext();

  return await context.MyEntities.ToListAsync();
  ```

- Use a flag to prevent multiple concurrent operations:

  ```
  if (Loading)
  {
      return;
  }

  try
  {
      Loading = true;

      ...
  }
  finally
  {
      Loading = false;
  }
  ```

  Place operations after the `Loading = true;` line in the `try` block.

- For longer-lived operations that take advantage of EF Core's change tracking or concurrency control, scope the context to the lifetime of the component.

New DbContext instances

The fastest way to create a new DbContext instance is by using `new` to create a new instance. However, there are several scenarios that may require resolving additional dependencies. For example, you may wish to use `DbContextOptions` to configure the context.

The recommended solution to create a new DbContext with dependencies is to use a factory. EF Core 5.0 or later provides a built-in factory for creating new contexts.

The following example configures SQLite and enables data logging. The code uses an extension method to configure the database factory for DI and provide default options:

```
services.AddDbContextFactory<ContactContext>(opt =>
    opt.UseSqlite($"Data Source={nameof(ContactContext.ContactsDb)}.db")
    .EnableSensitiveDataLogging());
```

The factory is injected into components and used to create new instances. For example, in `Pages/Index.razor`:

```
private async Task DeleteContactAsync()
{
    using var context = DbFactory.CreateDbContext();

    Filters.Loading = true;

    var contact = await context.Contacts.FirstAsync(
        c => c.Id == Wrapper.DeleteRequestId);

    if (contact != null)
    {
        context.Contacts.Remove(contact);
        await context.SaveChangesAsync();
    }

    Filters.Loading = false;

    await ReloadAsync();
}
```

NOTE

`Wrapper` is a component reference to the `GridWrapper` component. See the `Index` component (`Pages/Index.razor`) in the sample app.

Scope to the component lifetime

You may wish to create a DbContext that exists for the lifetime of a component. This allows you to use it as a unit of work and take advantage of built-in features, such as change tracking and concurrency resolution. You can use the factory to create a context and track it for the lifetime of the component. First, implement IDisposable and inject the factory as shown in `Pages/EditContact.razor`:

```
@implements IDisposable
@inject IDbContextFactory<ContactContext> DbFactory
```

The sample app ensures the context is disposed when the component is disposed:

```
public void Dispose()
{
    Context.Dispose();
}
```

Finally, `OnInitializedAsync` is overridden to create a new context. In the sample app, `OnInitializedAsync` loads the contact in the same method:

```
protected override async Task OnInitializedAsync()
{
    Busy = true;

    try
    {
        Context = DbFactory.CreateDbContext();
        Contact = await Context.Contacts
            .SingleOrDefaultAsync(c => c.Id == ContactId);
    }
    finally
    {
        Busy = false;
    }

    await base.OnInitializedAsync();
}
```

Blazor Server is a stateful app framework. The app maintains an ongoing connection to the server, and the user's state is held in the server's memory in a *circuit*. One example of user state is data held in dependency injection (DI) service instances that are scoped to the circuit. The unique application model that Blazor Server provides requires a special approach to use Entity Framework Core.

> **NOTE**
> This article addresses EF Core in Blazor Server apps. Blazor WebAssembly apps run in a WebAssembly sandbox that prevents most direct database connections. Running EF Core in Blazor WebAssembly is beyond the scope of this article.

Sample app

The sample app was built as a reference for Blazor Server apps that use EF Core. The sample app includes a grid with sorting and filtering, delete, add, and update operations. The sample demonstrates use of EF Core to handle optimistic concurrency.

View or download sample code (how to download)

The sample uses a local SQLite database so that it can be used on any platform. The sample also configures database logging to show the SQL queries that are generated. This is configured in `appsettings.Development.json`:

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  }
}
```

The grid, add, and view components use the "context-per-operation" pattern, where a context is created for each operation. The edit component uses the "context-per-component" pattern, where a context is created for each component.

> **NOTE**
>
> Some of the code examples in this topic require namespaces and services that aren't shown. To inspect the fully working code, including the required `@using` and `@inject` directives for Razor examples, see the sample app.

Database access

EF Core relies on a DbContext as the means to configure database access and act as a *unit of work*. EF Core provides the AddDbContext extension for ASP.NET Core apps that registers the context as a *scoped* service by default. In Blazor Server apps, this can be problematic because the instance is shared across components within the user's circuit. DbContext isn't thread safe and isn't designed for concurrent use. The existing lifetimes are inappropriate for these reasons:

- **Singleton** shares state across all users of the app and leads to inappropriate concurrent use.
- **Scoped** (the default) poses a similar issue between components for the same user.
- **Transient** results in a new instance per request; but as components can be long-lived, this results in a longer-lived context than may be intended.

The following recommendations are designed to provide a consistent approach to using EF Core in Blazor Server apps.

- By default, consider using one context per operation. The context is designed for fast, low overhead instantiation:

  ```
  var using context = new MyContext();

  return await context.MyEntities.ToListAsync();
  ```

- Use a flag to prevent multiple concurrent operations:

  ```
  if (Loading)
  {
      return;
  }

  try
  {
      Loading = true;

      ...
  }
  finally
  {
      Loading = false;
  }
  ```

  Place operations after the `Loading = true;` line in the `try` block.

- For longer-lived operations that take advantage of EF Core's change tracking or concurrency control, scope the context to the lifetime of the component.

New DbContext instances

The fastest way to create a new DbContext instance is by using `new` to create a new instance. However, there are several scenarios that may require resolving additional dependencies. For example, you may wish to use `DbContextOptions` to configure the context.

The recommended solution to create a new DbContext with dependencies is to use a factory. The sample app implements its own factory in `Data/DbContextFactory.cs`.

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace BlazorServerDbContextExample.Data
{
    public class DbContextFactory<TContext>
        : IDbContextFactory<TContext> where TContext : DbContext
    {
        private readonly IServiceProvider provider;

        public DbContextFactory(IServiceProvider provider)
        {
            this.provider = provider;
        }

        public TContext CreateDbContext()
        {
            if (provider == null)
            {
                throw new InvalidOperationException(
                    $"You must configure an instance of IServiceProvider");
            }

            return ActivatorUtilities.CreateInstance<TContext>(provider);
        }
    }
}
```

In the preceding factory, ActivatorUtilities.CreateInstance satisfies any dependencies via the service provider.

The following example configures SQLite and enables data logging. The code uses an extension method to configure the database factory for DI and provide default options:

```
services.AddDbContextFactory<ContactContext>(opt =>
    opt.UseSqlite($"Data Source={nameof(ContactContext.ContactsDb)}.db")
    .EnableSensitiveDataLogging());
```

The factory is injected into components and used to create new instances. For example, in `Pages/Index.razor`:

```
private async Task DeleteContactAsync()
{
    using var context = DbFactory.CreateDbContext();

    Filters.Loading = true;

    var contact = await context.Contacts.FirstAsync(
        c => c.Id == Wrapper.DeleteRequestId);

    if (contact != null)
    {
        context.Contacts.Remove(contact);
        await context.SaveChangesAsync();
    }

    Filters.Loading = false;

    await ReloadAsync();
}
```

> **NOTE**
>
> `Wrapper` is a component reference to the `GridWrapper` component. See the `Index` component (`Pages/Index.razor`) in the sample app.

Scope to the component lifetime

You may wish to create a DbContext that exists for the lifetime of a component. This allows you to use it as a unit of work and take advantage of built-in features, such as change tracking and concurrency resolution. You can use the factory to create a context and track it for the lifetime of the component. First, implement IDisposable and inject the factory as shown in `Pages/EditContact.razor`:

```
@implements IDisposable
@inject IDbContextFactory<ContactContext> DbFactory
```

The sample app ensures the context is disposed when the component is disposed:

```
public void Dispose()
{
    Context.Dispose();
}
```

Finally, `OnInitializedAsync` is overridden to create a new context. In the sample app, `OnInitializedAsync` loads the contact in the same method:

```
protected override async Task OnInitializedAsync()
{
    Busy = true;

    try
    {
        Context = DbFactory.CreateDbContext();
        Contact = await Context.Contacts
            .SingleOrDefaultAsync(c => c.Id == ContactId);
    }
    finally
    {
        Busy = false;
    }

    await base.OnInitializedAsync();
}
```

In the preceding example:

- When `Busy` is set to `true`, asynchronous operations may begin. When `Busy` is set back to `false`, asynchronous operations should be finished.
- Place additional error handling logic in a `catch` block.

Additional resources

- EF Core documentation

ASP.NET Core Blazor advanced scenarios

9/22/2020 • 10 minutes to read • Edit Online

By Luke Latham and Daniel Roth

Blazor Server circuit handler

Blazor Server allows code to define a *circuit handler*, which allows running code on changes to the state of a user's circuit. A circuit handler is implemented by deriving from `CircuitHandler` and registering the class in the app's service container. The following example of a circuit handler tracks open SignalR connections:

```
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Server.Circuits;

public class TrackingCircuitHandler : CircuitHandler
{
    private HashSet<Circuit> circuits = new HashSet<Circuit>();

    public override Task OnConnectionUpAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        circuits.Add(circuit);

        return Task.CompletedTask;
    }

    public override Task OnConnectionDownAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        circuits.Remove(circuit);

        return Task.CompletedTask;
    }

    public int ConnectedCircuits => circuits.Count;
}
```

Circuit handlers are registered using DI. Scoped instances are created per instance of a circuit. Using the `TrackingCircuitHandler` in the preceding example, a singleton service is created because the state of all circuits must be tracked:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddSingleton<CircuitHandler, TrackingCircuitHandler>();
}
```

If a custom circuit handler's methods throw an unhandled exception, the exception is fatal to the Blazor Server circuit. To tolerate exceptions in a handler's code or called methods, wrap the code in one or more `try-catch` statements with error handling and logging.

When a circuit ends because a user has disconnected and the framework is cleaning up the circuit state, the framework disposes of the circuit's DI scope. Disposing the scope disposes any circuit-scoped DI services that

implement System.IDisposable. If any DI service throws an unhandled exception during disposal, the framework logs the exception.

Manual RenderTreeBuilder logic

RenderTreeBuilder provides methods for manipulating components and elements, including building components manually in C# code.

> **NOTE**
> Use of RenderTreeBuilder to create components is an advanced scenario. A malformed component (for example, an unclosed markup tag) can result in undefined behavior.

Consider the following `PetDetails` component, which can be manually built into another component:

```
<h2>Pet Details Component</h2>

<p>@PetDetailsQuote</p>

@code
{
    [Parameter]
    public string PetDetailsQuote { get; set; }
}
```

In the following example, the loop in the `CreateComponent` method generates three `PetDetails` components. In RenderTreeBuilder methods with a sequence number, sequence numbers are source code line numbers. The Blazor difference algorithm relies on the sequence numbers corresponding to distinct lines of code, not distinct call invocations. When creating a component with RenderTreeBuilder methods, hardcode the arguments for sequence numbers. **Using a calculation or counter to generate the sequence number can lead to poor performance.** For more information, see the Sequence numbers relate to code line numbers and not execution order section.

`BuiltContent` component:

```
@page "/BuiltContent"

<h1>Build a component</h1>

@CustomRender

<button type="button" @onclick="RenderComponent">
    Create three Pet Details components
</button>

@code {
    private RenderFragment CustomRender { get; set; }

    private RenderFragment CreateComponent() => builder =>
    {
        for (var i = 0; i < 3; i++)
        {
            builder.OpenComponent(0, typeof(PetDetails));
            builder.AddAttribute(1, "PetDetailsQuote", "Someone's best friend!");
            builder.CloseComponent();
        }
    };

    private void RenderComponent()
    {
        CustomRender = CreateComponent();
    }
}
```

WARNING

The types in Microsoft.AspNetCore.Components.RenderTree allow processing of the *results* of rendering operations. These are internal details of the Blazor framework implementation. These types should be considered *unstable* and subject to change in future releases.

Sequence numbers relate to code line numbers and not execution order

Razor component files (`.razor`) are always compiled. Compilation is a potential advantage over interpreting code because the compile step can be used to inject information that improves app performance at runtime.

A key example of these improvements involves *sequence numbers*. Sequence numbers indicate to the runtime which outputs came from which distinct and ordered lines of code. The runtime uses this information to generate efficient tree diffs in linear time, which is far faster than is normally possible for a general tree diff algorithm.

Consider the following Razor component (`.razor`) file:

```
@if (someFlag)
{
    <text>First</text>
}

Second
```

The preceding code compiles to something like the following:

```
if (someFlag)
{
    builder.AddContent(0, "First");
}

builder.AddContent(1, "Second");
```

When the code executes for the first time, if `someFlag` is `true`, the builder receives:

SEQUENCE	TYPE	DATA
0	Text node	First
1	Text node	Second

Imagine that `someFlag` becomes `false`, and the markup is rendered again. This time, the builder receives:

SEQUENCE	TYPE	DATA
1	Text node	Second

When the runtime performs a diff, it sees that the item at sequence `0` was removed, so it generates the following trivial *edit script*:

- Remove the first text node.

The problem with generating sequence numbers programmatically

Imagine instead that you wrote the following render tree builder logic:

```
var seq = 0;

if (someFlag)
{
    builder.AddContent(seq++, "First");
}

builder.AddContent(seq++, "Second");
```

Now, the first output is:

SEQUENCE	TYPE	DATA
0	Text node	First
1	Text node	Second

This outcome is identical to the prior case, so no negative issues exist. `someFlag` is `false` on the second rendering, and the output is:

SEQUENCE	TYPE	DATA
0	Text node	Second

This time, the diff algorithm sees that *two* changes have occurred, and the algorithm generates the following edit script:

- Change the value of the first text node to `Second`.
- Remove the second text node.

Generating the sequence numbers has lost all the useful information about where the `if/else` branches and loops were present in the original code. This results in a diff **twice as long** as before.

This is a trivial example. In more realistic cases with complex and deeply nested structures, and especially with loops, the performance cost is usually higher. Instead of immediately identifying which loop blocks or branches have been inserted or removed, the diff algorithm has to recurse deeply into the render trees. This usually results in having to build longer edit scripts because the diff algorithm is misinformed about how the old and new structures relate to each other.

Guidance and conclusions

- App performance suffers if sequence numbers are generated dynamically.
- The framework can't create its own sequence numbers automatically at runtime because the necessary information doesn't exist unless it's captured at compile time.
- Don't write long blocks of manually-implemented RenderTreeBuilder logic. Prefer `.razor` files and allow the compiler to deal with the sequence numbers. If you're unable to avoid manual RenderTreeBuilder logic, split long blocks of code into smaller pieces wrapped in OpenRegion/CloseRegion calls. Each region has its own separate space of sequence numbers, so you can restart from zero (or any other arbitrary number) inside each region.
- If sequence numbers are hardcoded, the diff algorithm only requires that sequence numbers increase in value. The initial value and gaps are irrelevant. One legitimate option is to use the code line number as the sequence number, or start from zero and increase by ones or hundreds (or any preferred interval).
- Blazor uses sequence numbers, while other tree-diffing UI frameworks don't use them. Diffing is far faster when sequence numbers are used, and Blazor has the advantage of a compile step that deals with sequence numbers automatically for developers authoring `.razor` files.

Perform large data transfers in Blazor Server apps

In some scenarios, large amounts of data must be transferred between JavaScript and Blazor. Typically, large data transfers occur when:

- Browser file system APIs are used to upload or download a file.
- Interop with a third party library is required.

In Blazor Server, a limitation is in place to prevent passing single large messages that may result in performance issues.

Consider the following guidance when developing code that transfers data between JavaScript and Blazor:

- Slice the data into smaller pieces, and send the data segments sequentially until all of the data is received by the server.
- Don't allocate large objects in JavaScript and C# code.
- Don't block the main UI thread for long periods when sending or receiving data.
- Free any memory consumed when the process is completed or cancelled.
- Enforce the following additional requirements for security purposes:
  - Declare the maximum file or data size that can be passed.
  - Declare the minimum upload rate from the client to the server.
- After the data is received by the server, the data can be:
  - Temporarily stored in a memory buffer until all of the segments are collected.
  - Consumed immediately. For example, the data can be stored immediately in a database or written to disk as each segment is received.

The following file uploader class handles JS interop with the client. The uploader class uses JS interop to:

- Poll the client to send a data segment.
- Abort the transaction if polling times out.

```
using System;
using System.Buffers;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
using Microsoft.JSInterop;

public class FileUploader : IDisposable
{
    private readonly IJSRuntime jsRuntime;
    private readonly int segmentSize = 6144;
    private readonly int maxBase64SegmentSize = 8192;
    private readonly DotNetObjectReference<FileUploader> thisReference;
    private List<IMemoryOwner<byte>> uploadedSegments =
        new List<IMemoryOwner<byte>>();

    public FileUploader(IJSRuntime jsRuntime)
    {
        this.jsRuntime = jsRuntime;
    }

    public async Task<Stream> ReceiveFile(string selector, int maxSize)
    {
        var fileSize =
            await jsRuntime.InvokeAsync<int>("getFileSize", selector);

        if (fileSize > maxSize)
        {
            return null;
        }

        var numberOfSegments = Math.Floor(fileSize / (double)segmentSize) + 1;
        var lastSegmentBytes = 0;
        string base64EncodedSegment;

        for (var i = 0; i < numberOfSegments; i++)
        {
            try
            {
                base64EncodedSegment =
                    await jsRuntime.InvokeAsync<string>(
                        "receiveSegment", i, selector);

                if (base64EncodedSegment.Length < maxBase64SegmentSize &&
                    i < numberOfSegments - 1)
                {
                    return null;
                }
            }
            catch
            {
                return null;
            }

          var current = MemoryPool<byte>.Shared.Rent(segmentSize);

          if (!Convert.TryFromBase64String(base64EncodedSegment,
              current.Memory.Slice(0, segmentSize).Span, out lastSegmentBytes))
          {
              return null;
          }
```

```
            uploadedSegments.Add(current);
        }

        var segments = uploadedSegments;
        uploadedSegments = null;

        return new SegmentedStream(segments, segmentSize, lastSegmentBytes);
    }

    public void Dispose()
    {
        if (uploadedSegments != null)
        {
            foreach (var segment in uploadedSegments)
            {
                segment.Dispose();
            }
        }
    }
}
```

In the preceding example:

- The `maxBase64SegmentSize` is set to `8192`, which is calculated from `maxBase64SegmentSize = segmentSize * 4 / 3`.
- Low-level .NET Core memory management APIs are used to store the memory segments on the server in `uploadedSegments`.
- A `ReceiveFile` method is used to handle the upload through JS interop:
  - The file size is determined in bytes through JS interop with `jsRuntime.InvokeAsync<FileInfo>('getFileSize', selector)`.
  - The number of segments to receive are calculated and stored in `numberOfSegments`.
  - The segments are requested in a `for` loop through JS interop with `jsRuntime.InvokeAsync<string>('receiveSegment', i, selector)`. All segments but the last must be 8,192 bytes before decoding. The client is forced to send the data in an efficient manner.
  - For each segment received, checks are performed before decoding with TryFromBase64String.
  - A stream with the data is returned as a new Stream (`SegmentedStream`) after the upload is complete.

The segmented stream class exposes the list of segments as a readonly non-seekable Stream:

```
using System;
using System.Buffers;
using System.Collections.Generic;
using System.IO;

public class SegmentedStream : Stream
{
    private readonly ReadOnlySequence<byte> sequence;
    private long currentPosition = 0;

    public SegmentedStream(IList<IMemoryOwner<byte>> segments, int segmentSize,
        int lastSegmentSize)
    {
        if (segments.Count == 1)
        {
            sequence = new ReadOnlySequence<byte>(
                segments[0].Memory.Slice(0, lastSegmentSize));
            return;
        }

        var sequenceSegment = new BufferSegment<byte>(
            segments[0].Memory.Slice(0, segmentSize));
        var lastSegment = sequenceSegment;
```

```
        for (int i = 1; i < segments.Count; i++)
        {
            var isLastSegment = i + 1 == segments.Count;
            lastSegment = lastSegment.Append(segments[i].Memory.Slice(
                0, isLastSegment ? lastSegmentSize : segmentSize));
        }

        sequence = new ReadOnlySequence<byte>(
            sequenceSegment, 0, lastSegment, lastSegmentSize);
    }

    public override long Position
    {
        get => throw new NotImplementedException();
        set => throw new NotImplementedException();
    }

    public override int Read(byte[] buffer, int offset, int count)
    {
        var bytesToWrite = (int)(currentPosition + count < sequence.Length ?
            count : sequence.Length - currentPosition);
        var data = sequence.Slice(currentPosition, bytesToWrite);
        data.CopyTo(buffer.AsSpan(offset, bytesToWrite));
        currentPosition += bytesToWrite;

        return bytesToWrite;
    }

    private class BufferSegment<T> : ReadOnlySequenceSegment<T>
    {
        public BufferSegment(ReadOnlyMemory<T> memory)
        {
            Memory = memory;
        }

        public BufferSegment<T> Append(ReadOnlyMemory<T> memory)
        {
            var segment = new BufferSegment<T>(memory)
            {
                RunningIndex = RunningIndex + Memory.Length
            };

            Next = segment;

            return segment;
        }
    }

    public override bool CanRead => true;

    public override bool CanSeek => false;

    public override bool CanWrite => false;

    public override long Length => throw new NotImplementedException();

    public override void Flush() => throw new NotImplementedException();

    public override long Seek(long offset, SeekOrigin origin) =>
        throw new NotImplementedException();

    public override void SetLength(long value) =>
        throw new NotImplementedException();

    public override void Write(byte[] buffer, int offset, int count) =>
        throw new NotImplementedException();
}
```

The following code implements JavaScript functions to receive the data:

```
function getFileSize(selector) {
  const file = getFile(selector);
  return file.size;
}

async function receiveSegment(segmentNumber, selector) {
  const file = getFile(selector);
  var segments = getFileSegments(file);
  var index = segmentNumber * 6144;
  return await getNextChunk(file, index);
}

function getFile(selector) {
  const element = document.querySelector(selector);
  if (!element) {
    throw new Error('Invalid selector');
  }
  const files = element.files;
  if (!files || files.length === 0) {
    throw new Error(`Element ${elementId} doesn't contain any files.`);
  }
  const file = files[0];
  return file;
}

function getFileSegments(file) {
  const segments = Math.floor(size % 6144 === 0 ? size / 6144 : 1 + size / 6144);
  return segments;
}

async function getNextChunk(file, index) {
  const length = file.size - index <= 6144 ? file.size - index : 6144;
  const chunk = file.slice(index, index + length);
  index += length;
  const base64Chunk = await this.base64EncodeAsync(chunk);
  return { base64Chunk, index };
}

async function base64EncodeAsync(chunk) {
  const reader = new FileReader();
  const result = new Promise((resolve, reject) => {
    reader.addEventListener('load',
      () => {
        const base64Chunk = reader.result;
        const cleanChunk =
          base64Chunk.replace('data:application/octet-stream;base64,', '');
        resolve(cleanChunk);
      },
      false);
    reader.addEventListener('error', reject);
  });
  reader.readAsDataURL(chunk);
  return result;
}
```

Use the Angular project template with ASP.NET Core

9/22/2020 • 6 minutes to read • Edit Online

The updated Angular project template provides a convenient starting point for ASP.NET Core apps using Angular and the Angular CLI to implement a rich, client-side user interface (UI).

The template is equivalent to creating an ASP.NET Core project to act as an API backend and an Angular CLI project to act as a UI. The template offers the convenience of hosting both project types in a single app project. Consequently, the app project can be built and published as a single unit.

Create a new app

If you have ASP.NET Core 2.1 installed, there's no need to install the Angular project template.

Create a new project from a command prompt using the command `dotnet new angular` in an empty directory. For example, the following commands create the app in a *my-new-app* directory and switch to that directory:

```
dotnet new angular -o my-new-app
cd my-new-app
```

Run the app from either Visual Studio or the .NET Core CLI:

- Visual Studio
- .NET Core CLI

Open the generated *.csproj* file, and run the app as normal from there.

The build process restores npm dependencies on the first run, which can take several minutes. Subsequent builds are much faster.

The project template creates an ASP.NET Core app and an Angular app. The ASP.NET Core app is intended to be used for data access, authorization, and other server-side concerns. The Angular app, residing in the *ClientApp* subdirectory, is intended to be used for all UI concerns.

Add pages, images, styles, modules, etc.

The *ClientApp* directory contains a standard Angular CLI app. See the official Angular documentation for more information.

There are slight differences between the Angular app created by this template and the one created by Angular CLI itself (via `ng new`); however, the app's capabilities are unchanged. The app created by the template contains a Bootstrap-based layout and a basic routing example.

Run ng commands

In a command prompt, switch to the *ClientApp* subdirectory:

```
cd ClientApp
```

If you have the `ng` tool installed globally, you can run any of its commands. For example, you can run `ng lint`, `ng test`, or any of the other Angular CLI commands. There's no need to run `ng serve` though, because your

ASP.NET Core app deals with serving both server-side and client-side parts of your app. Internally, it uses `ng serve` in development.

If you don't have the `ng` tool installed, run `npm run ng` instead. For example, you can run `npm run ng lint` or `npm run ng test`.

Install npm packages

To install third-party npm packages, use a command prompt in the *ClientApp* subdirectory. For example:

```
cd ClientApp
npm install --save <package_name>
```

Publish and deploy

In development, the app runs in a mode optimized for developer convenience. For example, JavaScript bundles include source maps (so that when debugging, you can see your original TypeScript code). The app watches for TypeScript, HTML, and CSS file changes on disk and automatically recompiles and reloads when it sees those files change.

In production, serve a version of your app that's optimized for performance. This is configured to happen automatically. When you publish, the build configuration emits a minified, ahead-of-time (AoT) compiled build of your client-side code. Unlike the development build, the production build doesn't require Node.js to be installed on the server (unless you have enabled server-side rendering (SSR)).

You can use standard ASP.NET Core hosting and deployment methods.

Run "ng serve" independently

The project is configured to start its own instance of the Angular CLI server in the background when the ASP.NET Core app starts in development mode. This is convenient because you don't have to run a separate server manually.

There's a drawback to this default setup. Each time you modify your C# code and your ASP.NET Core app needs to restart, the Angular CLI server restarts. Around 10 seconds is required to start back up. If you're making frequent C# code edits and don't want to wait for Angular CLI to restart, run the Angular CLI server externally, independently of the ASP.NET Core process. To do so:

1. In a command prompt, switch to the *ClientApp* subdirectory, and launch the Angular CLI development server:

   ```
   cd ClientApp
   npm start
   ```

   IMPORTANT

   Use `npm start` to launch the Angular CLI development server, not `ng serve`, so that the configuration in *package.json* is respected. To pass additional parameters to the Angular CLI server, add them to the relevant `scripts` line in your *package.json* file.

2. Modify your ASP.NET Core app to use the external Angular CLI instance instead of launching one of its own. In your *Startup* class, replace the `spa.UseAngularCliServer` invocation with the following:

```
spa.UseProxyToSpaDevelopmentServer("http://localhost:4200");
```

When you start your ASP.NET Core app, it won't launch an Angular CLI server. The instance you started manually is used instead. This enables it to start and restart faster. It's no longer waiting for Angular CLI to rebuild your client app each time.

Pass data from .NET code into TypeScript code

During SSR, you might want to pass per-request data from your ASP.NET Core app into your Angular app. For example, you could pass cookie information or something read from a database. To do this, edit your *Startup* class. In the callback for `UseSpaPrerendering`, set a value for `options.SupplyData` such as the following:

```
options.SupplyData = (context, data) =>
{
    // Creates a new value called isHttpsRequest that's passed to TypeScript code
    data["isHttpsRequest"] = context.Request.IsHttps;
};
```

The `SupplyData` callback lets you pass arbitrary, per-request, JSON-serializable data (for example, strings, booleans, or numbers). Your *main.server.ts* code receives this as `params.data`. For example, the preceding code sample passes a boolean value as `params.data.isHttpsRequest` into the `createServerRenderer` callback. You can pass this to other parts of your app in any way supported by Angular. For example, see how *main.server.ts* passes the `BASE_URL` value to any component whose constructor is declared to receive it.

Drawbacks of SSR

Not all apps benefit from SSR. The primary benefit is perceived performance. Visitors reaching your app over a slow network connection or on slow mobile devices see the initial UI quickly, even if it takes a while to fetch or parse the JavaScript bundles. However, many SPAs are mainly used over fast, internal company networks on fast computers where the app appears almost instantly.

At the same time, there are significant drawbacks to enabling SSR. It adds complexity to your development process. Your code must run in two different environments: client-side and server-side (in a Node.js environment invoked from ASP.NET Core). Here are some things to bear in mind:

- SSR requires a Node.js installation on your production servers. This is automatically the case for some deployment scenarios, such as Azure App Services, but not for others, such as Azure Service Fabric.
- Enabling the `BuildServerSideRenderer` build flag causes your *node_modules* directory to publish. This folder contains 20,000+ files, which increases deployment time.
- To run your code in a Node.js environment, it can't rely on the existence of browser-specific JavaScript APIs such as `window` or `localStorage`. If your code (or some third-party library you reference) tries to use these APIs, you'll get an error during SSR. For example, don't use jQuery because it references browser-specific APIs in many places. To prevent errors, you must either avoid SSR or avoid browser-specific APIs or libraries. You can wrap any calls to such APIs in checks to ensure they aren't invoked during SSR. For example, use a check such as the following in JavaScript or TypeScript code:

  ```
  if (typeof window !== 'undefined') {
      // Call browser-specific APIs here
  }
  ```

Additional resources

- Introduction to authentication for Single Page Apps on ASP.NET Core

Use the React project template with ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

The updated React project template provides a convenient starting point for ASP.NET Core apps using React and create-react-app (CRA) conventions to implement a rich, client-side user interface (UI).

The template is equivalent to creating both an ASP.NET Core project to act as an API backend, and a standard CRA React project to act as a UI, but with the convenience of hosting both in a single app project that can be built and published as a single unit.

The React project template isn't meant for server-side rendering (SSR). For SSR with React and Node.js, consider Next.js or Razzle.

Create a new app

If you have ASP.NET Core 2.1 installed, there's no need to install the React project template.

Create a new project from a command prompt using the command `dotnet new react` in an empty directory. For example, the following commands create the app in a *my-new-app* directory and switch to that directory:

```
dotnet new react -o my-new-app
cd my-new-app
```

Run the app from either Visual Studio or the .NET Core CLI:

- Visual Studio
- .NET Core CLI

Open the generated *.csproj* file, and run the app as normal from there.

The build process restores npm dependencies on the first run, which can take several minutes. Subsequent builds are much faster.

The project template creates an ASP.NET Core app and a React app. The ASP.NET Core app is intended to be used for data access, authorization, and other server-side concerns. The React app, residing in the *ClientApp* subdirectory, is intended to be used for all UI concerns.

Add pages, images, styles, modules, etc.

The *ClientApp* directory is a standard CRA React app. See the official CRA documentation for more information.

There are slight differences between the React app created by this template and the one created by CRA itself; however, the app's capabilities are unchanged. The app created by the template contains a Bootstrap-based layout and a basic routing example.

Install npm packages

To install third-party npm packages, use a command prompt in the *ClientApp* subdirectory. For example:

```
cd ClientApp
npm install --save <package_name>
```

Publish and deploy

In development, the app runs in a mode optimized for developer convenience. For example, JavaScript bundles include source maps (so that when debugging, you can see your original source code). The app watches JavaScript, HTML, and CSS file changes on disk and automatically recompiles and reloads when it sees those files change.

In production, serve a version of your app that's optimized for performance. This is configured to happen automatically. When you publish, the build configuration emits a minified, transpiled build of your client-side code. Unlike the development build, the production build doesn't require Node.js to be installed on the server.

You can use standard ASP.NET Core hosting and deployment methods.

Run the CRA server independently

The project is configured to start its own instance of the CRA development server in the background when the ASP.NET Core app starts in development mode. This is convenient because it means you don't have to run a separate server manually.

There's a drawback to this default setup. Each time you modify your C# code and your ASP.NET Core app needs to restart, the CRA server restarts. A few seconds are required to start back up. If you're making frequent C# code edits and don't want to wait for the CRA server to restart, run the CRA server externally, independently of the ASP.NET Core process. To do so:

1. Add a *.env* file to the *ClientApp* subdirectory with the following setting:

   ```
   BROWSER=none
   ```

   This will prevent your web browser from opening when starting the CRA server externally.

2. In a command prompt, switch to the *ClientApp* subdirectory, and launch the CRA development server:

   ```
   cd ClientApp
   npm start
   ```

3. Modify your ASP.NET Core app to use the external CRA server instance instead of launching one of its own. In your *Startup* class, replace the `spa.UseReactDevelopmentServer` invocation with the following:

   ```
   spa.UseProxyToSpaDevelopmentServer("http://localhost:3000");
   ```

When you start your ASP.NET Core app, it won't launch a CRA server. The instance you started manually is used instead. This enables it to start and restart faster. It's no longer waiting for your React app to rebuild each time.

> **IMPORTANT**
> "Server-side rendering" is not a supported feature of this template. Our goal with this template is to meet parity with "create-react-app". As such, scenarios and features not included in a "create-react-app" project (such as SSR) are not supported and are left as an exercise for the user.

Additional resources

- Introduction to authentication for Single Page Apps on ASP.NET Core

Use the React-with-Redux project template with ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

The updated React-with-Redux project template provides a convenient starting point for ASP.NET Core apps using React, Redux, and create-react-app (CRA) conventions to implement a rich, client-side user interface (UI).

With the exception of the project creation command, all information about the React-with-Redux template is the same as the React template. To create this project type, run `dotnet new reactredux` instead of `dotnet new react`. For more information about the functionality common to both React-based templates, see React template documentation.

For information on configuring a React-with-Redux sub-application in IIS, see ReactRedux Template 2.1: Unable to use SPA on IIS (aspnet/Templating #555).

Use JavaScript Services to Create Single Page Applications in ASP.NET Core

9/22/2020 • 11 minutes to read • Edit Online

By Scott Addie and Fiyaz Hasan

A Single Page Application (SPA) is a popular type of web application due to its inherent rich user experience. Integrating client-side SPA frameworks or libraries, such as Angular or React, with server-side frameworks such as ASP.NET Core can be difficult. JavaScript Services was developed to reduce friction in the integration process. It enables seamless operation between the different client and server technology stacks.

> **WARNING**
> The features described in this article are obsolete as of ASP.NET Core 3.0. A simpler SPA frameworks integration mechanism is available in the Microsoft.AspNetCore.SpaServices.Extensions NuGet package. For more information, see [Announcement] Obsoleting Microsoft.AspNetCore.SpaServices and Microsoft.AspNetCore.NodeServices.

What is JavaScript Services

JavaScript Services is a collection of client-side technologies for ASP.NET Core. Its goal is to position ASP.NET Core as developers' preferred server-side platform for building SPAs.

JavaScript Services consists of two distinct NuGet packages:

- Microsoft.AspNetCore.NodeServices (NodeServices)
- Microsoft.AspNetCore.SpaServices (SpaServices)

These packages are useful in the following scenarios:

- Run JavaScript on the server
- Use a SPA framework or library
- Build client-side assets with Webpack

Much of the focus in this article is placed on using the SpaServices package.

What is SpaServices

SpaServices was created to position ASP.NET Core as developers' preferred server-side platform for building SPAs. SpaServices isn't required to develop SPAs with ASP.NET Core, and it doesn't lock developers into a particular client framework.

SpaServices provides useful infrastructure such as:

- Server-side prerendering
- Webpack Dev Middleware
- Hot Module Replacement
- Routing helpers

Collectively, these infrastructure components enhance both the development workflow and the runtime experience. The components can be adopted individually.

Prerequisites for using SpaServices

To work with SpaServices, install the following:

- Node.js (version 6 or later) with npm
  - To verify these components are installed and can be found, run the following from the command line:

    ```
    node -v && npm -v
    ```

  - If deploying to an Azure web site, no action is required—Node.js is installed and available in the server environments.
- .NET Core SDK 2.0 or later
  - On Windows using Visual Studio 2017, the SDK is installed by selecting the **.NET Core cross-platform development** workload.
- Microsoft.AspNetCore.SpaServices NuGet package

Server-side prerendering

A universal (also known as isomorphic) application is a JavaScript application capable of running both on the server and the client. Angular, React, and other popular frameworks provide a universal platform for this application development style. The idea is to first render the framework components on the server via Node.js, and then delegate further execution to the client.

ASP.NET Core Tag Helpers provided by SpaServices simplify the implementation of server-side prerendering by invoking the JavaScript functions on the server.

Server-side prerendering prerequisites

Install the aspnet-prerendering npm package:

```
npm i -S aspnet-prerendering
```

Server-side prerendering configuration

The Tag Helpers are made discoverable via namespace registration in the project's *_ViewImports.cshtml* file:

```
@using SpaServicesSampleApp
@addTagHelper "*, Microsoft.AspNetCore.Mvc.TagHelpers"
@addTagHelper "*, Microsoft.AspNetCore.SpaServices"
```

These Tag Helpers abstract away the intricacies of communicating directly with low-level APIs by leveraging an HTML-like syntax inside the Razor view:

```
<app asp-prerender-module="ClientApp/dist/main-server">Loading...</app>
```

asp-prerender-module Tag Helper

The `asp-prerender-module` Tag Helper, used in the preceding code example, executes *ClientApp/dist/main-server.js* on the server via Node.js. For clarity's sake, *main-server.js* file is an artifact of the TypeScript-to-JavaScript transpilation task in the Webpack build process. Webpack defines an entry point alias of `main-server`; and, traversal of the dependency graph for this alias begins at the *ClientApp/boot-server.ts* file:

```
entry: { 'main-server': './ClientApp/boot-server.ts' },
```

In the following Angular example, the *ClientApp/boot-server.ts* file utilizes the `createServerRenderer` function and `RenderResult` type of the `aspnet-prerendering` npm package to configure server rendering via Node.js. The HTML markup destined for server-side rendering is passed to a resolve function call, which is wrapped in a strongly-typed JavaScript `Promise` object. The `Promise` object's significance is that it asynchronously supplies the HTML markup to the page for injection in the DOM's placeholder element.

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: state.renderToString()
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

asp-prerender-data Tag Helper

When coupled with the `asp-prerender-module` Tag Helper, the `asp-prerender-data` Tag Helper can be used to pass contextual information from the Razor view to the server-side JavaScript. For example, the following markup passes user data to the `main-server` module:

```
<app asp-prerender-module="ClientApp/dist/main-server"
        asp-prerender-data='new {
            UserName = "John Doe"
        }'>Loading...</app>
```

The received `UserName` argument is serialized using the built-in JSON serializer and is stored in the `params.data` object. In the following Angular example, the data is used to construct a personalized greeting within an `h1` element:

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            const result = `<h1>Hello, ${params.data.userName}</h1>`;

            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: result
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

Property names passed in Tag Helpers are represented with **PascalCase** notation. Contrast that to JavaScript, where the same property names are represented with **camelCase**. The default JSON serialization configuration is responsible for this difference.

To expand upon the preceding code example, data can be passed from the server to the view by hydrating the `globals` property provided to the `resolve` function:

```
import { createServerRenderer, RenderResult } from 'aspnet-prerendering';

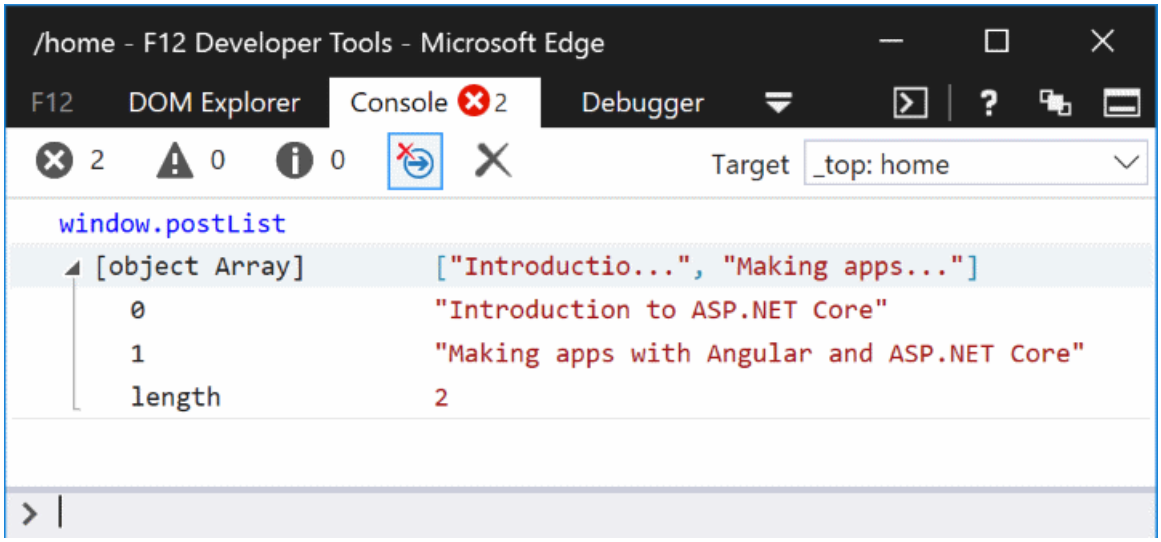
export default createServerRenderer(params => {
    const providers = [
        { provide: INITIAL_CONFIG, useValue: { document: '<app></app>', url: params.url } },
        { provide: 'ORIGIN_URL', useValue: params.origin }
    ];

    return platformDynamicServer(providers).bootstrapModule(AppModule).then(moduleRef => {
        const appRef = moduleRef.injector.get(ApplicationRef);
        const state = moduleRef.injector.get(PlatformState);
        const zone = moduleRef.injector.get(NgZone);

        return new Promise<RenderResult>((resolve, reject) => {
            const result = `<h1>Hello, ${params.data.userName}</h1>`;

            zone.onError.subscribe(errorInfo => reject(errorInfo));
            appRef.isStable.first(isStable => isStable).subscribe(() => {
                // Because 'onStable' fires before 'onError', we have to delay slightly before
                // completing the request in case there's an error to report
                setImmediate(() => {
                    resolve({
                        html: result,
                        globals: {
                            postList: [
                                'Introduction to ASP.NET Core',
                                'Making apps with Angular and ASP.NET Core'
                            ]
                        }
                    });
                    moduleRef.destroy();
                });
            });
        });
    });
});
```

The `postList` array defined inside the `globals` object is attached to the browser's global `window` object. This variable hoisting to global scope eliminates duplication of effort, particularly as it pertains to loading the same data once on the server and again on the client.



Webpack Dev Middleware

Webpack Dev Middleware introduces a streamlined development workflow whereby Webpack builds resources on demand. The middleware automatically compiles and serves client-side resources when a page is reloaded in the browser. The alternate approach is to manually invoke Webpack via the project's npm build script when a third-

party dependency or the custom code changes. An npm build script in the *package.json* file is shown in the following example:

```
"build": "npm run build:vendor && npm run build:custom",
```

Webpack Dev Middleware prerequisites

Install the aspnet-webpack npm package:

```
npm i -D aspnet-webpack
```

Webpack Dev Middleware configuration

Webpack Dev Middleware is registered into the HTTP request pipeline via the following code in the *Startup.cs* file's `Configure` method:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseWebpackDevMiddleware();
}
else
{
    app.UseExceptionHandler("/Home/Error");
}

// Call UseWebpackDevMiddleware before UseStaticFiles
app.UseStaticFiles();
```

The `UseWebpackDevMiddleware` extension method must be called before registering static file hosting via the `UseStaticFiles` extension method. For security reasons, register the middleware only when the app runs in development mode.

The *webpack.config.js* file's `output.publicPath` property tells the middleware to watch the `dist` folder for changes:

```
module.exports = (env) => {
        output: {
            filename: '[name].js',
            publicPath: '/dist/' // Webpack dev middleware, if enabled, handles requests for this URL prefix
        },
```

Hot Module Replacement

Think of Webpack's Hot Module Replacement (HMR) feature as an evolution of Webpack Dev Middleware. HMR introduces all the same benefits, but it further streamlines the development workflow by automatically updating page content after compiling the changes. Don't confuse this with a refresh of the browser, which would interfere with the current in-memory state and debugging session of the SPA. There's a live link between the Webpack Dev Middleware service and the browser, which means changes are pushed to the browser.

Hot Module Replacement prerequisites

Install the webpack-hot-middleware npm package:

```
npm i -D webpack-hot-middleware
```

Hot Module Replacement configuration

The HMR component must be registered into MVC's HTTP request pipeline in the `Configure` method:

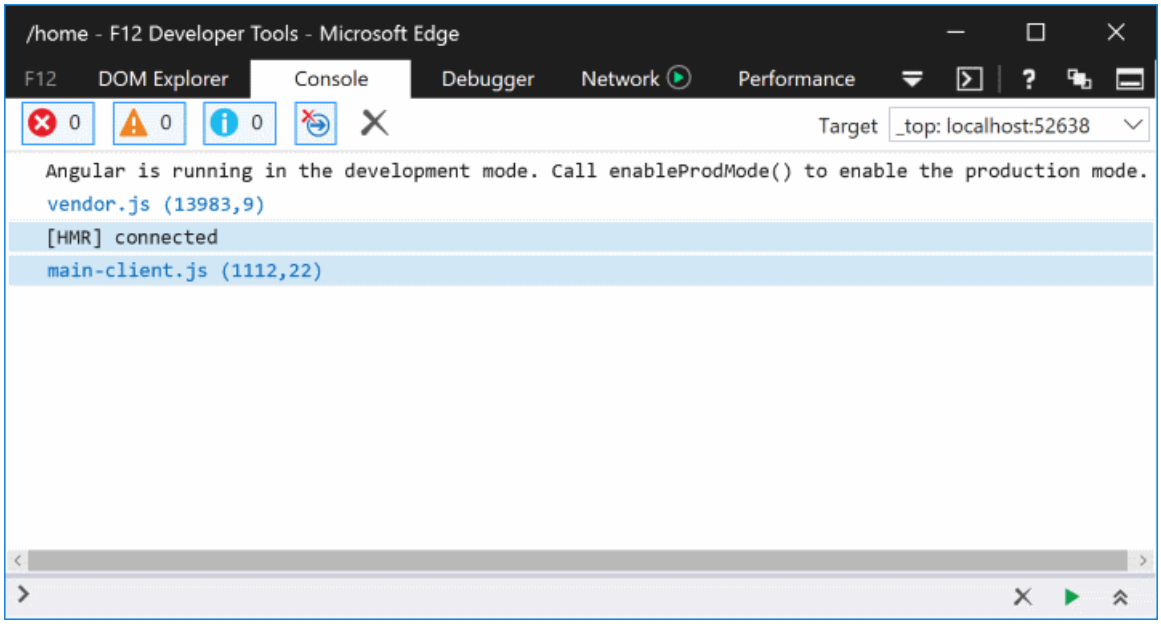
```
app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {
    HotModuleReplacement = true
});
```

As was true with Webpack Dev Middleware, the `UseWebpackDevMiddleware` extension method must be called before the `UseStaticFiles` extension method. For security reasons, register the middleware only when the app runs in development mode.

The *webpack.config.js* file must define a `plugins` array, even if it's left empty:

```
module.exports = (env) => {
        plugins: [new CheckerPlugin()]
```

After loading the app in the browser, the developer tools' Console tab provides confirmation of HMR activation:



Routing helpers

In most ASP.NET Core-based SPAs, client-side routing is often desired in addition to server-side routing. The SPA and MVC routing systems can work independently without interference. There's, however, one edge case posing challenges: identifying 404 HTTP responses.

Consider the scenario in which an extensionless route of `/some/page` is used. Assume the request doesn't pattern-match a server-side route, but its pattern does match a client-side route. Now consider an incoming request for `/images/user-512.png`, which generally expects to find an image file on the server. If that requested resource path doesn't match any server-side route or static file, it's unlikely that the client-side application would handle it—generally returning a 404 HTTP status code is desired.

Routing helpers prerequisites

Install the client-side routing npm package. Using Angular as an example:

```
npm i -S @angular/router
```

Routing helpers configuration

An extension method named `MapSpaFallbackRoute` is used in the `Configure` method:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");

    routes.MapSpaFallbackRoute(
        name: "spa-fallback",
        defaults: new { controller = "Home", action = "Index" });
});
```

Routes are evaluated in the order in which they're configured. Consequently, the `default` route in the preceding code example is used first for pattern matching.

Create a new project

JavaScript Services provide pre-configured application templates. SpaServices is used in these templates in conjunction with different frameworks and libraries such as Angular, React, and Redux.

These templates can be installed via the .NET Core CLI by running the following command:

```
dotnet new --install Microsoft.AspNetCore.SpaTemplates::*
```

A list of available SPA templates is displayed:

TEMPLATES	SHORT NAME	LANGUAGE	TAGS
MVC ASP.NET Core with Angular	angular	[C#]	Web/MVC/SPA
MVC ASP.NET Core with React.js	react	[C#]	Web/MVC/SPA
MVC ASP.NET Core with React.js and Redux	reactredux	[C#]	Web/MVC/SPA

To create a new project using one of the SPA templates, include the **Short Name** of the template in the dotnet new command. The following command creates an Angular application with ASP.NET Core MVC configured for the server side:

```
dotnet new angular
```

Set the runtime configuration mode

Two primary runtime configuration modes exist:

- **Development**:
  - Includes source maps to ease debugging.
  - Doesn't optimize the client-side code for performance.
- **Production**:
  - Excludes source maps.
  - Optimizes the client-side code via bundling and minification.

ASP.NET Core uses an environment variable named `ASPNETCORE_ENVIRONMENT` to store the configuration mode. For more information, see Set the environment.

Run with .NET Core CLI

Restore the required NuGet and npm packages by running the following command at the project root:

```
dotnet restore && npm i
```

Build and run the application:

```
dotnet run
```

The application starts on localhost according to the runtime configuration mode. Navigating to `http://localhost:5000` in the browser displays the landing page.

Run with Visual Studio 2017

Open the *.csproj* file generated by the dotnet new command. The required NuGet and npm packages are restored automatically upon project open. This restoration process may take up to a few minutes, and the application is ready to run when it completes. Click the green run button or press `Ctrl + F5`, and the browser opens to the application's landing page. The application runs on localhost according to the runtime configuration mode.

Test the app

SpaServices templates are pre-configured to run client-side tests using Karma and Jasmine. Jasmine is a popular unit testing framework for JavaScript, whereas Karma is a test runner for those tests. Karma is configured to work with the Webpack Dev Middleware such that the developer isn't required to stop and run the test every time changes are made. Whether it's the code running against the test case or the test case itself, the test runs automatically.

Using the Angular application as an example, two Jasmine test cases are already provided for the `CounterComponent` in the *counter.component.spec.ts* file:

```
it('should display a title', async(() => {
    const titleText = fixture.nativeElement.querySelector('h1').textContent;
    expect(titleText).toEqual('Counter');
}));

it('should start with count 0, then increments by 1 when clicked', async(() => {
    const countElement = fixture.nativeElement.querySelector('strong');
    expect(countElement.textContent).toEqual('0');

    const incrementButton = fixture.nativeElement.querySelector('button');
    incrementButton.click();
    fixture.detectChanges();
    expect(countElement.textContent).toEqual('1');
}));
```

Open the command prompt in the *ClientApp* directory. Run the following command:

```
npm test
```

The script launches the Karma test runner, which reads the settings defined in the *karma.conf.js* file. Among other settings, the *karma.conf.js* identifies the test files to be executed via its `files` array:

```
module.exports = function (config) {
    config.set({
        files: [
            '../../wwwroot/dist/vendor.js',
            './boot-tests.ts'
        ],
```

Publish the app

See this GitHub issue for more information on publishing to Azure.

Combining the generated client-side assets and the published ASP.NET Core artifacts into a ready-to-deploy package can be cumbersome. Thankfully, SpaServices orchestrates that entire publication process with a custom MSBuild target named `RunWebpack`:

```
<Target Name="RunWebpack" AfterTargets="ComputeFilesToPublish">
  <!-- As part of publishing, ensure the JS resources are freshly built in production mode -->
  <Exec Command="npm install" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --config webpack.config.vendor.js --env.prod" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --env.prod" />

  <!-- Include the newly-built files in the publish output -->
  <ItemGroup>
    <DistFiles Include="wwwroot\dist\**; ClientApp\dist\**" />
    <ResolvedFileToPublish Include="@(DistFiles->'%(FullPath)')" Exclude="@(ResolvedFileToPublish)">
      <RelativePath>%(DistFiles.Identity)</RelativePath>
      <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
    </ResolvedFileToPublish>
  </ItemGroup>
</Target>
```

The MSBuild target has the following responsibilities:

1. Restore the npm packages.
2. Create a production-grade build of the third-party, client-side assets.
3. Create a production-grade build of the custom client-side assets.
4. Copy the Webpack-generated assets to the publish folder.

The MSBuild target is invoked when running:

```
dotnet publish -c Release
```

Additional resources

- Angular Docs

Client-side library acquisition in ASP.NET Core with LibMan

9/22/2020 • 2 minutes to read • Edit Online

By Scott Addie

Library Manager (LibMan) is a lightweight, client-side library acquisition tool. LibMan downloads popular libraries and frameworks from the file system or from a content delivery network (CDN). The supported CDNs include CDNJS, jsDelivr, and unpkg. The selected library files are fetched and placed in the appropriate location within the ASP.NET Core project.

LibMan use cases

LibMan offers the following benefits:

- Only the library files you need are downloaded.
- Additional tooling, such as Node.js, npm, and WebPack, isn't necessary to acquire a subset of files in a library.
- Files can be placed in a specific location without resorting to build tasks or manual file copying.

For more information about LibMan's benefits, watch Modern front-end web development in Visual Studio 2017: LibMan segment.

LibMan isn't a package management system. If you're already using a package manager, such as npm or yarn, continue doing so. LibMan wasn't developed to replace those tools.

Additional resources

- Use LibMan with ASP.NET Core in Visual Studio
- Use the LibMan CLI with ASP.NET Core
- LibMan GitHub repository

Use the LibMan CLI with ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

By Scott Addie

The LibMan CLI is a cross-platform tool that's supported everywhere .NET Core is supported.

Prerequisites

- .NET Core 2.1 SDK or later

Installation

To install the LibMan CLI:

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

A .NET Core Global Tool is installed from the Microsoft.Web.LibraryManager.Cli NuGet package.

To install the LibMan CLI from a specific NuGet package source:

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli --version 1.0.94-g606058a278 --add-source C:\Temp\
```

In the preceding example, a .NET Core Global Tool is installed from the local Windows machine's *C:\Temp\Microsoft.Web.LibraryManager.Cli.1.0.94-g606058a278.nupkg* file.

Usage

After successful installation of the CLI, the following command can be used:

```
libman
```

To view the installed CLI version:

```
libman --version
```

To view the available CLI commands:

```
libman --help
```

The preceding command displays output similar to the following:

```
 1.0.163+g45474d37ed

Usage: libman [options] [command]

Options:
  --help|-h  Show help information
  --version  Show version information

Commands:
  cache      List or clean libman cache contents
  clean      Deletes all library files defined in libman.json from the project
  init       Create a new libman.json
  install    Add a library definition to the libman.json file, and download the
             library to the specified location
  restore    Downloads all files from provider and saves them to specified
             destination
  uninstall  Deletes all files for the specified library from their specified
             destination, then removes the specified library definition from
             libman.json
  update     Updates the specified library

Use "libman [command] --help" for more information about a command.
```

The following sections outline the available CLI commands.

Initialize LibMan in the project

The `libman init` command creates a *libman.json* file if one doesn't exist. The file is created with the default item template content.

Synopsis

```
libman init [-d|--default-destination] [-p|--default-provider] [--verbosity]
libman init [-h|--help]
```

Options

The following options are available for the `libman init` command:

- `-d|--default-destination <PATH>`

  A path relative to the current folder. Library files are installed in this location if no `destination` property is defined for a library in *libman.json*. The `<PATH>` value is written to the `defaultDestination` property of *libman.json*.

- `-p|--default-provider <PROVIDER>`

  The provider to use if no provider is defined for a given library. The `<PROVIDER>` value is written to the `defaultProvider` property of *libman.json*. Replace `<PROVIDER>` with one of the following values:

  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

- `quiet`
- `normal`
- `detailed`

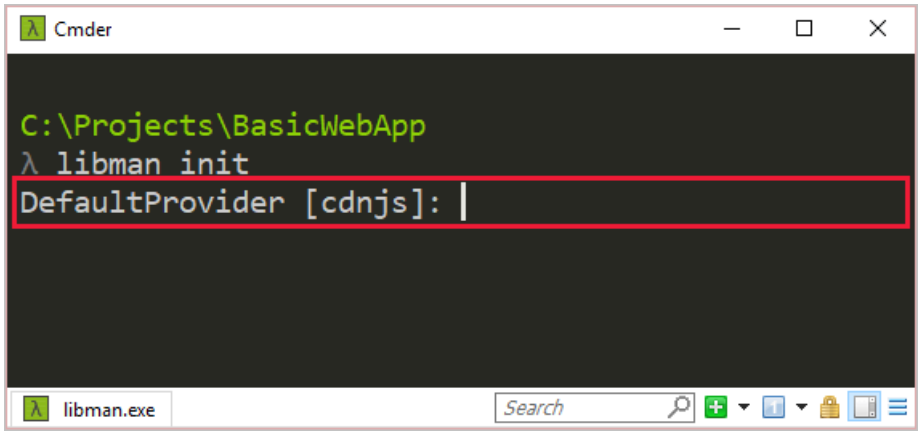
Examples

To create a *libman.json* file in an ASP.NET Core project:

- Navigate to the project root.
- Run the following command:

```
libman init
```

- Type the name of the default provider, or press `Enter` to use the default CDNJS provider. Valid values include:
  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`



A *libman.json* file is added to the project root with the following content:

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

Add library files

The `libman install` command downloads and installs library files into the project. A *libman.json* file is added if one doesn't exist. The *libman.json* file is modified to store configuration details for the library files.

Synopsis

```
libman install <LIBRARY> [-d|--destination] [--files] [-p|--provider] [--verbosity]
libman install [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to install. This name may include version number notation (for example, `@1.2.0`).

Options

The following options are available for the `libman install` command:

- `-d|--destination <PATH>`

  The location to install the library. If not specified, the default location is used. If no `defaultDestination` property is specified in *libman.json*, this option is required.

- `--files <FILE>`

  Specify the name of the file to install from the library. If not specified, all files from the library are installed. Provide one `--files` option per file to be installed. Relative paths are supported too. For example: `--files dist/browser/signalr.js`.

- `-p|--provider <PROVIDER>`

  The name of the provider to use for the library acquisition. Replace `<PROVIDER>` with one of the following values:

  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`

  If not specified, the `defaultProvider` property in *libman.json* is used. If no `defaultProvider` property is specified in *libman.json*, this option is required.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

  - `quiet`
  - `normal`
  - `detailed`

Examples

Consider the following *libman.json* file:

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

To install the jQuery version 3.2.1 *jquery.min.js* file to the *wwwroot/scripts/jquery* folder using the CDNJS provider:

```
libman install jquery@3.2.1 --provider cdnjs --destination wwwroot/scripts/jquery --files jquery.min.js
```

The *libman.json* file resembles the following:

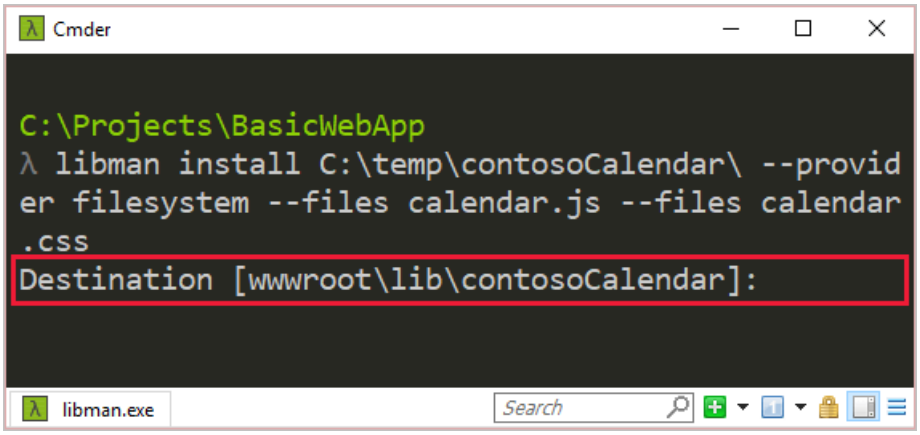
```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
        "jquery.min.js"
      ]
    }
  ]
}
```

To install the *calendar.js* and *calendar.css* files from *C:\temp\contosoCalendar* using the file system provider:

```
libman install C:\temp\contosoCalendar\ --provider filesystem --files calendar.js --files calendar.css
```

The following prompt appears for two reasons:

- The *libman.json* file doesn't contain a `defaultDestination` property.
- The `libman install` command doesn't contain the `-d|--destination` option.



After accepting the default destination, the *libman.json* file resembles the following:

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
        "jquery.min.js"
      ]
    },
    {
      "library": "C:\\temp\\contosoCalendar\\",
      "provider": "filesystem",
      "destination": "wwwroot/lib/contosoCalendar",
      "files": [
        "calendar.js",
        "calendar.css"
      ]
    }
  ]
}
```

Restore library files

The `libman restore` command installs library files defined in *libman.json*. The following rules apply:

- If no *libman.json* file exists in the project root, an error is returned.
- If a library specifies a provider, the `defaultProvider` property in *libman.json* is ignored.
- If a library specifies a destination, the `defaultDestination` property in *libman.json* is ignored.

Synopsis

```
libman restore [--verbosity]
libman restore [-h|--help]
```

Options

The following options are available for the `libman restore` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

  - `quiet`
  - `normal`
  - `detailed`

Examples

To restore the library files defined in *libman.json*:

```
libman restore
```

Delete library files

The `libman clean` command deletes library files previously restored via LibMan. Folders that become empty after this operation are deleted. The library files' associated configurations in the `libraries` property of *libman.json* aren't removed.

Synopsis

```
libman clean [--verbosity]
libman clean [-h|--help]
```

Options

The following options are available for the `libman clean` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

  - `quiet`

- `normal`
- `detailed`

Examples

To delete library files installed via LibMan:

```
libman clean
```

Uninstall library files

The `libman uninstall` command:

- Deletes all files associated with the specified library from the destination in *libman.json*.
- Removes the associated library configuration from *libman.json*.

An error occurs when:

- No *libman.json* file exists in the project root.
- The specified library doesn't exist.

If more than one library with the same name is installed, you're prompted to choose one.

Synopsis

```
libman uninstall <LIBRARY> [--verbosity]
libman uninstall [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to uninstall. This name may include version number notation (for example, `@1.2.0`).

Options

The following options are available for the `libman uninstall` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

  - `quiet`
  - `normal`
  - `detailed`

Examples

Consider the following *libman.json* file:

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

- To uninstall jQuery, either of the following commands succeed:

```
libman uninstall jquery
```

```
libman uninstall jquery@3.3.1
```

- To uninstall the Lodash files installed via the `filesystem` provider:

```
libman uninstall C:\temp\lodash\
```

Update library version

The `libman update` command updates a library installed via LibMan to the specified version.

An error occurs when:

- No *libman.json* file exists in the project root.
- The specified library doesn't exist.

If more than one library with the same name is installed, you're prompted to choose one.

Synopsis

```
libman update <LIBRARY> [-pre] [--to] [--verbosity]
libman update [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to update.

Options

The following options are available for the `libman update` command:

- `-pre`

  Obtain the latest prerelease version of the library.

- `--to <VERSION>`

  Obtain a specific version of the library.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

  - `quiet`
  - `normal`
  - `detailed`

Examples

- To update jQuery to the latest version:

  ```
  libman update jquery
  ```

- To update jQuery to version 3.3.1:

  ```
  libman update jquery --to 3.3.1
  ```

- To update jQuery to the latest prerelease version:

  ```
  libman update jquery -pre
  ```

Manage library cache

The `libman cache` command manages the LibMan library cache. The `filesystem` provider doesn't use the library cache.

Synopsis

```
libman cache clean [<PROVIDER>] [--verbosity]
libman cache list [--files] [--libraries] [--verbosity]
libman cache [-h|--help]
```

Arguments

`PROVIDER`

Only used with the `clean` command. Specifies the provider cache to clean. Valid values include:

- `cdnjs`
- `filesystem`
- `jsdelivr`
- `unpkg`

Options

The following options are available for the `libman cache` command:

- `--files`

  List the names of files that are cached.

- `--libraries`

  List the names of libraries that are cached.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

  - `quiet`
  - `normal`
  - `detailed`

Examples

- To view the names of cached libraries per provider, use one of the following commands:

  ```
  libman cache list
  ```

  ```
  libman cache list --libraries
  ```

  Output similar to the following is displayed:

  ```
  Cache contents:
  ---------------
  unpkg:
      knockout
      react
      vue
  cdnjs:
      font-awesome
      jquery
      knockout
      lodash.js
      react
  ```

- To view the names of cached library files per provider:

  ```
  libman cache list --files
  ```

  Output similar to the following is displayed:

```
Cache contents:
---------------
unpkg:
    knockout:
        <list omitted for brevity>
    react:
        <list omitted for brevity>
    vue:
        <list omitted for brevity>
cdnjs:
    font-awesome
        metadata.json
    jquery
        metadata.json
        3.2.1\core.js
        3.2.1\jquery.js
        3.2.1\jquery.min.js
        3.2.1\jquery.min.map
        3.2.1\jquery.slim.js
        3.2.1\jquery.slim.min.js
        3.2.1\jquery.slim.min.map
        3.3.1\core.js
        3.3.1\jquery.js
        3.3.1\jquery.min.js
        3.3.1\jquery.min.map
        3.3.1\jquery.slim.js
        3.3.1\jquery.slim.min.js
        3.3.1\jquery.slim.min.map
    knockout
        metadata.json
        3.4.2\knockout-debug.js
        3.4.2\knockout-min.js
    lodash.js
        metadata.json
        4.17.10\lodash.js
        4.17.10\lodash.min.js
    react
        metadata.json
```

Notice the preceding output shows that jQuery versions 3.2.1 and 3.3.1 are cached under the CDNJS provider.

- To empty the library cache for the CDNJS provider:

```
libman cache clean cdnjs
```

After emptying the CDNJS provider cache, the `libman cache list` command displays the following:

```
Cache contents:
---------------
unpkg:
    knockout
    react
    vue
cdnjs:
    (empty)
```

- To empty the cache for all supported providers:

```
libman cache clean
```

After emptying all provider caches, the `libman cache list` command displays the following:

```
Cache contents:
---------------
unpkg:
    (empty)
cdnjs:
    (empty)
```

Additional resources

- Install a Global Tool
- Use LibMan with ASP.NET Core in Visual Studio
- LibMan GitHub repository

Use LibMan with ASP.NET Core in Visual Studio

9/22/2020 • 7 minutes to read • Edit Online

By Scott Addie

Visual Studio has built-in support for LibMan in ASP.NET Core projects, including:

- Support for configuring and running LibMan restore operations on build.
- Menu items for triggering LibMan restore and clean operations.
- Search dialog for finding libraries and adding the files to a project.
- Editing support for *libman.json*—the LibMan manifest file.

View or download sample code (how to download)

Prerequisites

- Visual Studio 2019 with the **ASP.NET and web development** workload

Add library files

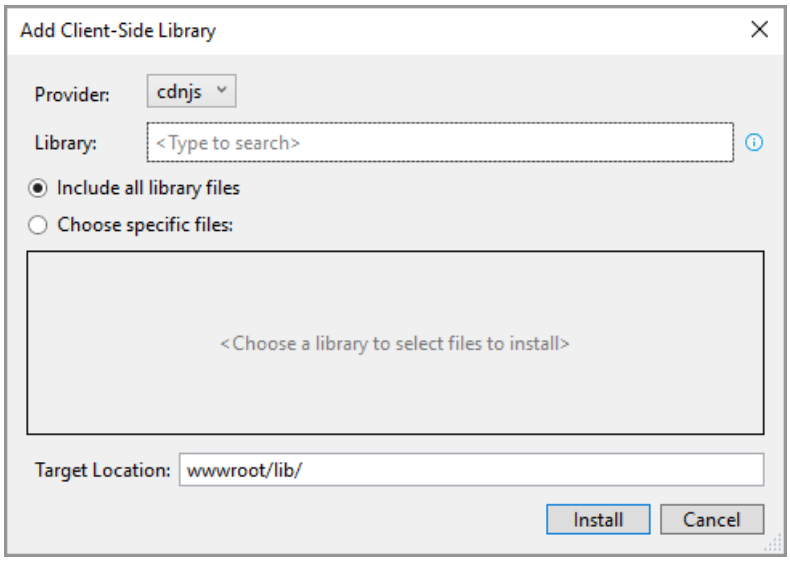
Library files can be added to an ASP.NET Core project in two different ways:

1. Use the Add Client-Side Library dialog
2. Manually configure LibMan manifest file entries

Use the Add Client-Side Library dialog

Follow these steps to install a client-side library:

- In **Solution Explorer**, right-click the project folder in which the files should be added. Choose **Add** > **Client-Side Library**. The **Add Client-Side Library** dialog appears:



- Select the library provider from the **Provider** drop down. CDNJS is the default provider.
- Type the library name to fetch in the **Library** text box. IntelliSense provides a list of libraries beginning with the provided text.
- Select the library from the IntelliSense list. Notice the library name is suffixed with the `@` symbol and the

latest stable version known to the selected provider.

- Decide which files to include:
  - Select the **Include all library files** radio button to include all of the library's files.
  - Select the **Choose specific files** radio button to include a subset of the library's files. When the radio button is selected, the file selector tree is enabled. Check the boxes to the left of the file names to download.
- Specify the project folder for storing the files in the **Target Location** text box. As a recommendation, store each library in a separate folder.

  The suggested **Target Location** folder is based on the location from which the dialog launched:

  - If launched from the project root:
    - *wwwroot/lib* is used if *wwwroot* exists.
    - *lib* is used if *wwwroot* doesn't exist.
  - If launched from a project folder, the corresponding folder name is used.

  The folder suggestion is suffixed with the library name. The following table illustrates folder suggestions when installing jQuery in a Razor Pages project.

LAUNCH LOCATION	SUGGESTED FOLDER
project root (if *wwwroot* exists)	*wwwroot/lib/jquery/*
project root (if *wwwroot* doesn't exist)	*lib/jquery/*
Pages folder in project	*Pages/jquery/*

- Click the **Install** button to download the files, per the configuration in *libman.json*.
- Review the **Library Manager** feed of the **Output** window for installation details. For example:

```
Restore operation started...
Restoring libraries for project LibManSample
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

Manually configure LibMan manifest file entries

All LibMan operations in Visual Studio are based on the content of the project root's LibMan manifest (*libman.json*). You can manually edit *libman.json* to configure library files for the project. Visual Studio restores all library files once *libman.json* is saved.

To open *libman.json* for editing, the following options exist:

- Double-click the *libman.json* file in **Solution Explorer**.
- Right-click the project in **Solution Explorer** and select **Manage Client-Side Libraries**. †
- Select **Manage Client-Side Libraries** from the Visual Studio **Project** menu. †

† If the *libman.json* file doesn't already exist in the project root, it will be created with the default item template content.

Visual Studio offers rich JSON editing support such as colorization, formatting, IntelliSense, and schema validation.

The LibMan manifest's JSON schema is found at https://json.schemastore.org/libman.

With the following manifest file, LibMan retrieves files per the configuration defined in the `libraries` property. An explanation of the object literals defined within `libraries` follows:

- A subset of jQuery version 3.3.1 is retrieved from the CDNJS provider. The subset is defined in the `files` property—*jquery.min.js*, *jquery.js*, and *jquery.min.map*. The files are placed in the project's *wwwroot/lib/jquery* folder.
- The entirety of Bootstrap version 4.1.3 is retrieved and placed in a *wwwroot/lib/bootstrap* folder. The object literal's `provider` property overrides the `defaultProvider` property value. LibMan retrieves the Bootstrap files from the unpkg provider.
- A subset of Lodash was approved by a governing body within the organization. The *lodash.js* and *lodash.min.js* files are retrieved from the local file system at *C:\temp\lodash*. The files are copied to the project's *wwwroot/lib/lodash* folder.

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

> **NOTE**
> LibMan only supports one version of each library from each provider. The *libman.json* file fails schema validation if it contains two libraries with the same library name for a given provider.

Restore library files

To restore library files from within Visual Studio, there must be a valid *libman.json* file in the project root. Restored files are placed in the project at the location specified for each library.

Library files can be restored in an ASP.NET Core project in two ways:

1. Restore files during build
2. Restore files manually

Restore files during build

LibMan can restore the defined library files as part of the build process. By default, the *restore-on-build* behavior is disabled.

To enable and test the restore-on-build behavior:

- Right-click *libman.json* in **Solution Explorer** and select **Enable Restore Client-Side Libraries on Build** from the context menu.
- Click the **Yes** button when prompted to install a NuGet package. The Microsoft.Web.LibraryManager.Build NuGet package is added to the project:

  ```
  <PackageReference Include="Microsoft.Web.LibraryManager.Build" Version="1.0.113" />
  ```

- Build the project to confirm LibMan file restoration occurs. The `Microsoft.Web.LibraryManager.Build` package injects an MSBuild target that runs LibMan during the project's build operation.
- Review the **Build** feed of the **Output** window for a LibMan activity log:

  ```
  1>------ Build started: Project: LibManSample, Configuration: Debug Any CPU ------
  1>
  1>Restore operation started...
  1>Restoring library jquery@3.3.1...
  1>Restoring library bootstrap@4.1.3...
  1>
  1>2 libraries restored in 10.66 seconds
  1>LibManSample -> C:\LibManSample\bin\Debug\netcoreapp2.1\LibManSample.dll
  ========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
  ```

When the restore-on-build behavior is enabled, the *libman.json* context menu displays a **Disable Restore Client-Side Libraries on Build** option. Selecting this option removes the `Microsoft.Web.LibraryManager.Build` package reference from the project file. Consequently, the client-side libraries are no longer restored on each build.

Regardless of the restore-on-build setting, you can manually restore at any time from the *libman.json* context menu. For more information, see Restore files manually.

Restore files manually

To manually restore library files:

- For all projects in the solution:
  - Right-click the solution name in **Solution Explorer**.
  - Select the **Restore Client-Side Libraries** option.
- For a specific project:
  - Right-click the *libman.json* file in **Solution Explorer**.
  - Select the **Restore Client-Side Libraries** option.

While the restore operation is running:

- The Task Status Center (TSC) icon on the Visual Studio status bar will be animated and will read *Restore operation started*. Clicking the icon opens a tooltip listing the known background tasks.
- Messages will be sent to the status bar and the **Library Manager** feed of the **Output** window. For example:

```
Restore operation started...
Restoring libraries for project LibManSample
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

Delete library files

To perform the *clean* operation, which deletes library files previously restored in Visual Studio:

- Right-click the *libman.json* file in **Solution Explorer**.
- Select the **Clean Client-Side Libraries** option.

To prevent unintentional removal of non-library files, the clean operation doesn't delete whole directories. It only removes files that were included in the previous restore.

While the clean operation is running:

- The TSC icon on the Visual Studio status bar will be animated and will read *Client libraries operation started*. Clicking the icon opens a tooltip listing the known background tasks.
- Messages are sent to the status bar and the **Library Manager** feed of the **Output** window. For example:

```
Clean libraries operation started...
Clean libraries operation completed
2 libraries were successfully deleted in 1.91 secs
```

The clean operation only deletes files from the project. Library files stay in the cache for faster retrieval on future restore operations. To manage library files stored in the local machine's cache, use the LibMan CLI.

Uninstall library files

To uninstall library files:

- Open *libman.json*.
- Position the caret inside the corresponding `libraries` object literal.
- Click the light bulb icon that appears in the left margin, and select **Uninstall <library_name>@<library_version>**:

Alternatively, you can manually edit and save the LibMan manifest (*libman.json*). The restore operation runs when the file is saved. Library files that are no longer defined in *libman.json* are removed from the project.

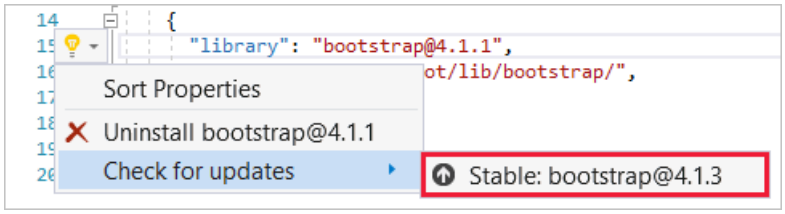
Update library version

To check for an updated library version:

- Open *libman.json*.
- Position the caret inside the corresponding `libraries` object literal.
- Click the light bulb icon that appears in the left margin. Hover over **Check for updates**.

LibMan checks for a library version newer than the version installed. The following outcomes can occur:

- A **No updates found** message is displayed if the latest version is already installed.
- The latest stable version is displayed if not already installed.



- If a pre-release newer than the installed version is available, the pre-release is displayed.

To downgrade to an older library version, manually edit the *libman.json* file. When the file is saved, the LibMan restore operation:

- Removes redundant files from the previous version.
- Adds new and updated files from the new version.

Additional resources

- Use the LibMan CLI with ASP.NET Core
- LibMan GitHub repository

Use Grunt in ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

Grunt is a JavaScript task runner that automates script minification, TypeScript compilation, code quality "lint" tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is fully supported in Visual Studio.

This example uses an empty ASP.NET Core project as its starting point, to show how to automate the client build process from scratch.

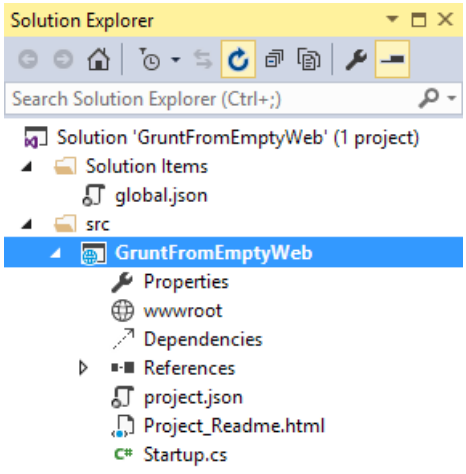
The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

- **grunt**: The Grunt task runner package.
- **grunt-contrib-clean**: A plugin that removes files or directories.
- **grunt-contrib-jshint**: A plugin that reviews JavaScript code quality.
- **grunt-contrib-concat**: A plugin that joins files into a single file.
- **grunt-contrib-uglify**: A plugin that minifies JavaScript to reduce size.
- **grunt-contrib-watch**: A plugin that watches file activity.

Preparing the application

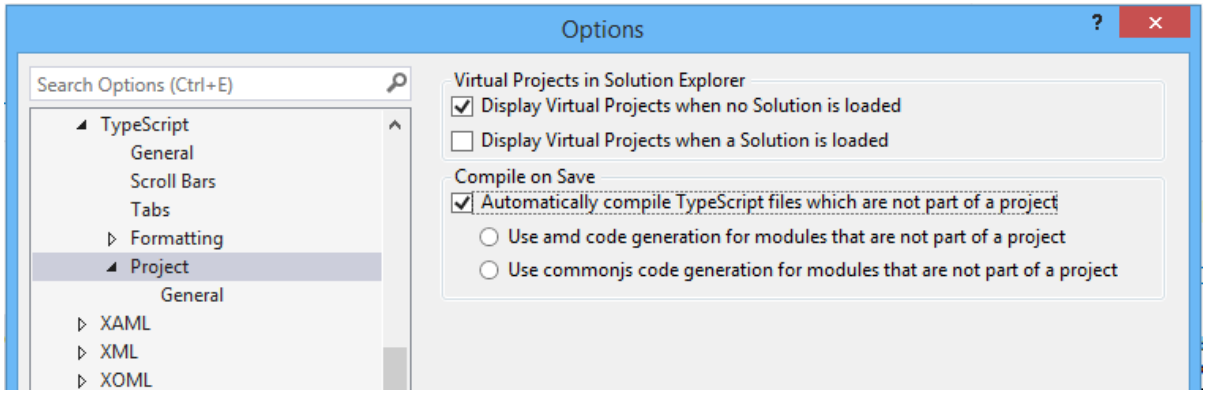
To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio settings and will be our raw material to process using Grunt.

1. In Visual Studio, create a new `ASP.NET Web Application`.
2. In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template and click the OK button.
3. In the Solution Explorer, review the project structure. The `\src` folder includes empty `wwwroot` and `Dependencies` nodes.



4. Add a new folder named `TypeScript` to your project directory.

5. Before adding any files, make sure that Visual Studio has the option 'compile on save' for TypeScript files checked. Navigate to **Tools** > **Options** > **Text Editor** > **Typescript** > **Project**:



6. Right-click the `TypeScript` directory and select **Add** > **New Item** from the context menu. Select the **JavaScript file** item and name the file *Tastes.ts* (note the *.ts extension). Copy the line of TypeScript code below into the file (when you save, a new *Tastes.js* file will appear with the JavaScript source).

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. Add a second file to the **TypeScript** directory and name it `Food.ts`. Copy the code below into the file.

```
class Food {
  constructor(name: string, calories: number) {
    this._name = name;
    this._calories = calories;
  }

  private _name: string;
  get Name() {
    return this._name;
  }

  private _calories: number;
  get Calories() {
    return this._calories;
  }

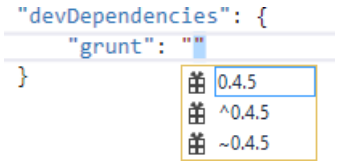
  private _taste: Tastes;
  get Taste(): Tastes { return this._taste }
  set Taste(value: Tastes) {
    this._taste = value;
  }
}
```

Configuring NPM

Next, configure NPM to download grunt and grunt-tasks.

1. In the Solution Explorer, right-click the project and select **Add** > **New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, *package.json*, and click the **Add** button.

2. In the *package.json* file, inside the `devDependencies` object braces, enter "grunt". Select `grunt` from the Intellisense list and press the Enter key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press `Ctrl-Space` if Intellisense doesn't appear).



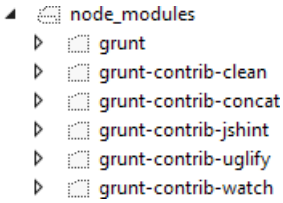
> **NOTE**
>
> NPM uses semantic versioning to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme <major>.<minor>.<patch>. Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (0.4.5 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version. See the NPM semver version parser reference as a guide to the full expressivity that SemVer provides.

3. Add more dependencies to load grunt-contrib-* packages for *clean*, *jshint*, *concat*, *uglify*, and *watch* as shown in the example below. The versions don't need to match the example.

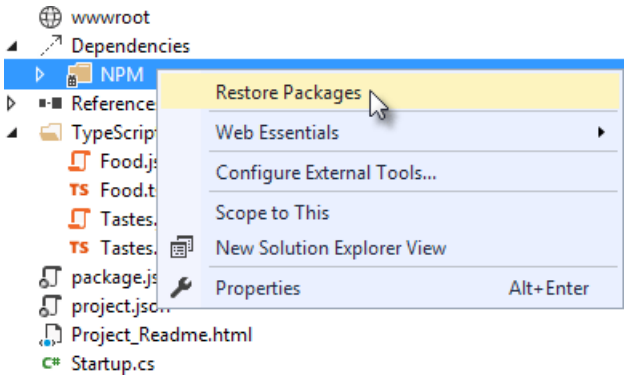
```
"devDependencies": {
  "grunt": "0.4.5",
  "grunt-contrib-clean": "0.6.0",
  "grunt-contrib-jshint": "0.11.0",
  "grunt-contrib-concat": "0.5.1",
  "grunt-contrib-uglify": "0.8.0",
  "grunt-contrib-watch": "0.6.1"
}
```

4. Save the *package.json* file.

The packages for each `devDependencies` item will download, along with any files that each package requires. You can find the package files in the *node_modules* directory by enabling the **Show All Files** button in **Solution Explorer**.



> **NOTE**
>
> If you need to, you can manually restore dependencies in **Solution Explorer** by right-clicking on `Dependencies\NPM` and selecting the **Restore Packages** menu option.



Configuring Grunt

Grunt is configured using a manifest named *Gruntfile.js* that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add** > **New Item**. Select the **JavaScript File** item template, change the name to *Gruntfile.js*, and click the **Add** button.

2. Add the following code to *Gruntfile.js*. The `initConfig` function sets options for each package, and the remainder of the module loads and register tasks.

```
module.exports = function (grunt) {
  grunt.initConfig({
  });
};
```

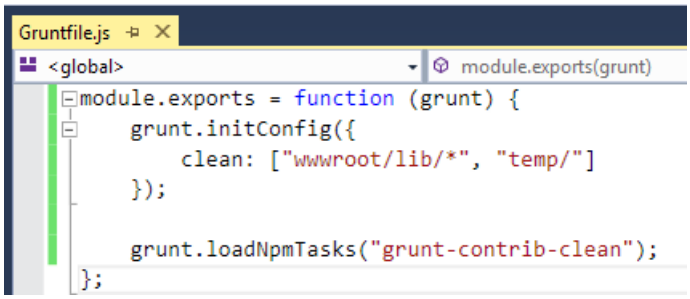
3. Inside the `initConfig` function, add options for the `clean` task as shown in the example *Gruntfile.js* below. The `clean` task accepts an array of directory strings. This task removes files from *wwwroot/lib* and removes the entire */temp* directory.

```
module.exports = function (grunt) {
  grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"],
  });
};
```

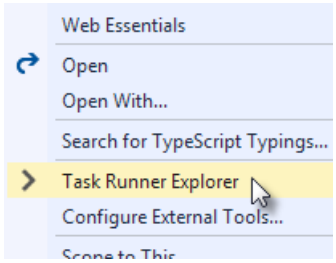
4. Below the `initConfig` function, add a call to `grunt.loadNpmTasks`. This will make the task runnable from Visual Studio.

```
grunt.loadNpmTasks("grunt-contrib-clean");
```

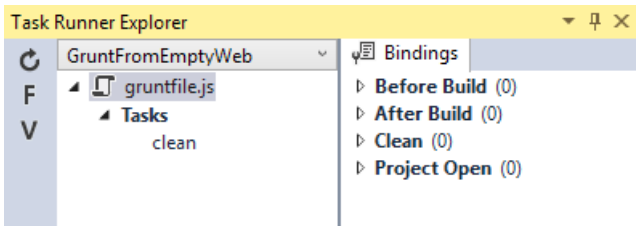
5. Save *Gruntfile.js*. The file should look something like the screenshot below.



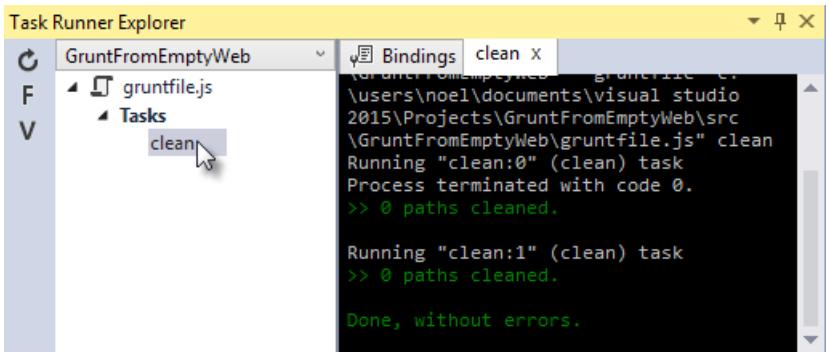
6. Right-click *Gruntfile.js* and select **Task Runner Explorer** from the context menu. The **Task Runner Explorer** window will open.



7. Verify that `clean` shows under **Tasks** in the **Task Runner Explorer**.



8. Right-click the clean task and select **Run** from the context menu. A command window displays progress of the task.



> **NOTE**
>
> There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

9. In the `initConfig` function, add an entry for `concat` using the code below.

   The `src` property array lists files to combine, in the order that they should be combined. The `dest` property assigns the path to the combined file that's produced.

```
concat: {
  all: {
    src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
    dest: 'temp/combined.js'
  }
},
```

> **NOTE**
>
> The `all` property in the code above is the name of a target. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using IntelliSense or assign your own.

10. Add the `jshint` task using the code below.

    The jshint `code-quality` utility is run against every JavaScript file found in the *temp* directory.

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

> **NOTE**
>
> The option "-W069" is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, i.e. `Tastes["Sweet"]` instead of `Tastes.Sweet`. The option turns off the warning to allow the rest of the process to continue.

11. Add the `uglify` task using the code below.

    The task minifies the *combined.js* file found in the temp directory and creates the result file in wwwroot/lib following the standard naming convention *<file name>.min.js*.

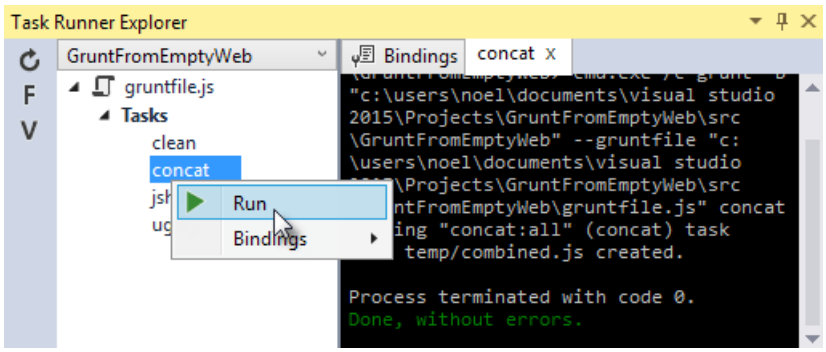
```
uglify: {
 all: {
   src: ['temp/combined.js'],
   dest: 'wwwroot/lib/combined.min.js'
 }
},
```

12. Under the call to `grunt.loadNpmTasks` that loads `grunt-contrib-clean`, include the same call for jshint, concat, and uglify using the code below.

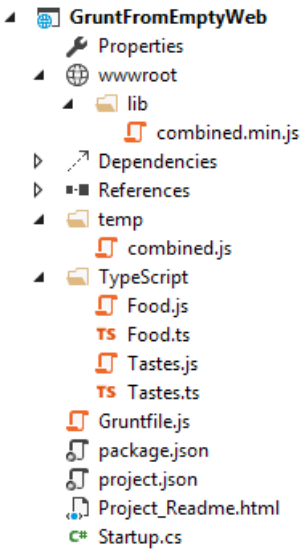
```
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
```

13. Save *Gruntfile.js*. The file should look something like the example below.

14. Notice that the **Task Runner Explorer** Tasks list includes `clean`, `concat`, `jshint` and `uglify` tasks. Run each task in order and observe the results in **Solution Explorer**. Each task should run without errors.



The concat task creates a new *combined.js* file and places it into the temp directory. The `jshint` task simply runs and doesn't produce output. The `uglify` task creates a new *combined.min.js* file and places it into *wwwroot/lib*. On completion, the solution should look something like the screenshot below:



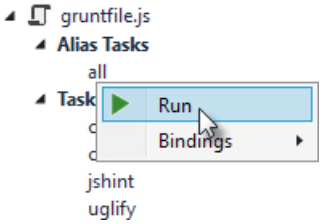
> **NOTE**
>
> For more information on the options for each package, visit https://www.npmjs.com/ and lookup the package name in the search box on the main page. For example, you can look up the grunt-contrib-clean package to get a documentation link that explains all of its parameters.

All together now

Use the Grunt `registerTask()` method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean -> concat -> jshint -> uglify, add the code below to the module. The code should be added to the same level as the loadNpmTasks() calls, outside initConfig.

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The `all` task will run `clean`, `concat`, `jshint` and `uglify`, in order.



Watching for changes

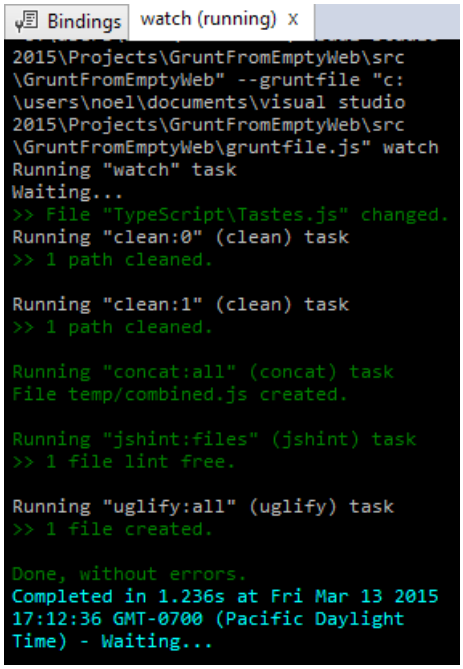
A `watch` task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to initConfig to watch for changes to *.js files in the TypeScript directory. If a JavaScript file is changed, `watch` will run the `all` task.

```
watch: {
  files: ["TypeScript/*.js"],
  tasks: ["all"]
}
```

Add a call to `loadNpmTasks()` to show the `watch` task in Task Runner Explorer.

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

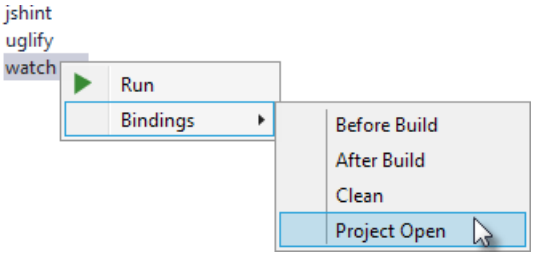
Right-click the watch task in Task Runner Explorer and select Run from the context menu. The command window that shows the watch task running will display a "Waiting..." message. Open one of the TypeScript files, add a space, and then save the file. This will trigger the watch task and trigger the other tasks to run in order. The screenshot below shows a sample run.



Binding to Visual Studio events

Unless you want to manually start your tasks every time you work in Visual Studio, bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Bind `watch` so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the watch task and select **Bindings** > **Project Open** from the context menu.



Unload and reload the project. When the project loads again, the watch task starts running automatically.

Summary

Grunt is a powerful task runner that can be used to automate most client-build tasks. Grunt leverages NPM to deliver its packages, and features tooling integration with Visual Studio. Visual Studio's Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

Bundle and minify static assets in ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

By Scott Addie and David Pine

This article explains the benefits of applying bundling and minification, including how these features can be used with ASP.NET Core web apps.

What is bundling and minification

Bundling and minification are two distinct performance optimizations you can apply in a web app. Used together, bundling and minification improve performance by reducing the number of server requests and reducing the size of the requested static assets.

Bundling and minification primarily improve the first page request load time. Once a web page has been requested, the browser caches the static assets (JavaScript, CSS, and images). Consequently, bundling and minification don't improve performance when requesting the same page, or pages, on the same site requesting the same assets. If the expires header isn't set correctly on the assets and if bundling and minification isn't used, the browser's freshness heuristics mark the assets stale after a few days. Additionally, the browser requires a validation request for each asset. In this case, bundling and minification provide a performance improvement even after the first page request.

Bundling

Bundling combines multiple files into a single file. Bundling reduces the number of server requests that are necessary to render a web asset, such as a web page. You can create any number of individual bundles specifically for CSS, JavaScript, etc. Fewer files means fewer HTTP requests from the browser to the server or from the service providing your application. This results in improved first page load performance.

Minification

Minification removes unnecessary characters from code without altering functionality. The result is a significant size reduction in requested assets (such as CSS, images, and JavaScript files). Common side effects of minification include shortening variable names to one character and removing comments and unnecessary whitespace.

Consider the following JavaScript function:

```
AddAltToImg = function (imageTagAndImageID, imageContext) {
    ///<signature>
    ///<summary> Adds an alt tab to the image
    // </summary>
    //<param name="imgElement" type="String">The image selector.</param>
    //<param name="ContextForImage" type="String">The image context.</param>
    ///</signature>
    var imageElement = $(imageTagAndImageID, imageContext);
    imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

Minification reduces the function to the following:

```
AddAltToImg=function(t,a){var r=$(t,a);r.attr("alt",r.attr("id").replace(/ID/,""))};
```

In addition to removing the comments and unnecessary whitespace, the following parameter and variable names were renamed as follows:

ORIGINAL	RENAMED
`imageTagAndImageID`	`t`
`imageContext`	`a`
`imageElement`	`r`

Impact of bundling and minification

The following table outlines differences between individually loading assets and using bundling and minification:

ACTION	WITH B/M	WITHOUT B/M	CHANGE
File Requests	7	18	157%
KB Transferred	156	264.68	70%
Load Time (ms)	885	2360	167%

Browsers are fairly verbose with regard to HTTP request headers. The total bytes sent metric saw a significant reduction when bundling. The load time shows a significant improvement, however this example ran locally. Greater performance gains are realized when using bundling and minification with assets transferred over a network.

Choose a bundling and minification strategy

The MVC and Razor Pages project templates provide a solution for bundling and minification consisting of a JSON configuration file. Third-party tools, such as the Grunt task runner, accomplish the same tasks with a bit more complexity. A third-party tool is a great fit when your development workflow requires processing beyond bundling and minification—such as linting and image optimization. By using design-time bundling and minification, the minified files are created prior to the app's deployment. Bundling and minifying before deployment provides the advantage of reduced server load. However, it's important to recognize that design-time bundling and minification increases build complexity and only works with static files.

Configure bundling and minification

> **NOTE**
> The BuildBundlerMinifier NuGet package needs to be added to your project for this to work.

In ASP.NET Core 2.0 or earlier, the MVC and Razor Pages project templates provide a *bundleconfig.json* configuration file that defines the options for each bundle:

In ASP.NET Core 2.1 or later, add a new JSON file, named *bundleconfig.json*, to the MVC or Razor Pages project root. Include the following JSON in that file as a starting point:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

The *bundleconfig.json* file defines the options for each bundle. In the preceding example, a single bundle configuration is defined for the custom JavaScript (*wwwroot/js/site.js*) and stylesheet (*wwwroot/css/site.css*) files.

Configuration options include:

- `outputFileName`: The name of the bundle file to output. Can contain a relative path from the *bundleconfig.json* file. **required**
- `inputFiles`: An array of files to bundle together. These are relative paths to the configuration file. **optional**, *an empty value results in an empty output file. globbing patterns are supported.
- `minify`: The minification options for the output type. **optional**, *default* - `minify: { enabled: true }`
  - Configuration options are available per output file type.
    - CSS Minifier
    - JavaScript Minifier
    - HTML Minifier
- `includeInProject`: Flag indicating whether to add generated files to project file. **optional**, *default - false*
- `sourceMap`: Flag indicating whether to generate a source map for the bundled file. **optional**, *default - false*
- `sourceMapRootPath`: The root path for storing the generated source map file.

Add files to workflow

Consider an example in which an additional *custom.css* file is added resembling the following:

```
.about, [role=main], [role=complementary] {
    margin-top: 60px;
}

footer {
    margin-top: 10px;
}
```

To minify *custom.css* and bundle it with *site.css* into a *site.min.css* file, add the relative path to *bundleconfig.json*:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css",
      "wwwroot/css/custom.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

NOTE

Alternatively, the following globbing pattern could be used:

```
"inputFiles": ["wwwroot/**/!(*.min).css" ]
```

This globbing pattern matches all CSS files and excludes the minified file pattern.

Build the application. Open *site.min.css* and notice the content of *custom.css* is appended to the end of the file.

Environment-based bundling and minification

As a best practice, the bundled and minified files of your app should be used in a production environment. During development, the original files make for easier debugging of the app.

Specify which files to include in your pages by using the Environment Tag Helper in your views. The Environment Tag Helper only renders its contents when running in specific environments.

The following `environment` tag renders the unprocessed CSS files when running in the `Development` environment:

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

```
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

The following `environment` tag renders the bundled and minified CSS files when running in an environment other than `Development`. For example, running in `Production` or `Staging` triggers the rendering of these stylesheets:

```
<environment exclude="Development">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

```
<environment names="Staging,Production">
    <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
</environment>
```

Consume bundleconfig.json from Gulp

There are cases in which an app's bundling and minification workflow requires additional processing. Examples include image optimization, cache busting, and CDN asset processing. To satisfy these requirements, you can convert the bundling and minification workflow to use Gulp.

Manually convert the bundling and minification workflow to use Gulp

Add a *package.json* file, with the following `devDependencies`, to the project root:

> **WARNING**
> The `gulp-uglify` module doesn't support ECMAScript (ES) 2015 / ES6 and later. Install gulp-terser instead of `gulp-uglify` to use ES2015 / ES6 or later.

```
"devDependencies": {
  "del": "^3.0.0",
  "gulp": "^4.0.0",
  "gulp-concat": "^2.6.1",
  "gulp-cssmin": "^0.2.0",
  "gulp-htmlmin": "^3.0.0",
  "gulp-uglify": "^3.0.0",
  "merge-stream": "^1.0.1"
}
```

Install the dependencies by running the following command at the same level as *package.json*:

```
npm i
```

Install the Gulp CLI as a global dependency:

```
npm i -g gulp-cli
```

Copy the *gulpfile.js* file below to the project root:

```
'use strict';

var gulp = require('gulp'),
    concat = require('gulp-concat'),
```

```
    cssmin = require('gulp-cssmin'),
    htmlmin = require('gulp-htmlmin'),
    uglify = require('gulp-uglify'),
    merge = require('merge-stream'),
    del = require('del'),
    bundleconfig = require('./bundleconfig.json');

const regex = {
    css: /\.css$/,
    html: /\.(html|htm)$/,
    js: /\.js$/
};

gulp.task('min:js', async function () {
    merge(getBundles(regex.js).map(bundle => {
        return gulp.src(bundle.inputFiles, { base: '.' })
            .pipe(concat(bundle.outputFileName))
            .pipe(uglify())
            .pipe(gulp.dest('.'));
    }))
});

gulp.task('min:css', async function () {
    merge(getBundles(regex.css).map(bundle => {
        return gulp.src(bundle.inputFiles, { base: '.' })
            .pipe(concat(bundle.outputFileName))
            .pipe(cssmin())
            .pipe(gulp.dest('.'));
    }))
});

gulp.task('min:html', async function () {
    merge(getBundles(regex.html).map(bundle => {
        return gulp.src(bundle.inputFiles, { base: '.' })
            .pipe(concat(bundle.outputFileName))
            .pipe(htmlmin({ collapseWhitespace: true, minifyCSS: true, minifyJS: true }))
            .pipe(gulp.dest('.'));
    }))
});

gulp.task('min', gulp.series(['min:js', 'min:css', 'min:html']));

gulp.task('clean', () => {
    return del(bundleconfig.map(bundle => bundle.outputFileName));
});

gulp.task('watch', () => {
    getBundles(regex.js).forEach(
        bundle => gulp.watch(bundle.inputFiles, gulp.series(["min:js"])));

    getBundles(regex.css).forEach(
        bundle => gulp.watch(bundle.inputFiles, gulp.series(["min:css"])));

    getBundles(regex.html).forEach(
        bundle => gulp.watch(bundle.inputFiles, gulp.series(['min:html'])));
});

const getBundles = (regexPattern) => {
    return bundleconfig.filter(bundle => {
        return regexPattern.test(bundle.outputFileName);
    });
};

gulp.task('default', gulp.series("min"));
```

Run Gulp tasks

To trigger the Gulp minification task before the project builds in Visual Studio:

1. Install the BuildBundlerMinifier NuGet package.
2. Add the following MSBuild Target to the project file:

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="gulp min" />
</Target>
```

In this example, any tasks defined within the `MyPreCompileTarget` target run before the predefined `Build` target. Output similar to the following appears in Visual Studio's Output window:

```
1>------ Build started: Project: BuildBundlerMinifierApp, Configuration: Debug Any CPU ------
1>BuildBundlerMinifierApp -> C:\BuildBundlerMinifierApp\bin\Debug\netcoreapp2.0\BuildBundlerMinifierApp.dll
1>[14:17:49] Using gulpfile C:\BuildBundlerMinifierApp\gulpfile.js
1>[14:17:49] Starting 'min:js'...
1>[14:17:49] Starting 'min:css'...
1>[14:17:49] Starting 'min:html'...
1>[14:17:49] Finished 'min:js' after 83 ms
1>[14:17:49] Finished 'min:css' after 88 ms
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Additional resources

- Use Grunt
- Use multiple environments
- Tag Helpers

Browser Link in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Nicolò Carandini, Mike Wasson, and Tom Dykstra

Browser Link is a Visual Studio feature. It creates a communication channel between the development environment and one or more web browsers. You can use Browser Link to refresh your web app in several browsers at once, which is useful for cross-browser testing.

Browser Link setup

Add the Microsoft.VisualStudio.Web.BrowserLink package to your project. For ASP.NET Core Razor Pages or MVC projects, also enable runtime compilation of Razor (*.cshtml*) files as described in Razor file compilation in ASP.NET Core. Razor syntax changes are applied only when runtime compilation has been enabled.

When converting an ASP.NET Core 2.0 project to ASP.NET Core 2.1 and transitioning to the Microsoft.AspNetCore.App metapackage, install the Microsoft.VisualStudio.Web.BrowserLink package for Browser Link functionality. The ASP.NET Core 2.1 project templates use the `Microsoft.AspNetCore.App` metapackage by default.

The ASP.NET Core 2.0 **Web Application**, **Empty**, and **Web API** project templates use the Microsoft.AspNetCore.All metapackage, which contains a package reference for Microsoft.VisualStudio.Web.BrowserLink. Therefore, using the `Microsoft.AspNetCore.All` metapackage requires no further action to make Browser Link available for use.

The ASP.NET Core 1.x **Web Application** project template has a package reference for the Microsoft.VisualStudio.Web.BrowserLink package. Other project types require you to add a package reference to `Microsoft.VisualStudio.Web.BrowserLink`.

Configuration

Call `UseBrowserLink` in the `Startup.Configure` method:

```
app.UseBrowserLink();
```

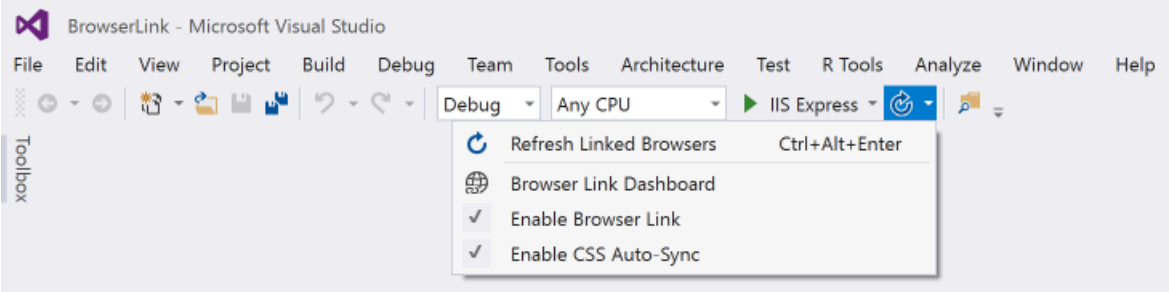
The `UseBrowserLink` call is typically placed inside an `if` block that only enables Browser Link in the Development environment. For example:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseBrowserLink();
}
```

For more information, see Use multiple environments in ASP.NET Core.

How to use Browser Link

When you have an ASP.NET Core project open, Visual Studio shows the Browser Link toolbar control next to the **Debug Target** toolbar control:

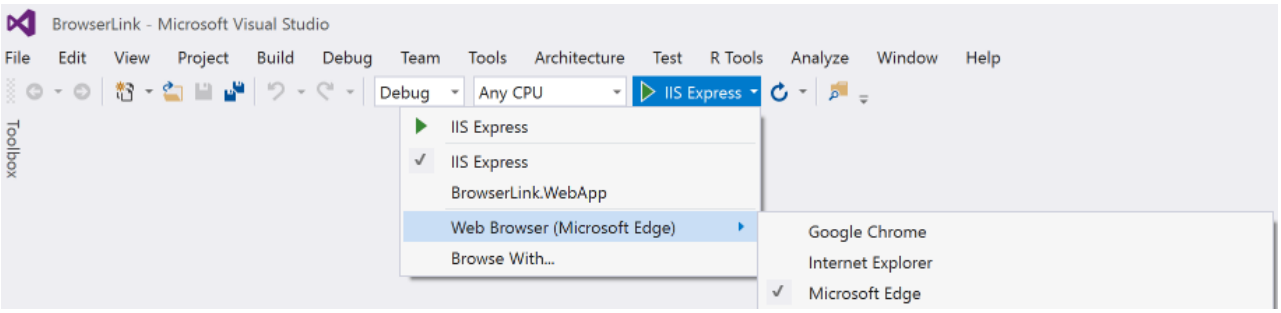


From the Browser Link toolbar control, you can:

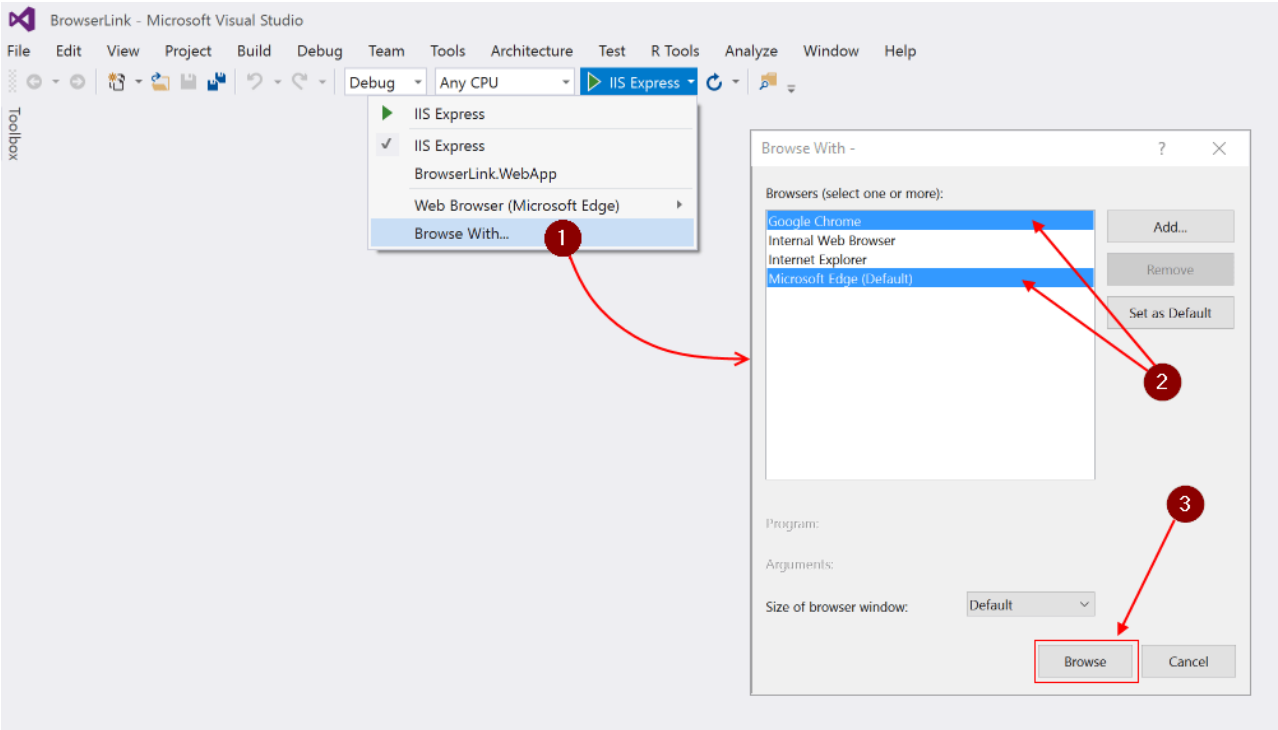
- Refresh the web app in several browsers at once.
- Open the **Browser Link Dashboard**.
- Enable or disable **Browser Link**. Note: Browser Link is disabled by default in Visual Studio.
- Enable or disable CSS Auto-Sync.

Refresh the web app in several browsers at once

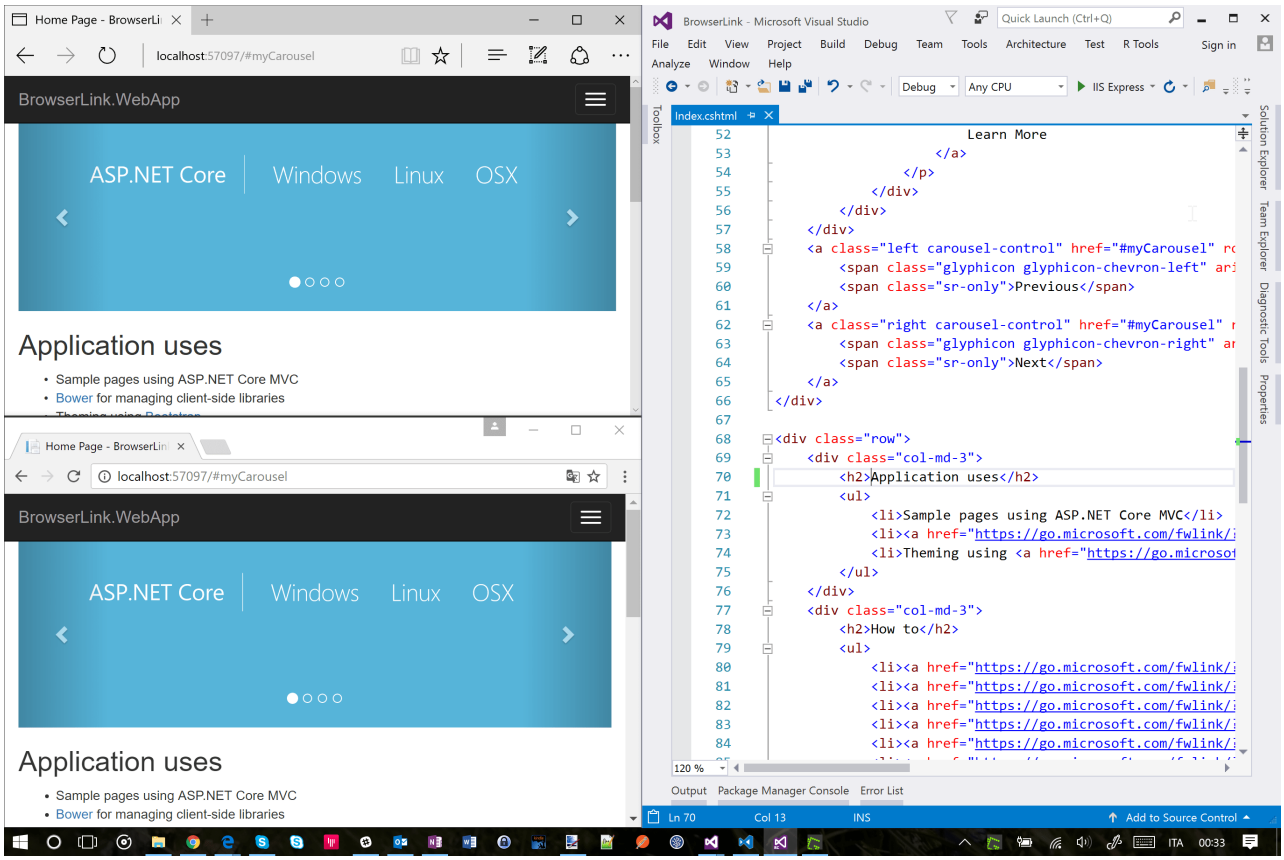
To choose a single web browser to launch when starting the project, use the drop-down menu in the **Debug Target** toolbar control:



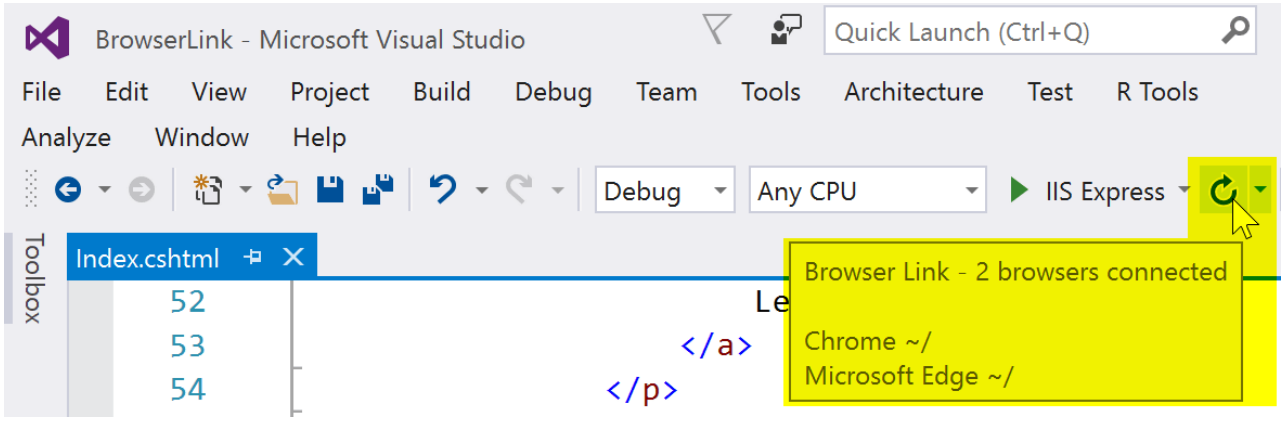
To open multiple browsers at once, choose **Browse with...** from the same drop-down. Hold down the `Ctrl` key to select the browsers you want, and then click **Browse**:



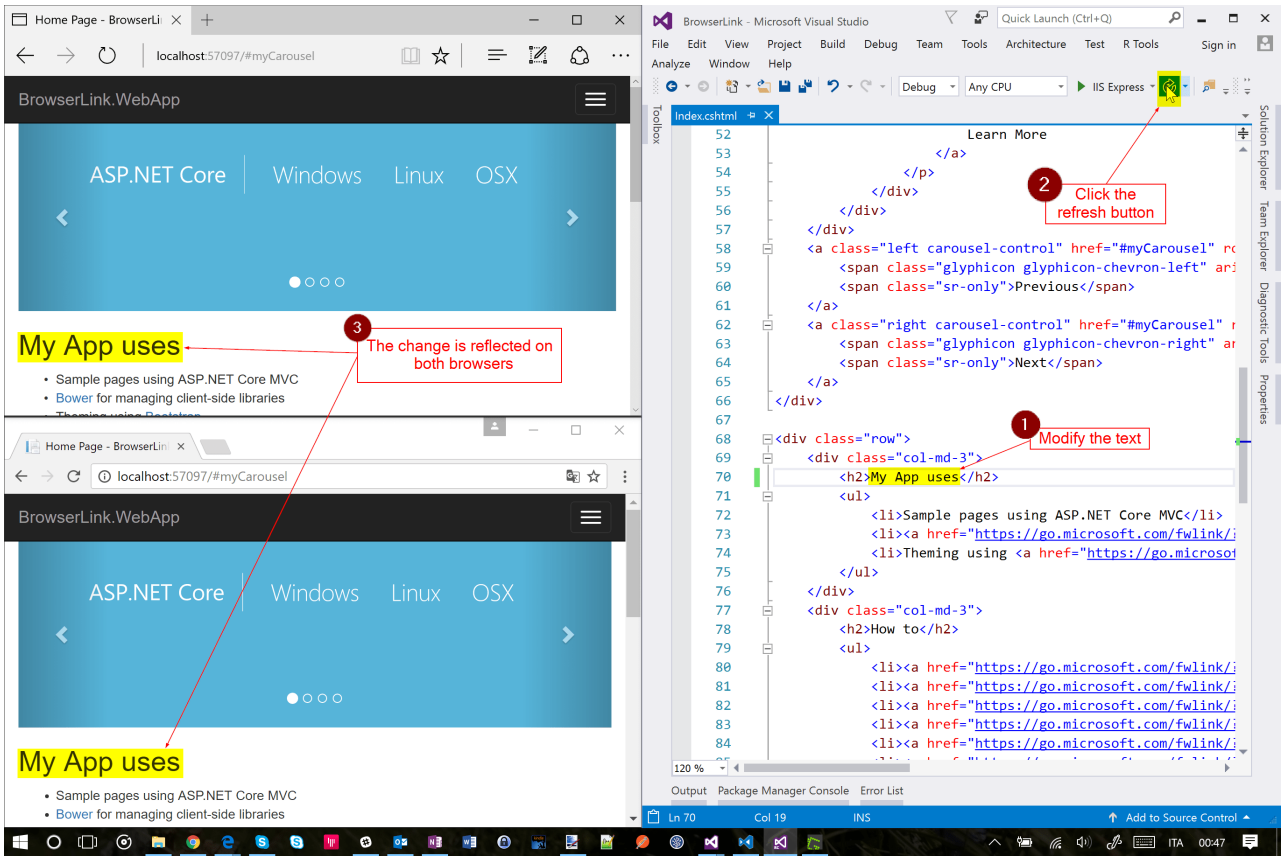
The following screenshot shows Visual Studio with the Index view open and two open browsers:



Hover over the Browser Link toolbar control to see the browsers that are connected to the project:



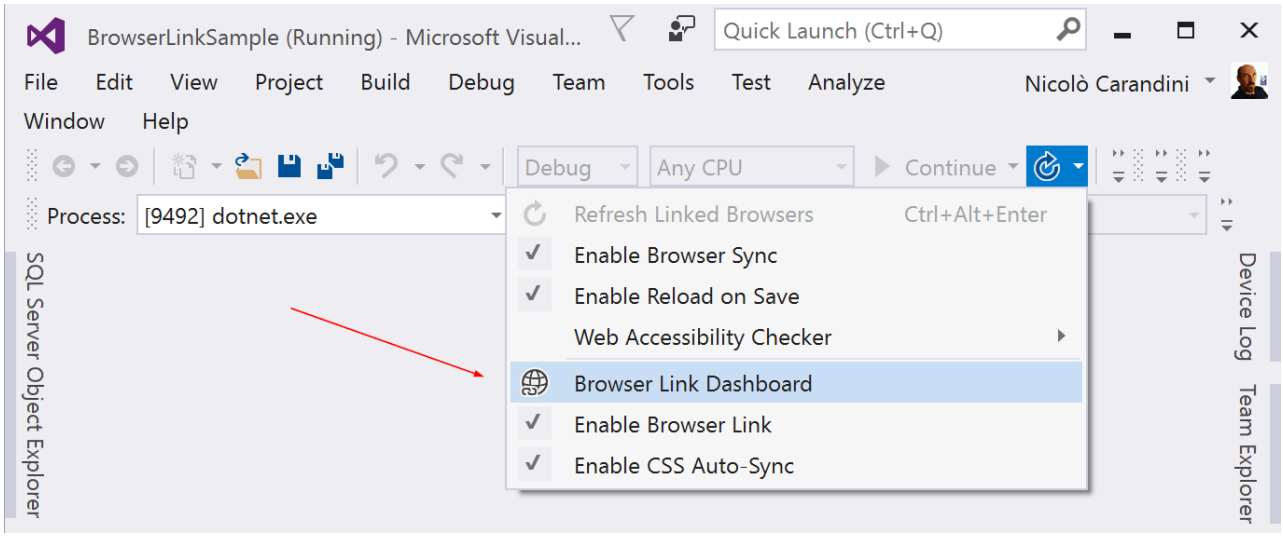
Change the Index view, and all connected browsers are updated when you click the Browser Link refresh button:



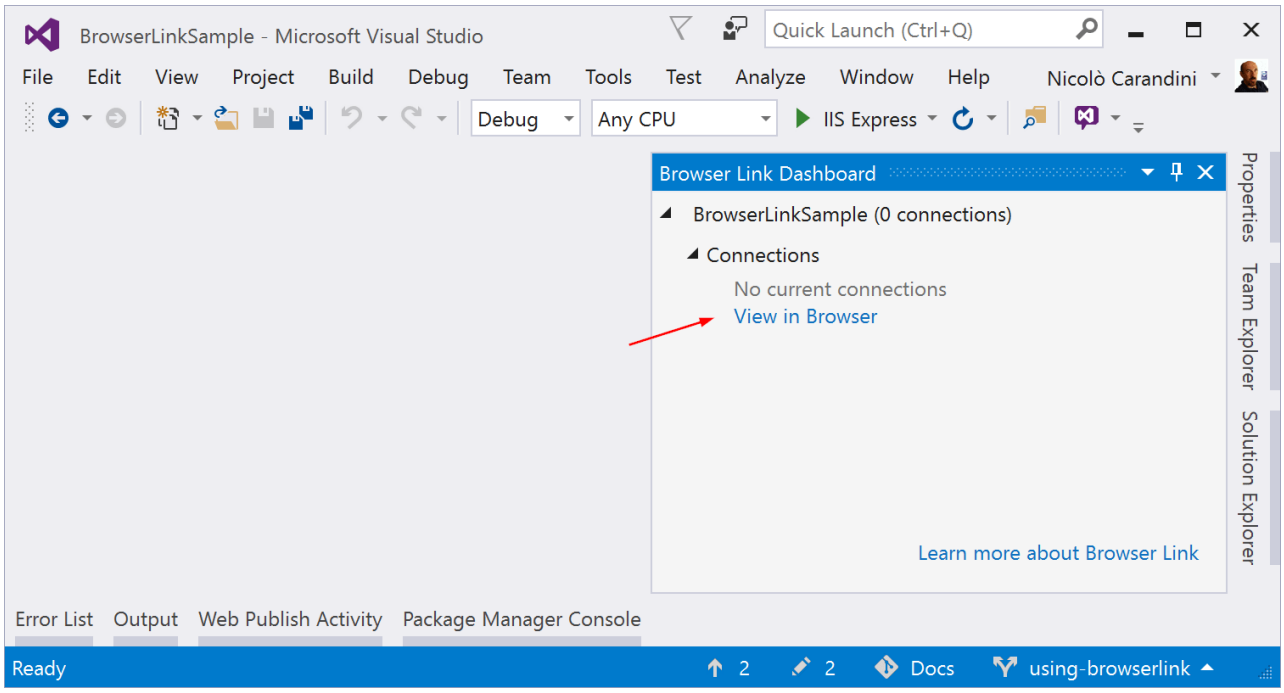
Browser Link also works with browsers that you launch from outside Visual Studio and navigate to the app URL.

The Browser Link Dashboard

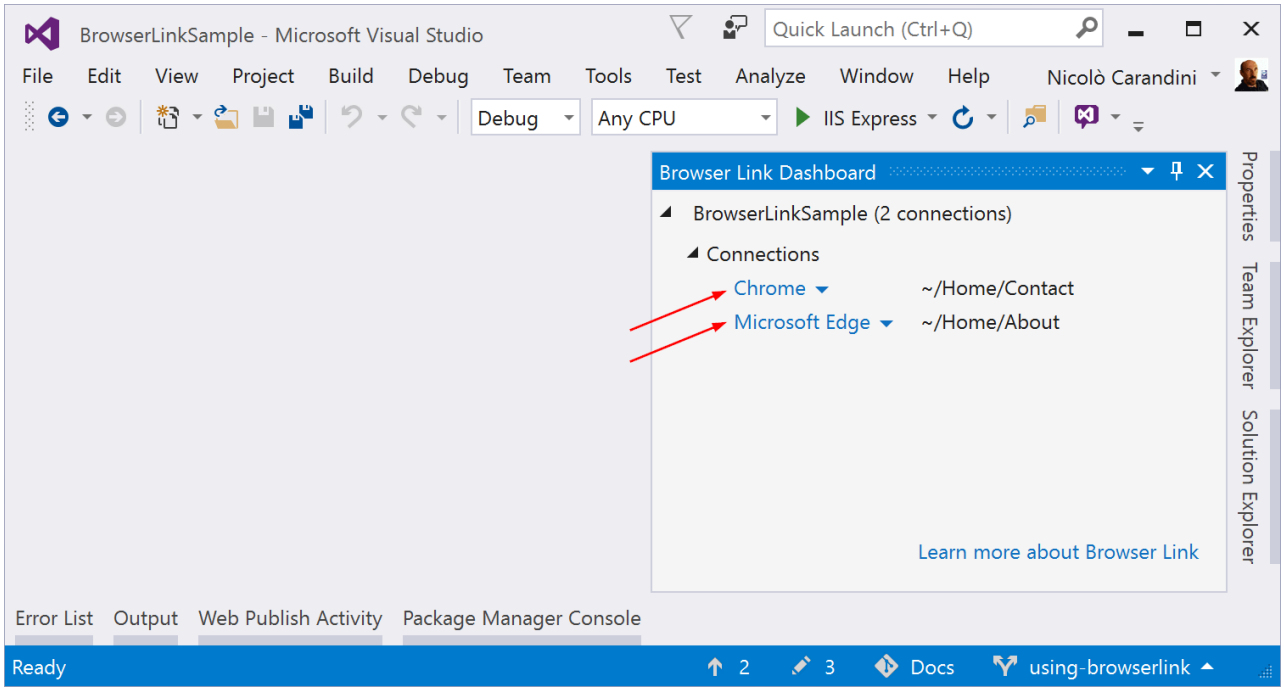
Open the **Browser Link Dashboard** window from the Browser Link drop down menu to manage the connection with open browsers:



If no browser is connected, you can start a non-debugging session by selecting the **View in Browser** link:



Otherwise, the connected browsers are shown with the path to the page that each browser is showing:



You can also click on an individual browser name to refresh only that browser.

Enable or disable Browser Link

When you re-enable Browser Link after disabling it, you must refresh the browsers to reconnect them.

Enable or disable CSS Auto-Sync

When CSS Auto-Sync is enabled, connected browsers are automatically refreshed when you make any change to CSS files.

How it works

Browser Link uses SignalR to create a communication channel between Visual Studio and the browser. When Browser Link is enabled, Visual Studio acts as a SignalR server that multiple clients (browsers) can connect to. Browser Link also registers a middleware component in the ASP.NET Core request pipeline. This component injects special `<script>` references into every page request from the server. You can see the script references by selecting

View source in the browser and scrolling to the end of the `<body>` tag content:

```
    <!-- Visual Studio Browser Link -->
    <script type="application/json" id="__browserLink_initializationData">
        {"requestId":"a717d5a07c1741949a7cefd6fa2bad08","requestMappingFromServer":false}
    </script>
    <script type="text/javascript" src="http://localhost:54139/b6e36e429d034f578ebccd6a79bf19bf/browserLink"
async="async"></script>
    <!-- End Browser Link -->
</body>
```

Your source files aren't modified. The middleware component injects the script references dynamically.

Because the browser-side code is all JavaScript, it works on all browsers that SignalR supports without requiring a browser plug-in.

Session and state management in ASP.NET Core

9/22/2020 • 31 minutes to read • Edit Online

By Rick Anderson, Kirk Larkin, and Diana LaRose

HTTP is a stateless protocol. By default, HTTP requests are independent messages that don't retain user values. This article describes several approaches to preserve user data between requests.

View or download sample code (how to download)

State management

State can be stored using several approaches. Each approach is described later in this topic.

STORAGE APPROACH	STORAGE MECHANISM
Cookies	HTTP cookies. May include data stored using server-side app code.
Session state	HTTP cookies and server-side app code
TempData	HTTP cookies or session state
Query strings	HTTP query strings
Hidden fields	HTTP form fields
HttpContext.Items	Server-side app code
Cache	Server-side app code

Cookies

Cookies store data across requests. Because cookies are sent with every request, their size should be kept to a minimum. Ideally, only an identifier should be stored in a cookie with the data stored by the app. Most browsers restrict cookie size to 4096 bytes. Only a limited number of cookies are available for each domain.

Because cookies are subject to tampering, they must be validated by the app. Cookies can be deleted by users and expire on clients. However, cookies are generally the most durable form of data persistence on the client.

Cookies are often used for personalization, where content is customized for a known user. The user is only identified and not authenticated in most cases. The cookie can store the user's name, account name, or unique user ID such as a GUID. The cookie can be used to access the user's personalized settings, such as their preferred website background color.

See the European Union General Data Protection Regulations (GDPR) when issuing cookies and dealing with privacy concerns. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

Session state

Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app. Session state

uses a store maintained by the app to persist data across requests from a client. The session data is backed by a cache and considered ephemeral data. The site should continue to function without the session data. Critical application data should be stored in the user database and cached in session only as a performance optimization.

Session isn't supported in SignalR apps because a SignalR Hub may execute independent of an HTTP context. For example, this can occur when a long polling request is held open by a hub beyond the lifetime of the request's HTTP context.

ASP.NET Core maintains session state by providing a cookie to the client that contains a session ID. The cookie session ID:

- Is sent to the app with each request.
- Is used by the app to fetch the session data.

Session state exhibits the following behaviors:

- The session cookie is specific to the browser. Sessions aren't shared across browsers.
- Session cookies are deleted when the browser session ends.
- If a cookie is received for an expired session, a new session is created that uses the same session cookie.
- Empty sessions aren't retained. The session must have at least one value set to persist the session across requests. When a session isn't retained, a new session ID is generated for each new request.
- The app retains a session for a limited time after the last request. The app either sets the session timeout or uses the default value of 20 minutes. Session state is ideal for storing user data:
  - That's specific to a particular session.
  - Where the data doesn't require permanent storage across sessions.
- Session data is deleted either when the ISession.Clear implementation is called or when the session expires.
- There's no default mechanism to inform app code that a client browser has been closed or when the session cookie is deleted or expired on the client.
- Session state cookies aren't marked essential by default. Session state isn't functional unless tracking is permitted by the site visitor. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

> **WARNING**
> Don't store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows. A session might not be restricted to a single user. The next user might continue to browse the app with the same session cookie.

The in-memory cache provider stores session data in the memory of the server where the app resides. In a server farm scenario:

- Use *sticky sessions* to tie each session to a specific app instance on an individual server. Azure App Service uses Application Request Routing (ARR) to enforce sticky sessions by default. However, sticky sessions can affect scalability and complicate web app updates. A better approach is to use a Redis or SQL Server distributed cache, which doesn't require sticky sessions. For more information, see Distributed caching in ASP.NET Core.
- The session cookie is encrypted via IDataProtector. Data Protection must be properly configured to read session cookies on each machine. For more information, see ASP.NET Core Data Protection and Key storage providers.

Configure session state

The Microsoft.AspNetCore.Session package:

- Is included implicitly by the framework.
- Provides middleware for managing session state.

To enable the session middleware, `Startup` must contain:

- Any of the IDistributedCache memory caches. The `IDistributedCache` implementation is used as a backing store for session. For more information, see Distributed caching in ASP.NET Core.
- A call to AddSession in `ConfigureServices`.
- A call to UseSession in `Configure`.

The following code shows how to set up the in-memory session provider with a default in-memory implementation of `IDistributedCache`:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDistributedMemoryCache();

        services.AddSession(options =>
        {
            options.IdleTimeout = TimeSpan.FromSeconds(10);
            options.Cookie.HttpOnly = true;
            options.Cookie.IsEssential = true;
        });

        services.AddControllersWithViews();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseSession();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute();
            endpoints.MapRazorPages();
        });
    }
}
```

The preceding code sets a short timeout to simplify testing.

The order of middleware is important. Call `UseSession` after `UseRouting` and before `UseEndpoints`. See Middleware Ordering.

HttpContext.Session is available after session state is configured.

`HttpContext.Session` can't be accessed before `UseSession` has been called.

A new session with a new session cookie can't be created after the app has begun writing to the response stream. The exception is recorded in the web server log and not displayed in the browser.

Load session state asynchronously

The default session provider in ASP.NET Core loads session records from the underlying IDistributedCache backing store asynchronously only if the ISession.LoadAsync method is explicitly called before the TryGetValue, Set, or Remove methods. If `LoadAsync` isn't called first, the underlying session record is loaded synchronously, which can incur a performance penalty at scale.

To have apps enforce this pattern, wrap the DistributedSessionStore and DistributedSession implementations with versions that throw an exception if the `LoadAsync` method isn't called before `TryGetValue`, `Set`, or `Remove`. Register the wrapped versions in the services container.

Session options

To override session defaults, use SessionOptions.

OPTION	DESCRIPTION
Cookie	Determines the settings used to create the cookie. Name defaults to SessionDefaults.CookieName (`.AspNetCore.Session`). Path defaults to SessionDefaults.CookiePath (`/`). SameSite defaults to SameSiteMode.Lax (`1`). HttpOnly defaults to `true`. IsEssential defaults to `false`.
IdleTimeout	The `IdleTimeout` indicates how long the session can be idle before its contents are abandoned. Each session access resets the timeout. This setting only applies to the content of the session, not the cookie. The default is 20 minutes.
IOTimeout	The maximum amount of time allowed to load a session from the store or to commit it back to the store. This setting may only apply to asynchronous operations. This timeout can be disabled using InfiniteTimeSpan. The default is 1 minute.

Session uses a cookie to track and identify requests from a single browser. By default, this cookie is named `.AspNetCore.Session`, and it uses a path of `/`. Because the cookie default doesn't specify a domain, it isn't made available to the client-side script on the page (because HttpOnly defaults to `true`).

To override cookie session defaults, use SessionOptions:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDistributedMemoryCache();

    services.AddSession(options =>
    {
        options.Cookie.Name = ".AdventureWorks.Session";
        options.IdleTimeout = TimeSpan.FromSeconds(10);
        options.Cookie.IsEssential = true;
    });

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

The app uses the IdleTimeout property to determine how long a session can be idle before its contents in the server's cache are abandoned. This property is independent of the cookie expiration. Each request that passes through the Session Middleware resets the timeout.

Session state is *non-locking*. If two requests simultaneously attempt to modify the contents of a session, the last request overrides the first. `Session` is implemented as a *coherent session*, which means that all the contents are stored together. When two requests seek to modify different session values, the last request may override session changes made by the first.

Set and get Session values

Session state is accessed from a Razor Pages PageModel class or MVC Controller class with HttpContext.Session. This property is an ISession implementation.

The `ISession` implementation provides several extension methods to set and retrieve integer and string values. The extension methods are in the Microsoft.AspNetCore.Http namespace.

`ISession` extension methods:

- Get(ISession, String)
- GetInt32(ISession, String)
- GetString(ISession, String)
- SetInt32(ISession, String, Int32)
- SetString(ISession, String, String)

The following example retrieves the session value for the `IndexModel.SessionKeyName` key (`_Name` in the sample app) in a Razor Pages page:

```
@page
@using Microsoft.AspNetCore.Http
@model IndexModel

...

Name: @HttpContext.Session.GetString(IndexModel.SessionKeyName)
```

The following example shows how to set and get an integer and a string:

```
public class IndexModel : PageModel
{
    public const string SessionKeyName = "_Name";
    public const string SessionKeyAge = "_Age";
    const string SessionKeyTime = "_Time";

    public string SessionInfo_Name { get; private set; }
    public string SessionInfo_Age { get; private set; }
    public string SessionInfo_CurrentTime { get; private set; }
    public string SessionInfo_SessionTime { get; private set; }
    public string SessionInfo_MiddlewareValue { get; private set; }

    public void OnGet()
    {
        // Requires: using Microsoft.AspNetCore.Http;
        if (string.IsNullOrEmpty(HttpContext.Session.GetString(SessionKeyName)))
        {
            HttpContext.Session.SetString(SessionKeyName, "The Doctor");
            HttpContext.Session.SetInt32(SessionKeyAge, 773);
        }

        var name = HttpContext.Session.GetString(SessionKeyName);
        var age = HttpContext.Session.GetInt32(SessionKeyAge);
```

All session data must be serialized to enable a distributed cache scenario, even when using the in-memory cache. String and integer serializers are provided by the extension methods of ISession. Complex types must be serialized by the user using another mechanism, such as JSON.

Use the following sample code to serialize objects:

```
public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonSerializer.Serialize(value));
    }

    public static T Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return value == null ? default : JsonSerializer.Deserialize<T>(value);
    }
}
```

The following example shows how to set and get a serializable object with the `SessionExtensions` class:

```
// Requires SessionExtensions from sample download.
if (HttpContext.Session.Get<DateTime>(SessionKeyTime) == default)
{
    HttpContext.Session.Set<DateTime>(SessionKeyTime, currentTime);
}
```

TempData

ASP.NET Core exposes the Razor Pages TempData or Controller TempData. This property stores data until it's read in another request. The Keep(String) and Peek(string) methods can be used to examine the data without deletion at the end of the request. Keep marks all items in the dictionary for retention. `TempData` is:

- Useful for redirection when data is required for more than a single request.
- Implemented by `TempData` providers using either cookies or session state.

TempData samples

Consider the following page that creates a customer:

```
public class CreateModel : PageModel
{
    private readonly RazorPagesContactsContext _context;

    public CreateModel(RazorPagesContactsContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";

        return RedirectToPage("./IndexPeek");
    }
}
```

The following page displays `TempData["Message"]`:

```
@page
@model IndexModel

<h1>Peek Contacts</h1>

@{
    if (TempData.Peek("Message") != null)
    {
        <h3>Message: @TempData.Peek("Message")</h3>
    }
}

@*Content removed for brevity.*@
```

In the preceding markup, at the end of the request, `TempData["Message"]` is **not** deleted because `Peek` is used. Refreshing the page displays the contents of `TempData["Message"]`.

The following markup is similar to the preceding code, but uses `Keep` to preserve the data at the end of the request:

```
@page
@model IndexModel

<h1>Contacts Keep</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
    TempData.Keep("Message");
}

@*Content removed for brevity.*@
```

Navigating between the *IndexPeek* and *IndexKeep* pages won't delete `TempData["Message"]`.

The following code displays `TempData["Message"]`, but at the end of the request, `TempData["Message"]` is deleted:

```
@page
@model IndexModel

<h1>Index no Keep or Peek</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
}

@*Content removed for brevity.*@
```

TempData providers

The cookie-based TempData provider is used by default to store TempData in cookies.

The cookie data is encrypted using IDataProtector, encoded with Base64UrlTextEncoder, then chunked. The maximum cookie size is less than 4096 bytes due to encryption and chunking. The cookie data isn't compressed because compressing encrypted data can lead to security problems such as the CRIME and BREACH attacks. For more information on the cookie-based TempData provider, see CookieTempDataProvider.

Choose a TempData provider

Choosing a TempData provider involves several considerations, such as:

- Does the app already use session state? If so, using the session state TempData provider has no additional cost to the app beyond the size of the data.
- Does the app use TempData only sparingly for relatively small amounts of data, up to 500 bytes? If so, the cookie TempData provider adds a small cost to each request that carries TempData. If not, the session state TempData provider can be beneficial to avoid round-tripping a large amount of data in each request until the TempData is consumed.
- Does the app run in a server farm on multiple servers? If so, there's no additional configuration required to use the cookie TempData provider outside of Data Protection (see ASP.NET Core Data Protection and Key storage providers).

Most web clients such as web browsers enforce limits on the maximum size of each cookie and the total number of cookies. When using the cookie TempData provider, verify the app won't exceed these limits. Consider the total size of the data. Account for increases in cookie size due to encryption and chunking.

Configure the TempData provider

The cookie-based TempData provider is enabled by default.

To enable the session-based TempData provider, use the AddSessionStateTempDataProvider extension method. Only one call to `AddSessionStateTempDataProvider` is required:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews()
        .AddSessionStateTempDataProvider();
    services.AddRazorPages()
        .AddSessionStateTempDataProvider();

    services.AddSession();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseSession();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });
}
```

Query strings

A limited amount of data can be passed from one request to another by adding it to the new request's query string. This is useful for capturing state in a persistent manner that allows links with embedded state to be shared through email or social networks. Because URL query strings are public, never use query strings for sensitive data.

In addition to unintended sharing, including data in query strings can expose the app to Cross-Site Request Forgery (CSRF) attacks. Any preserved session state must protect against CSRF attacks. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Hidden fields

Data can be saved in hidden form fields and posted back on the next request. This is common in multi-page forms. Because the client can potentially tamper with the data, the app must always revalidate the data stored in hidden fields.

HttpContext.Items

The HttpContext.Items collection is used to store data while processing a single request. The collection's contents are discarded after a request is processed. The `Items` collection is often used to allow components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters.

In the following example, middleware adds `isVerified` to the `Items` collection:

```
public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
{
    app.UseRouting();

    app.Use(async (context, next) =>
    {
        logger.LogInformation($"Before setting: Verified: {context.Items["isVerified"]}");
        context.Items["isVerified"] = true;
        await next.Invoke();
    });

    app.Use(async (context, next) =>
    {
        logger.LogInformation($"Next: Verified: {context.Items["isVerified"]}");
        await next.Invoke();
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync($"Verified: {context.Items["isVerified"]}");
        });
    });
}
```

For middleware that's only used in a single app, fixed `string` keys are acceptable. Middleware shared between apps should use unique object keys to avoid key collisions. The following example shows how to use a unique object key defined in a middleware class:

```
public class HttpContextItemsMiddleware
{
    private readonly RequestDelegate _next;
    public static readonly object HttpContextItemsMiddlewareKey = new Object();

    public HttpContextItemsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";

        await _next(httpContext);
    }
}

public static class HttpContextItemsMiddlewareExtensions
{
    public static IApplicationBuilder
        UseHttpContextItemsMiddleware(this IApplicationBuilder app)
    {
        return app.UseMiddleware<HttpContextItemsMiddleware>();
    }
}
```

Other code can access the value stored in `HttpContext.Items` using the key exposed by the middleware class:

```
HttpContext.Items
    .TryGetValue(HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey,
        out var middlewareSetValue);
SessionInfo_MiddlewareValue =
    middlewareSetValue?.ToString() ?? "Middleware value not set!";
```

This approach also has the advantage of eliminating the use of key strings in the code.

Cache

Caching is an efficient way to store and retrieve data. The app can control the lifetime of cached items. For more information, see Response caching in ASP.NET Core.

Cached data isn't associated with a specific request, user, or session. **Do not cache user-specific data that may be retrieved by other user requests.**

To cache application wide data, see Cache in-memory in ASP.NET Core.

Common errors

- "Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

  This is typically caused by failing to configure at least one `IDistributedCache` implementation. For more information, see Distributed caching in ASP.NET Core and Cache in-memory in ASP.NET Core.

If the session middleware fails to persist a session:

- The middleware logs the exception and the request continues normally.
- This leads to unpredictable behavior.

The session middleware can fail to persist a session if the backing store isn't available. For example, a user stores a shopping cart in session. The user adds an item to the cart but the commit fails. The app doesn't know about the failure so it reports to the user that the item was added to their cart, which isn't true.

The recommended approach to check for errors is to call `await feature.Session.CommitAsync` when the app is done writing to the session. CommitAsync throws an exception if the backing store is unavailable. If `CommitAsync` fails, the app can process the exception. LoadAsync throws under the same conditions when the data store is unavailable.

SignalR and session state

SignalR apps should not use session state to store information. SignalR apps can store per connection state in `Context.Items` in the hub.

Additional resources

Host ASP.NET Core in a web farm

By Rick Anderson, Steve Smith, Diana LaRose, and Luke Latham

HTTP is a stateless protocol. Without taking additional steps, HTTP requests are independent messages that don't retain user values or app state. This article describes several approaches to preserve user data and app state between requests.

View or download sample code (how to download)

State management

State can be stored using several approaches. Each approach is described later in this topic.

STORAGE APPROACH	STORAGE MECHANISM
Cookies	HTTP cookies (may include data stored using server-side app code)
Session state	HTTP cookies and server-side app code
TempData	HTTP cookies or session state
Query strings	HTTP query strings
Hidden fields	HTTP form fields
HttpContext.Items	Server-side app code
Cache	Server-side app code
Dependency Injection	Server-side app code

Cookies

Cookies store data across requests. Because cookies are sent with every request, their size should be kept to a minimum. Ideally, only an identifier should be stored in a cookie with the data stored by the app. Most browsers restrict cookie size to 4096 bytes. Only a limited number of cookies are available for each domain.

Because cookies are subject to tampering, they must be validated by the app. Cookies can be deleted by users and expire on clients. However, cookies are generally the most durable form of data persistence on the client.

Cookies are often used for personalization, where content is customized for a known user. The user is only identified and not authenticated in most cases. The cookie can store the user's name, account name, or unique user ID (such as a GUID). You can then use the cookie to access the user's personalized settings, such as their preferred website background color.

Be mindful of the European Union General Data Protection Regulations (GDPR) when issuing cookies and dealing with privacy concerns. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

Session state

Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app. Session state uses a store maintained by the app to persist data across requests from a client. The session data is backed by a cache and considered ephemeral data—the site should continue to function without the session data. Critical application data should be stored in the user database and cached in session only as a performance optimization.

> **NOTE**
> Session isn't supported in SignalR apps because a SignalR Hub may execute independent of an HTTP context. For example, this can occur when a long polling request is held open by a hub beyond the lifetime of the request's HTTP context.

ASP.NET Core maintains session state by providing a cookie to the client that contains a session ID, which is sent to the app with each request. The app uses the session ID to fetch the session data.

Session state exhibits the following behaviors:

- Because the session cookie is specific to the browser, sessions aren't shared across browsers.
- Session cookies are deleted when the browser session ends.
- If a cookie is received for an expired session, a new session is created that uses the same session cookie.
- Empty sessions aren't retained—the session must have at least one value set into it to persist the session across requests. When a session isn't retained, a new session ID is generated for each new request.
- The app retains a session for a limited time after the last request. The app either sets the session timeout or uses the default value of 20 minutes. Session state is ideal for storing user data that's specific to a particular session but where the data doesn't require permanent storage across sessions.
- Session data is deleted either when the ISession.Clear implementation is called or when the session expires.
- There's no default mechanism to inform app code that a client browser has been closed or when the session cookie is deleted or expired on the client.
- The ASP.NET Core MVC and Razor pages templates include support for General Data Protection Regulation (GDPR). Session state cookies aren't marked essential by default, so session state isn't functional unless tracking is permitted by the site visitor. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

> **WARNING**
> Don't store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows. A session might not be restricted to a single user—the next user might continue to browse the app with the same session cookie.

The in-memory cache provider stores session data in the memory of the server where the app resides. In a server farm scenario:

- Use *sticky sessions* to tie each session to a specific app instance on an individual server. Azure App Service uses Application Request Routing (ARR) to enforce sticky sessions by default. However, sticky sessions can affect scalability and complicate web app updates. A better approach is to use a Redis or SQL Server distributed cache, which doesn't require sticky sessions. For more information, see Distributed caching in ASP.NET Core.
- The session cookie is encrypted via IDataProtector. Data Protection must be properly configured to read session cookies on each machine. For more information, see ASP.NET Core Data Protection and Key storage providers.

Configure session state

The Microsoft.AspNetCore.Session package, which is included in the Microsoft.AspNetCore.App metapackage, provides middleware for managing session state. To enable the session middleware, `Startup` must contain:

- Any of the IDistributedCache memory caches. The `IDistributedCache` implementation is used as a backing store for session. For more information, see Distributed caching in ASP.NET Core.
- A call to AddSession in `ConfigureServices`.
- A call to UseSession in `Configure`.

The following code shows how to set up the in-memory session provider with a default in-memory implementation of `IDistributedCache`:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDistributedMemoryCache();

        services.AddSession(options =>
        {
            // Set a short timeout for easy testing.
            options.IdleTimeout = TimeSpan.FromSeconds(10);
            options.Cookie.HttpOnly = true;
            // Make the session cookie essential
            options.Cookie.IsEssential = true;
        });

        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseSession();
        app.UseHttpContextItemsMiddleware();
        app.UseMvc();
    }
}
```

The order of middleware is important. In the preceding example, an `InvalidOperationException` exception occurs when `UseSession` is invoked after `UseMvc`. For more information, see Middleware Ordering.

HttpContext.Session is available after session state is configured.

`HttpContext.Session` can't be accessed before `UseSession` has been called.

A new session with a new session cookie can't be created after the app has begun writing to the response stream. The exception is recorded in the web server log and not displayed in the browser.

Load session state asynchronously

The default session provider in ASP.NET Core loads session records from the underlying IDistributedCache backing store asynchronously only if the ISession.LoadAsync method is explicitly called before the TryGetValue, Set, or Remove methods. If `LoadAsync` isn't called first, the underlying session record is loaded synchronously, which can incur a performance penalty at scale.

To have apps enforce this pattern, wrap the DistributedSessionStore and DistributedSession implementations with versions that throw an exception if the `LoadAsync` method isn't called before `TryGetValue`, `Set`, or `Remove`. Register the wrapped versions in the services container.

Session options

To override session defaults, use SessionOptions.

OPTION	DESCRIPTION
Cookie	Determines the settings used to create the cookie. Name defaults to SessionDefaults.CookieName (`.AspNetCore.Session`). Path defaults to SessionDefaults.CookiePath (`/`). SameSite defaults to SameSiteMode.Lax (`1`). HttpOnly defaults to `true` . IsEssential defaults to `false` .
IdleTimeout	The `IdleTimeout` indicates how long the session can be idle before its contents are abandoned. Each session access resets the timeout. This setting only applies to the content of the session, not the cookie. The default is 20 minutes.
IOTimeout	The maximum amount of time allowed to load a session from the store or to commit it back to the store. This setting may only apply to asynchronous operations. This timeout can be disabled using InfiniteTimeSpan. The default is 1 minute.

Session uses a cookie to track and identify requests from a single browser. By default, this cookie is named `.AspNetCore.Session` , and it uses a path of `/` . Because the cookie default doesn't specify a domain, it isn't made available to the client-side script on the page (because HttpOnly defaults to `true`).

To override cookie session defaults, use `SessionOptions` :

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDistributedMemoryCache();

    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddSession(options =>
    {
        options.Cookie.Name = ".AdventureWorks.Session";
        options.IdleTimeout = TimeSpan.FromSeconds(10);
        options.Cookie.IsEssential = true;
    });
}
```

The app uses the IdleTimeout property to determine how long a session can be idle before its contents in the server's cache are abandoned. This property is independent of the cookie expiration. Each request that passes through the Session Middleware resets the timeout.

Session state is *non-locking*. If two requests simultaneously attempt to modify the contents of a session, the last request overrides the first. `Session` is implemented as a *coherent session*, which means that all the contents are stored together. When two requests seek to modify different session values, the last request may override session changes made by the first.

Set and get Session values

Session state is accessed from a Razor Pages PageModel class or MVC Controller class with HttpContext.Session. This property is an ISession implementation.

The `ISession` implementation provides several extension methods to set and retrieve integer and string values. The extension methods are in the Microsoft.AspNetCore.Http namespace (add a `using Microsoft.AspNetCore.Http;` statement to gain access to the extension methods) when the Microsoft.AspNetCore.Http.Extensions package is referenced by the project. Both packages are included in the Microsoft.AspNetCore.App metapackage.

`ISession` extension methods:

- Get(ISession, String)
- GetInt32(ISession, String)
- GetString(ISession, String)
- SetInt32(ISession, String, Int32)
- SetString(ISession, String, String)

The following example retrieves the session value for the `IndexModel.SessionKeyName` key (`_Name` in the sample app) in a Razor Pages page:

```
@page
@using Microsoft.AspNetCore.Http
@model IndexModel

...

Name: @HttpContext.Session.GetString(IndexModel.SessionKeyName)
```

The following example shows how to set and get an integer and a string:

```
public class IndexModel : PageModel
{
    public const string SessionKeyName = "_Name";
    public const string SessionKeyAge = "_Age";
    const string SessionKeyTime = "_Time";

    public string SessionInfo_Name { get; private set; }
    public string SessionInfo_Age { get; private set; }
    public string SessionInfo_CurrentTime { get; private set; }
    public string SessionInfo_SessionTime { get; private set; }
    public string SessionInfo_MiddlewareValue { get; private set; }

    public void OnGet()
    {
        // Requires: using Microsoft.AspNetCore.Http;
        if (string.IsNullOrEmpty(HttpContext.Session.GetString(SessionKeyName)))
        {
            HttpContext.Session.SetString(SessionKeyName, "The Doctor");
            HttpContext.Session.SetInt32(SessionKeyAge, 773);
        }

        var name = HttpContext.Session.GetString(SessionKeyName);
        var age = HttpContext.Session.GetInt32(SessionKeyAge);
```

All session data must be serialized to enable a distributed cache scenario, even when using the in-memory cache. String and integer serializers are provided by the extension methods of ISession). Complex types must be serialized by the user using another mechanism, such as JSON.

Add the following extension methods to set and get serializable objects:

```
public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);

        return value == null ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }
}
```

The following example shows how to set and get a serializable object with the extension methods:

```
// Requires you add the Set and Get extension method mentioned in the topic.
if (HttpContext.Session.Get<DateTime>(SessionKeyTime) == default(DateTime))
{
    HttpContext.Session.Set<DateTime>(SessionKeyTime, currentTime);
}
```

TempData

ASP.NET Core exposes the Razor Pages TempData or Controller TempData. This property stores data until it's read in another request. Keep(String) and Peek(string) methods can be used to examine the data without deletion at the

end of the request. Keep() marks all items in the dictionary for retention. `TempData` is particularly useful for redirection when data is required for more than a single request. `TempData` is implemented by `TempData` providers using either cookies or session state.

TempData samples

Consider the following page that creates a customer:

```
public class CreateModel : PageModel
{
    private readonly RazorPagesContactsContext _context;

    public CreateModel(RazorPagesContactsContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";

        return RedirectToPage("./IndexPeek");
    }
}
```

The following page displays `TempData["Message"]`:

```
@page
@model IndexModel

<h1>Peek Contacts</h1>

@{
    if (TempData.Peek("Message") != null)
    {
        <h3>Message: @TempData.Peek("Message")</h3>
    }
}

@*Content removed for brevity.*@
```

In the preceding markup, at the end of the request, `TempData["Message"]` is **not** deleted because `Peek` is used. Refreshing the page displays `TempData["Message"]`.

The following markup is similar to the preceding code, but uses `Keep` to preserve the data at the end of the request:

```
@page
@model IndexModel

<h1>Contacts Keep</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
    TempData.Keep("Message");
}

@*Content removed for brevity.*@
```

Navigating between the *IndexPeek* and *IndexKeep* pages won't delete `TempData["Message"]`.

The following code displays `TempData["Message"]`, but at the end of the request, `TempData["Message"]` is deleted:

```
@page
@model IndexModel

<h1>Index no Keep or Peek</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
}

@*Content removed for brevity.*@
```

TempData providers

The cookie-based TempData provider is used by default to store TempData in cookies.

The cookie data is encrypted using IDataProtector, encoded with Base64UrlTextEncoder, then chunked. Because the cookie is chunked, the single cookie size limit found in ASP.NET Core 1.x doesn't apply. The cookie data isn't compressed because compressing encrypted data can lead to security problems such as the CRIME and BREACH attacks. For more information on the cookie-based TempData provider, see CookieTempDataProvider.

Choose a TempData provider

Choosing a TempData provider involves several considerations, such as:

1. Does the app already use session state? If so, using the session state TempData provider has no additional cost to the app (aside from the size of the data).
2. Does the app use TempData only sparingly for relatively small amounts of data (up to 500 bytes)? If so, the cookie TempData provider adds a small cost to each request that carries TempData. If not, the session state TempData provider can be beneficial to avoid round-tripping a large amount of data in each request until the TempData is consumed.
3. Does the app run in a server farm on multiple servers? If so, there's no additional configuration required to use the cookie TempData provider outside of Data Protection (see ASP.NET Core Data Protection and Key storage providers).

> **NOTE**
>
> Most web clients (such as web browsers) enforce limits on the maximum size of each cookie, the total number of cookies, or both. When using the cookie TempData provider, verify the app won't exceed these limits. Consider the total size of the data. Account for increases in cookie size due to encryption and chunking.

Configure the TempData provider

The cookie-based TempData provider is enabled by default.

To enable the session-based TempData provider, use the AddSessionStateTempDataProvider extension method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
        .AddSessionStateTempDataProvider();

    services.AddSession();
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseSession();
    app.UseMvc();
}
```

The order of middleware is important. In the preceding example, an `InvalidOperationException` exception occurs when `UseSession` is invoked after `UseMvc`. For more information, see Middleware Ordering.

> **IMPORTANT**
>
> If targeting .NET Framework and using the session-based TempData provider, add the Microsoft.AspNetCore.Session package to the project.

Query strings

A limited amount of data can be passed from one request to another by adding it to the new request's query string. This is useful for capturing state in a persistent manner that allows links with embedded state to be shared through email or social networks. Because URL query strings are public, never use query strings for sensitive data.

In addition to unintended sharing, including data in query strings can create opportunities for Cross-Site Request Forgery (CSRF) attacks, which can trick users into visiting malicious sites while authenticated. Attackers can then steal user data from the app or take malicious actions on behalf of the user. Any preserved app or session state must protect against CSRF attacks. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Hidden fields

Data can be saved in hidden form fields and posted back on the next request. This is common in multi-page forms. Because the client can potentially tamper with the data, the app must always revalidate the data stored in hidden fields.

HttpContext.Items

The HttpContext.Items collection is used to store data while processing a single request. The collection's contents are discarded after a request is processed. The `Items` collection is often used to allow components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters.

In the following example, middleware adds `isVerified` to the `Items` collection.

```
app.Use(async (context, next) =>
{
    // perform some verification
    context.Items["isVerified"] = true;
    await next.Invoke();
});
```

Later in the pipeline, another middleware can access the value of `isVerified`:

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync($"Verified: {context.Items["isVerified"]}");
});
```

For middleware that's only used by a single app, `string` keys are acceptable. Middleware shared between app instances should use unique object keys to avoid key collisions. The following example shows how to use a unique object key defined in a middleware class:

```
public class HttpContextItemsMiddleware
{
    private readonly RequestDelegate _next;
    public static readonly object HttpContextItemsMiddlewareKey = new Object();

    public HttpContextItemsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";

        await _next(httpContext);
    }
}

public static class HttpContextItemsMiddlewareExtensions
{
    public static IApplicationBuilder
        UseHttpContextItemsMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<HttpContextItemsMiddleware>();
    }
}
```

Other code can access the value stored in `HttpContext.Items` using the key exposed by the middleware class:

```
HttpContext.Items
    .TryGetValue(HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey,
        out var middlewareSetValue);
SessionInfo_MiddlewareValue =
    middlewareSetValue?.ToString() ?? "Middleware value not set!";
```

This approach also has the advantage of eliminating the use of key strings in the code.

Cache

Caching is an efficient way to store and retrieve data. The app can control the lifetime of cached items.

Cached data isn't associated with a specific request, user, or session. **Be careful not to cache user-specific data that may be retrieved by other users' requests.**

For more information, see Response caching in ASP.NET Core.

Dependency Injection

Use Dependency Injection to make data available to all users:

1. Define a service containing the data. For example, a class named `MyAppData` is defined:

   ```
   public class MyAppData
   {
       // Declare properties and methods
   }
   ```

2. Add the service class to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<MyAppData>();
}
```

3. Consume the data service class:

```
public class IndexModel : PageModel
{
    public IndexModel(MyAppData myService)
    {
        // Do something with the service
        //    Examples: Read data, store in a field or property
    }
}
```

Common errors

- "Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

  This is usually caused by failing to configure at least one `IDistributedCache` implementation. For more information, see Distributed caching in ASP.NET Core and Cache in-memory in ASP.NET Core.

- In the event that the session middleware fails to persist a session (for example, if the backing store isn't available), the middleware logs the exception and the request continues normally. This leads to unpredictable behavior.

  For example, a user stores a shopping cart in session. The user adds an item to the cart but the commit fails. The app doesn't know about the failure so it reports to the user that the item was added to their cart, which isn't true.

  The recommended approach to check for errors is to call `await feature.Session.CommitAsync();` from app code when the app is done writing to the session. `CommitAsync` throws an exception if the backing store is unavailable. If `CommitAsync` fails, the app can process the exception. `LoadAsync` throws under the same conditions where the data store is unavailable.

SignalR and session state

SignalR apps should not use session state to store information. SignalR apps can store per connection state in `Context.Items` in the hub.

Additional resources

Host ASP.NET Core in a web farm

Layout in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

By Steve Smith and Dave Brock

Pages and views frequently share visual and programmatic elements. This article demonstrates how to:

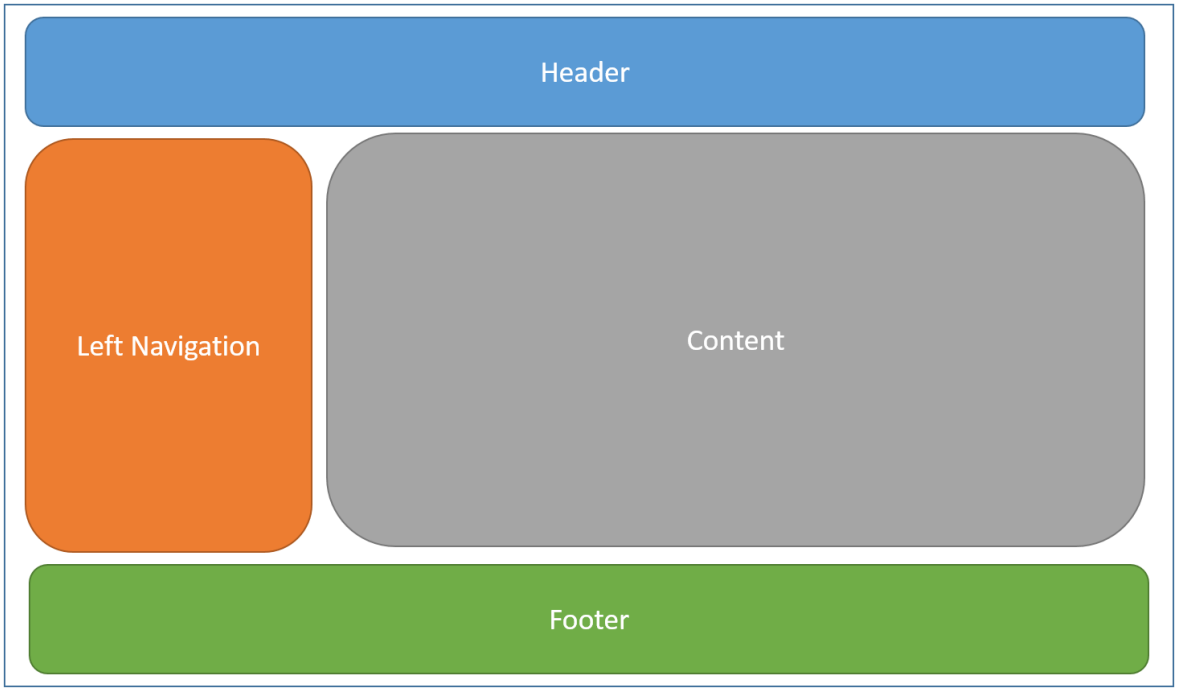
- Use common layouts.
- Share directives.
- Run common code before rendering pages or views.

This document discusses layouts for the two different approaches to ASP.NET Core MVC: Razor Pages and controllers with views. For this topic, the differences are minimal:

- Razor Pages are in the *Pages* folder.
- Controllers with views uses a *Views* folder for views.

What is a Layout

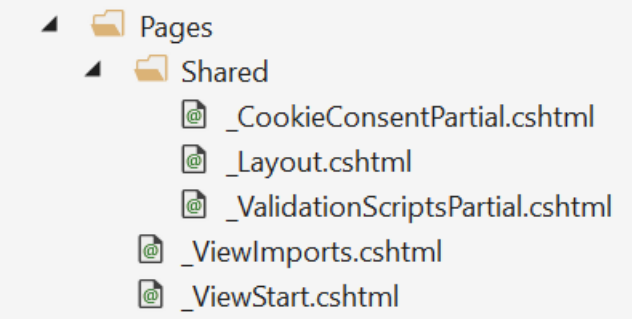
Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.



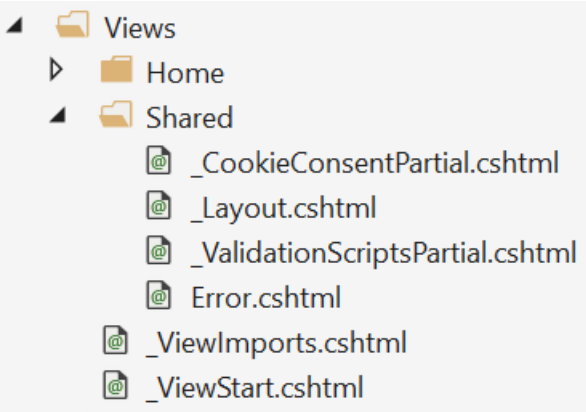
Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a *layout* file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views.

By convention, the default layout for an ASP.NET Core app is named *_Layout.cshtml*. The layout files for new ASP.NET Core projects created with the templates are:

- Razor Pages: *Pages/Shared/_Layout.cshtml*



- Controller with views: *Views/Shared/_Layout.cshtml*



The layout defines a top level template for views in the app. Apps don't require a layout. Apps can define more than one layout, with different views specifying different layouts.

The following code shows the layout file for a template created project with a controller and views:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-
test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">WebApplication1</a>
            </div>
            <div class="navbar-collapse collapse">
```

```
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Contact">Contact</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - WebApplication1</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

Specifying a Layout

Razor views have a `Layout` property. Individual views specify a layout by setting this property:

```
@{
    Layout = "_Layout";
}
```

The layout specified can use a full path (for example, */Pages/Shared/_Layout.cshtml* or */Views/Shared/_Layout.cshtml*) or a partial name (example: `_Layout`). When a partial name is provided, the Razor view engine searches for the layout file using its standard discovery process. The folder where the handler method (or controller) exists is searched first, followed by the *Shared* folder. This discovery process is identical to the process used to discover partial views.

By default, every layout must call `RenderBody`. Wherever the call to `RenderBody` is placed, the contents of the view will be rendered.

Sections

A layout can optionally reference one or more *sections*, by calling `RenderSection`. Sections provide a way to organize where certain page elements should be placed. Each call to `RenderSection` can specify whether that section is required or optional:

```
<script type="text/javascript" src="~/scripts/global.js"></script>

@RenderSection("Scripts", required: false)
```

If a required section isn't found, an exception is thrown. Individual views specify the content to be rendered within a section using the `@section` Razor syntax. If a page or view defines a section, it must be rendered (or an error will occur).

An example `@section` definition in Razor Pages view:

```
@section Scripts {
     <script type="text/javascript" src="~/scripts/main.js"></script>
}
```

In the preceding code, *scripts/main.js* is added to the `scripts` section on a page or view. Other pages or views in the same app might not require this script and wouldn't define a scripts section.

The following markup uses the Partial Tag Helper to render *_ValidationScriptsPartial.cshtml*:

```
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

The preceding markup was generated by scaffolding Identity.

Sections defined in a page or view are available only in its immediate layout page. They cannot be referenced from partials, view components, or other parts of the view system.

Ignoring sections

By default, the body and all sections in a content page must all be rendered by the layout page. The Razor view engine enforces this by tracking whether the body and each section have been rendered.

To instruct the view engine to ignore the body or sections, call the `IgnoreBody` and `IgnoreSection` methods.

The body and every section in a Razor page must be either rendered or ignored.

Importing Shared Directives

Views and pages can use Razor directives to import namespaces and use dependency injection. Directives shared by many views may be specified in a common *_ViewImports.cshtml* file. The `_ViewImports` file supports the following directives:

- `@addTagHelper`
- `@removeTagHelper`
- `@tagHelperPrefix`
- `@using`
- `@model`
- `@inherits`
- `@inject`

The file doesn't support other Razor features, such as functions and section definitions.

A sample `_ViewImports.cshtml` file:

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The *_ViewImports.cshtml* file for an ASP.NET Core MVC app is typically placed in the *Pages* (or *Views*) folder. A *_ViewImports.cshtml* file can be placed within any folder, in which case it will only be applied to pages or views within that folder and its subfolders. `_ViewImports` files are processed starting at the root level and then for each folder leading up to the location of the page or view itself. `_ViewImports` settings specified at the root level may be overridden at the folder level.

For example, suppose:

- The root level *_ViewImports.cshtml* file includes `@model MyModel1` and `@addTagHelper *, MyTagHelper1`.
- A subfolder *_ViewImports.cshtml* file includes `@model MyModel2` and `@addTagHelper *, MyTagHelper2`.

Pages and views in the subfolder will have access to both Tag Helpers and the `MyModel2` model.

If multiple *_ViewImports.cshtml* files are found in the file hierarchy, the combined behavior of the directives are:

- `@addTagHelper`, `@removeTagHelper`: all run, in order
- `@tagHelperPrefix`: the closest one to the view overrides any others
- `@model`: the closest one to the view overrides any others
- `@inherits`: the closest one to the view overrides any others
- `@using`: all are included; duplicates are ignored
- `@inject`: for each property, the closest one to the view overrides any others with the same property name

Running Code Before Each View

Code that needs to run before each view or page should be placed in the *_ViewStart.cshtml* file. By convention, the *_ViewStart.cshtml* file is located in the *Pages* (or *Views*) folder. The statements listed in *_ViewStart.cshtml* are run before every full view (not layouts, and not partial views). Like ViewImports.cshtml, *_ViewStart.cshtml* is hierarchical. If a *_ViewStart.cshtml* file is defined in the view or pages folder, it will be run after the one defined in the root of the *Pages* (or *Views*) folder (if any).

A sample *_ViewStart.cshtml* file:

```
@{
    Layout = "_Layout";
}
```

The file above specifies that all views will use the *_Layout.cshtml* layout.

_ViewStart.cshtml and *_ViewImports.cshtml* are **not** typically placed in the */Pages/Shared* (or */Views/Shared*) folder. The app-level versions of these files should be placed directly in the */Pages* (or */Views*) folder.

Razor syntax reference for ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, Taylor Mullen, and Dan Vicarel

Razor is a markup syntax for embedding server-based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a *.cshtml* file extension. Razor is also found in Razor components files (*.razor*).

Rendering HTML

The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in *.cshtml* Razor files is rendered by the server unchanged.

Razor syntax

Razor supports C# and uses the `@` symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output.

When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup. Otherwise, it transitions into plain C#.

To escape an `@` symbol in Razor markup, use a second `@` symbol:

```
<p>@@Username</p>
```

The code is rendered in HTML with a single `@` symbol:

```
<p>@Username</p>
```

HTML attributes and content containing email addresses don't treat the `@` symbol as a transition character. The email addresses in the following example are untouched by Razor parsing:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Implicit Razor expressions

Implicit Razor expressions start with `@` followed by C# code:

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

With the exception of the C# `await` keyword, implicit expressions must not contain spaces. If the C# statement has a clear ending, spaces can be intermingled:

```
<p>@await DoSomething("hello", "world")</p>
```

Implicit expressions **cannot** contain C# generics, as the characters inside the brackets (`<>`) are interpreted as

an HTML tag. The following code is **not** valid:

```
<p>@GenericMethod<int>()</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element wasn't closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?`

Generic method calls must be wrapped in an explicit Razor expression or a Razor code block.

Explicit Razor expressions

Explicit Razor expressions consist of an `@` symbol with balanced parenthesis. To render last week's time, the following Razor markup is used:

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Any content within the `@()` parenthesis is evaluated and rendered to the output.

Implicit expressions, described in the previous section, generally can't contain spaces. In the following code, one week isn't subtracted from the current time:

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

The code renders the following HTML:

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

Explicit expressions can be used to concatenate text with an expression result:

```
@{
    var joe = new Person("Joe", 33);
}

<p>Age@(joe.Age)</p>
```

Without the explicit expression, `<p>Age@joe.Age</p>` is treated as an email address, and `<p>Age@joe.Age</p>` is rendered. When written as an explicit expression, `<p>Age33</p>` is rendered.

Explicit expressions can be used to render output from generic methods in *.cshtml* files. The following markup shows how to correct the error shown earlier caused by the brackets of a C# generic. The code is written as an explicit expression:

```
<p>@(GenericMethod<int>())</p>
```

Expression encoding

C# expressions that evaluate to a string are HTML encoded. C# expressions that evaluate to `IHtmlContent` are rendered directly through `IHtmlContent.WriteTo`. C# expressions that don't evaluate to `IHtmlContent` are converted to a string by `ToString` and encoded before they're rendered.

```
@("<span>Hello World</span>")
```

The preceding code renders the following HTML:

```
&lt;span&gt;Hello World&lt;/span&gt;
```

The HTML is shown in the browser as plain text:

Hello World

`HtmlHelper.Raw` output isn't encoded but rendered as HTML markup.

> **WARNING**
> Using `HtmlHelper.Raw` on unsanitized user input is a security risk. User input might contain malicious JavaScript or other exploits. Sanitizing user input is difficult. Avoid using `HtmlHelper.Raw` with user input.

```
@Html.Raw("<span>Hello World</span>")
```

The code renders the following HTML:

```
<span>Hello World</span>
```

Razor code blocks

Razor code blocks start with `@` and are enclosed by `{}`. Unlike expressions, C# code inside code blocks isn't rendered. Code blocks and expressions in a view share the same scope and are defined in order:

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.";
}

<p>@quote</p>
```

The code renders the following HTML:

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King, Jr.</p>
```

In code blocks, declare local functions with markup to serve as templating methods:

```
@{
    void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }

    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}
```

The code renders the following HTML:

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

Implicit transitions

The default language in a code block is C#, but the Razor Page can transition back to HTML:

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Explicit delimited transition

To define a subsection of a code block that should render HTML, surround the characters for rendering with the Razor `<text>` tag:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

Use this approach to render HTML that isn't surrounded by an HTML tag. Without an HTML or Razor tag, a Razor runtime error occurs.

The `<text>` tag is useful to control whitespace when rendering content:

- Only the content between the `<text>` tag is rendered.
- No whitespace before or after the `<text>` tag appears in the HTML output.

Explicit line transition

To render the rest of an entire line as HTML inside a code block, use `@:` syntax:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

Without the `@:` in the code, a Razor runtime error is generated.

Extra `@` characters in a Razor file can cause compiler errors at statements later in the block. These compiler errors can be difficult to understand because the actual error occurs before the reported error. This error is common after combining multiple implicit/explicit expressions into a single code block.

Control structures

Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures:

Conditionals `@if, else if, else, and @switch`

`@if` controls when code runs:

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

`else` and `else if` don't require the `@` symbol:

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

The following markup shows how to use a switch statement:

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

Looping `@for, @foreach, @while, and @do while`

Templated HTML can be rendered with looping control statements. To render a list of people:

```
@{
    var people = new Person[]
    {
          new Person("Weston", 33),
          new Person("Johnathon", 41),
          ...
    };
}
```

The following looping statements are supported:

`@for`

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@foreach`

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@while`

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

`@do while`

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

Compound `@using`

In C#, a `using` statement is used to ensure an object is disposed. In Razor, the same mechanism is used to create HTML Helpers that contain additional content. In the following code, HTML Helpers render a `<form>` tag with the `@using` statement:

```
@using (Html.BeginForm())
{
    <div>
        Email: <input type="email" id="Email" value="">
        <button>Register</button>
    </div>
}
```

`@try, catch, finally`

Exception handling is similar to C#:

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

`@lock`

Razor has the capability to protect critical sections with lock statements:

```
@lock (SomeLock)
{
    // Do critical section work
}
```

Comments

Razor supports C# and HTML comments:

```
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
```

The code renders the following HTML:

```
<!-- HTML comment -->
```

Razor comments are removed by the server before the webpage is rendered. Razor uses `@* *@` to delimit comments. The following code is commented out, so the server doesn't render any markup:

```
@*
    @{
        /* C# comment */
        // Another C# comment
    }
    <!-- HTML comment -->
*@
```

Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive typically changes the way a view is parsed or enables different functionality.

Understanding how Razor generates code for a view makes it easier to understand how directives work.

```
@{
    var quote = "Getting old ain't for wimps! - Anonymous";
}

<div>Quote of the Day: @quote</div>
```

The code generates a class similar to the following:

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

Later in this article, the section Inspect the Razor C# class generated for a view explains how to view this generated class.

`@attribute`

The `@attribute` directive adds the given attribute to the class of the generated page or view. The following example adds the `[Authorize]` attribute:

```
@attribute [Authorize]
```

`@code`

This scenario only applies to Razor components (.razor).

The `@code` block enables a Razor component to add C# members (fields, properties, and methods) to a component:

```
@code {
    // C# members (fields, properties, and methods)
}
```

For Razor components, `@code` is an alias of `@functions` and recommended over `@functions`. More than one `@code` block is permissible.

`@functions`

The `@functions` directive enables adding C# members (fields, properties, and methods) to the generated class:

```
@functions {
    // C# members (fields, properties, and methods)
}
```

In Razor components, use `@code` over `@functions` to add C# members.

For example:

```
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

The code generates the following HTML markup:

```
<div>From method: Hello</div>
```

The following code is the generated Razor C# class:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Razor;

public class _Views_Home_Test_cshtml : RazorPage<dynamic>
{
    // Functions placed between here
    public string GetHello()
    {
        return "Hello";
    }
    // And here.
#pragma warning disable 1998
    public override async Task ExecuteAsync()
    {
        WriteLiteral("\r\n<div>From method: ");
        Write(GetHello());
        WriteLiteral("</div>\r\n");
    }
#pragma warning restore 1998
```

`@functions` methods serve as templating methods when they have markup:

```
@{
    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}

@functions {
    private void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }
}
```

The code renders the following HTML:

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

`@implements`

The `@implements` directive implements an interface for the generated class.

The following example implements System.IDisposable so that the Dispose method can be called:

```
@implements IDisposable

<h1>Example</h1>

@functions {
    private bool _isDisposed;

    ...

    public void Dispose() => _isDisposed = true;
}
```

`@inherits`

The `@inherits` directive provides full control of the class the view inherits:

```
@inherits TypeNameOfClassToInheritFrom
```

The following code is a custom Razor page type:

```
using Microsoft.AspNetCore.Mvc.Razor;

public abstract class CustomRazorPage<TModel> : RazorPage<TModel>
{
    public string CustomText { get; } =
        "Gardyloo! - A Scottish warning yelled from a window before dumping" +
        "a slop bucket on the street below.";
}
```

The `CustomText` is displayed in a view:

```
@inherits CustomRazorPage<TModel>

<div>Custom text: @CustomText</div>
```

The code renders the following HTML:

```
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping
    a slop bucket on the street below.
</div>
```

`@model` and `@inherits` can be used in the same view. `@inherits` can be in a *_ViewImports.cshtml* file that the view imports:

```
@inherits CustomRazorPage<TModel>
```

The following code is an example of a strongly-typed view:

```
@inherits CustomRazorPage<TModel>

<div>The Login Email: @Model.Email</div>
<div>Custom text: @CustomText</div>
```

If "rick@contoso.com" is passed in the model, the view generates the following HTML markup:

```
<div>The Login Email: rick@contoso.com</div>
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before dumping
    a slop bucket on the street below.
</div>
```

`@inject`

The `@inject` directive enables the Razor Page to inject a service from the service container into a view. For more information, see Dependency injection into views.

`@layout`

This scenario only applies to Razor components (.razor).

The `@layout` directive specifies a layout for a Razor component. Layout components are used to avoid code duplication and inconsistency. For more information, see ASP.NET Core Blazor layouts.

`@model`

This scenario only applies to MVC views and Razor Pages (.cshtml).

The `@model` directive specifies the type of the model passed to a view or page:

```
@model TypeNameOfModel
```

In an ASP.NET Core MVC or Razor Pages app created with individual user accounts, *Views/Account/Login.cshtml* contains the following model declaration:

```
@model LoginViewModel
```

The class generated inherits from `RazorPage<dynamic>`:

```
public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor exposes a `Model` property for accessing the model passed to the view:

```
<div>The Login Email: @Model.Email</div>
```

The `@model` directive specifies the type of the `Model` property. The directive specifies the `T` in `RazorPage<T>` that the generated class that the view derives from. If the `@model` directive isn't specified, the `Model` property is of type `dynamic`. For more information, see Strongly typed models and the @model keyword.

`@namespace`

The `@namespace` directive:

- Sets the namespace of the class of the generated Razor page, MVC view, or Razor component.
- Sets the root derived namespaces of a pages, views, or components classes from the closest imports file in the directory tree, *_ViewImports.cshtml* (views or pages) or *_Imports.razor* (Razor components).

```
@namespace Your.Namespace.Here
```

For the Razor Pages example shown in the following table:

- Each page imports *Pages/_ViewImports.cshtml*.
- *Pages/_ViewImports.cshtml* contains `@namespace Hello.World`.
- Each page has `Hello.World` as the root of it's namespace.

PAGE	NAMESPACE
Pages/Index.cshtml	`Hello.World`
Pages/MorePages/Page.cshtml	`Hello.World.MorePages`
Pages/MorePages/EvenMorePages/Page.cshtml	`Hello.World.MorePages.EvenMorePages`

The preceding relationships apply to import files used with MVC views and Razor components.

When multiple import files have a `@namespace` directive, the file closest to the page, view, or component in the directory tree is used to set the root namespace.

If the *EvenMorePages* folder in the preceding example has an imports file with `@namespace Another.Planet` (or the *Pages/MorePages/EvenMorePages/Page.cshtml* file contains `@namespace Another.Planet`), the result is shown in the following table.

PAGE	NAMESPACE
Pages/Index.cshtml	`Hello.World`
Pages/MorePages/Page.cshtml	`Hello.World.MorePages`
Pages/MorePages/EvenMorePages/Page.cshtml	`Another.Planet`

`@page`

The `@page` directive has different effects depending on the type of the file where it appears. The directive:

- In a *.cshtml* file indicates that the file is a Razor Page. For more information, see Custom routes and Introduction to Razor Pages in ASP.NET Core.
- Specifies that a Razor component should handle requests directly. For more information, see ASP.NET Core Blazor routing.

The `@page` directive on the first line of a *.cshtml* file indicates that the file is a Razor Page. For more information, see Introduction to Razor Pages in ASP.NET Core.

`@section`

This scenario only applies to MVC views and Razor Pages (.cshtml).

The `@section` directive is used in conjunction with MVC and Razor Pages layouts to enable views or pages to render content in different parts of the HTML page. For more information, see Layout in ASP.NET Core.

`@using`

The `@using` directive adds the C# `using` directive to the generated view:

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

In Razor components, `@using` also controls which components are in scope.

Directive attributes

Razor directive attributes are represented by implicit expressions with reserved keywords following the `@` symbol. A directive attribute typically changes the way an element is parsed or enables different functionality.

`@attributes`

This scenario only applies to Razor components (.razor).

`@attributes` allows a component to render non-declared attributes. For more information, see Create and use ASP.NET Core Razor components.

`@bind`

This scenario only applies to Razor components (.razor).

Data binding in components is accomplished with the `@bind` attribute. For more information, see ASP.NET Core Blazor data binding.

`@on{EVENT}`

This scenario only applies to Razor components (.razor).

Razor provides event handling features for components. For more information, see ASP.NET Core Blazor event handling.

`@on{EVENT}:preventDefault`

This scenario only applies to Razor components (.razor).

Prevents the default action for the event.

`@on{EVENT}:stopPropagation`

This scenario only applies to Razor components (.razor).

Stops event propagation for the event.

`@key`

This scenario only applies to Razor components (.razor).

The `@key` directive attribute causes the components diffing algorithm to guarantee preservation of elements or components based on the key's value. For more information, see Create and use ASP.NET Core Razor components.

`@ref`

This scenario only applies to Razor components (.razor).

Component references (`@ref`) provide a way to reference a component instance so that you can issue commands to that instance. For more information, see Create and use ASP.NET Core Razor components.

`@typeparam`

This scenario only applies to Razor components (.razor).

The `@typeparam` directive declares a generic type parameter for the generated component class. For more information, see ASP.NET Core Blazor templated components.

Templated Razor delegates

Razor templates allow you to define a UI snippet with the following format:

```
@<tag>...</tag>
```

The following example illustrates how to specify a templated Razor delegate as a Func<T,TResult>. The dynamic type is specified for the parameter of the method that the delegate encapsulates. An object type is specified as the return value of the delegate. The template is used with a List<T> of `Pet` that has a `Name` property.

```
public class Pet
{
    public string Name { get; set; }
}
```

```
@{
    Func<dynamic, object> petTemplate = @<p>You have a pet named <strong>@item.Name</strong>.</p>;

    var pets = new List<Pet>
    {
        new Pet { Name = "Rin Tin Tin" },
        new Pet { Name = "Mr. Bigglesworth" },
        new Pet { Name = "K-9" }
    };
}
```

The template is rendered with `pets` supplied by a `foreach` statement:

```
@foreach (var pet in pets)
{
    @petTemplate(pet)
}
```

Rendered output:

```
<p>You have a pet named <strong>Rin Tin Tin</strong>.</p>
<p>You have a pet named <strong>Mr. Bigglesworth</strong>.</p>
<p>You have a pet named <strong>K-9</strong>.</p>
```

You can also supply an inline Razor template as an argument to a method. In the following example, the `Repeat` method receives a Razor template. The method uses the template to produce HTML content with repeats of items supplied from a list:

```
@using Microsoft.AspNetCore.Html

@functions {
    public static IHtmlContent Repeat(IEnumerable<dynamic> items, int times,
        Func<dynamic, IHtmlContent> template)
    {
        var html = new HtmlContentBuilder();

        foreach (var item in items)
        {
            for (var i = 0; i < times; i++)
            {
                html.AppendHtml(template(item));
            }
        }

        return html;
    }
}
```

Using the list of pets from the prior example, the `Repeat` method is called with:

- List<T> of `Pet`.
- Number of times to repeat each pet.
- Inline template to use for the list items of an unordered list.

```
<ul>
    @Repeat(pets, 3, @<li>@item.Name</li>)
</ul>
```

Rendered output:

```
<ul>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>K-9</li>
    <li>K-9</li>
    <li>K-9</li>
</ul>
```

Tag Helpers

This scenario only applies to MVC views and Razor Pages (.cshtml).

There are three directives that pertain to Tag Helpers.

DIRECTIVE	FUNCTION
`@addTagHelper`	Makes Tag Helpers available to a view.
`@removeTagHelper`	Removes Tag Helpers previously added from a view.

DIRECTIVE	FUNCTION
`@tagHelperPrefix`	Specifies a tag prefix to enable Tag Helper support and to make Tag Helper usage explicit.

Razor reserved keywords

Razor keywords

- `page` (Requires ASP.NET Core 2.1 or later)
- `namespace`
- `functions`
- `inherits`
- `model`
- `section`
- `helper` (Not currently supported by ASP.NET Core)

Razor keywords are escaped with `@(Razor Keyword)` (for example, `@(functions)`).

C# Razor keywords

- `case`
- `do`
- `default`
- `for`
- `foreach`
- `if`
- `else`
- `lock`
- `switch`
- `try`
- `catch`
- `finally`
- `using`
- `while`

C# Razor keywords must be double-escaped with `@(@C# Razor Keyword)` (for example, `@(@case)`). The first `@` escapes the Razor parser. The second `@` escapes the C# parser.

Reserved keywords not used by Razor

- `class`

Inspect the Razor C# class generated for a view

With .NET Core SDK 2.1 or later, the Razor SDK handles compilation of Razor files. When building a project, the Razor SDK generates an *obj/<build_configuration>/<target_framework_moniker>/Razor* directory in the project root. The directory structure within the *Razor* directory mirrors the project's directory structure.

Consider the following directory structure in an ASP.NET Core 2.1 Razor Pages project targeting .NET Core 2.1:

```
Areas/
  Admin/
    Pages/
      Index.cshtml
      Index.cshtml.cs
Pages/
  Shared/
    _Layout.cshtml
  _ViewImports.cshtml
  _ViewStart.cshtml
  Index.cshtml
  Index.cshtml.cs
```

Building the project in *Debug* configuration yields the following *obj* directory:

```
obj/
  Debug/
    netcoreapp2.1/
      Razor/
        Areas/
          Admin/
            Pages/
              Index.g.cshtml.cs
        Pages/
          Shared/
            _Layout.g.cshtml.cs
          _ViewImports.g.cshtml.cs
          _ViewStart.g.cshtml.cs
          Index.g.cshtml.cs
```

To view the generated class for *Pages/Index.cshtml*, open *obj/Debug/netcoreapp2.1/Razor/Pages/Index.g.cshtml.cs*.

Add the following class to the ASP.NET Core MVC project:

```
#if V2
using Microsoft.AspNetCore.Mvc.Razor.Extensions;
using Microsoft.AspNetCore.Razor.Language;

public class CustomTemplateEngine : MvcRazorTemplateEngine
{
    public CustomTemplateEngine(RazorEngine engine, RazorProject project)
        : base(engine, project)
    {
    }

    public override RazorCSharpDocument GenerateCode(RazorCodeDocument codeDocument)
    {
        var csharpDocument = base.GenerateCode(codeDocument);
        var generatedCode = csharpDocument.GeneratedCode;

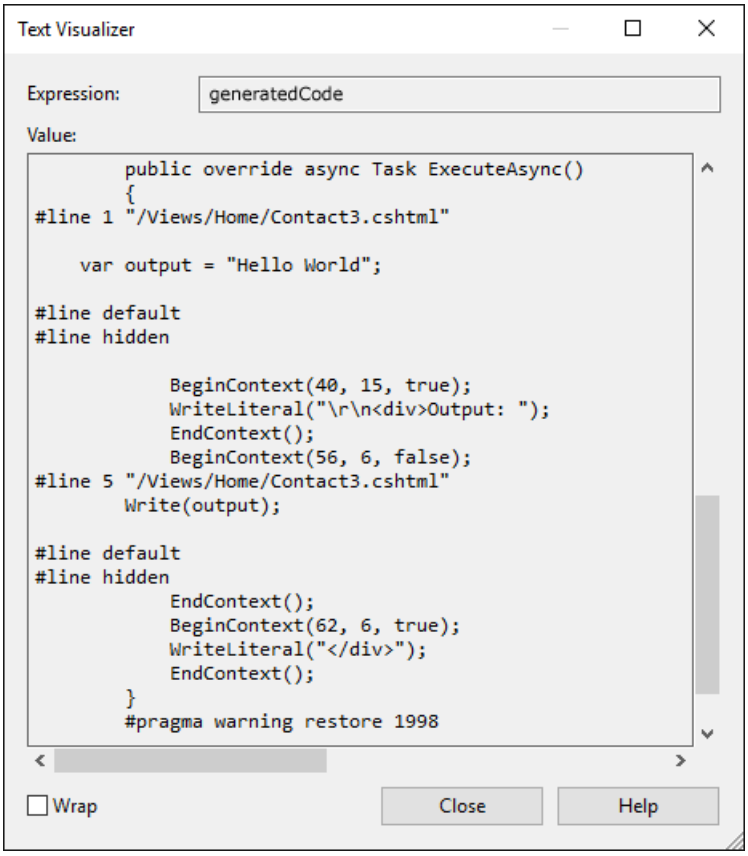
        // Look at generatedCode

        return csharpDocument;
    }
}
#endif
```

In `Startup.ConfigureServices`, override the `RazorTemplateEngine` added by MVC with the `CustomTemplateEngine` class:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
```

Set a breakpoint on the `return csharpDocument;` statement of `CustomTemplateEngine`. When program execution stops at the breakpoint, view the value of `generatedCode`.



View lookups and case sensitivity

The Razor view engine performs case-sensitive lookups for views. However, the actual lookup is determined by the underlying file system:

- File based source:
  - On operating systems with case insensitive file systems (for example, Windows), physical file provider lookups are case insensitive. For example, `return View("Test")` results in matches for */Views/Home/Test.cshtml*, */Views/home/test.cshtml*, and any other casing variant.
  - On case-sensitive file systems (for example, Linux, OSX, and with `EmbeddedFileProvider`), lookups are case-sensitive. For example, `return View("Test")` specifically matches */Views/Home/Test.cshtml*.
- Precompiled views: With ASP.NET Core 2.0 and later, looking up precompiled views is case insensitive on all operating systems. The behavior is identical to physical file provider's behavior on Windows. If two precompiled views differ only in case, the result of lookup is non-deterministic.

Developers are encouraged to match the casing of file and directory names to the casing of:

- Area, controller, and action names.
- Razor Pages.

Matching case ensures the deployments find their views regardless of the underlying file system.

Additional resources

Introduction to ASP.NET Web Programming Using the Razor Syntax provides many samples of programming with Razor syntax.

Create reusable UI using the Razor class library project in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

Razor views, pages, controllers, page models, Razor components, View components, and data models can be built into a Razor class library (RCL). The RCL can be packaged and reused. Applications can include the RCL and override the views and pages it contains. When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (*.cshtml* file) in the web app takes precedence.

View or download sample code (how to download)

Create a class library containing Razor UI

- Visual Studio
- .NET Core CLI

- From Visual Studio select **Create new a new project**.
- Select **Razor Class Library** > **Next**.
- Name the library (for example, "RazorClassLib"), > **Create**. To avoid a file name collision with the generated view library, ensure the library name doesn't end in `.Views`.
- Select **Support pages and views** if you need to support views. By default, only Razor Pages are supported. Select **Create**.

The Razor class library (RCL) template defaults to Razor component development by default. The **Support pages and views** option supports pages and views.

Add Razor files to the RCL.

The ASP.NET Core templates assume the RCL content is in the *Areas* folder. See RCL Pages layout to create an RCL that exposes content in `~/Pages` rather than `~/Areas/Pages`.

Reference RCL content

The RCL can be referenced by:

- NuGet package. See Creating NuGet packages and dotnet add package and Create and publish a NuGet package.
- *{ProjectName}.csproj*. See dotnet-add reference.

Override views, partial views, and pages

When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (*.cshtml* file) in the web app takes precedence. For example, add *WebApp1/Areas/MyFeature/Pages/Page1.cshtml* to WebApp1, and Page1 in the WebApp1 will take precedence over Page1 in the RCL.

In the sample download, rename *WebApp1/Areas/MyFeature2* to *WebApp1/Areas/MyFeature* to test precedence.

Copy the *RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml* partial view to

WebApp1/Areas/MyFeature/Pages/Shared/_Message.cshtml. Update the markup to indicate the new location. Build and run the app to verify the app's version of the partial is being used.

RCL Pages layout

To reference RCL content as though it is part of the web app's *Pages* folder, create the RCL project with the following file structure:

- *RazorUIClassLib/Pages*
- *RazorUIClassLib/Pages/Shared*

Suppose *RazorUIClassLib/Pages/Shared* contains two partial files: *_Header.cshtml* and *_Footer.cshtml*. The `<partial>` tags could be added to *_Layout.cshtml* file:

```
<body>
  <partial name="_Header">
  @RenderBody()
  <partial name="_Footer">
</body>
```

Create an RCL with static assets

An RCL may require companion static assets that can be referenced by either the RCL or the consuming app of the RCL. ASP.NET Core allows creating RCLs that include static assets that are available to a consuming app.

To include companion assets as part of an RCL, create a *wwwroot* folder in the class library and include any required files in that folder.

When packing an RCL, all companion assets in the *wwwroot* folder are automatically included in the package.

Use the `dotnet pack` command rather than the NuGet.exe version `nuget pack`.

Exclude static assets

To exclude static assets, add the desired exclusion path to the `$(DefaultItemExcludes)` property group in the project file. Separate entries with a semicolon (`;`).

In the following example, the *lib.css* stylesheet in the *wwwroot* folder isn't considered a static asset and isn't included in the published RCL:

```
<PropertyGroup>
  <DefaultItemExcludes>$(DefaultItemExcludes);wwwroot\lib.css</DefaultItemExcludes>
</PropertyGroup>
```

Typescript integration

To include TypeScript files in an RCL:

1. Place the TypeScript files (*.ts*) outside of the *wwwroot* folder. For example, place the files in a *Client* folder.
2. Configure the TypeScript build output for the *wwwroot* folder. Set the `TypescriptOutDir` property inside of a `PropertyGroup` in the project file:

   ```
   <TypescriptOutDir>wwwroot</TypescriptOutDir>
   ```

3. Include the TypeScript target as a dependency of the `ResolveCurrentProjectStaticWebAssets` target by adding the following target inside of a `PropertyGroup` in the project file:

```
<ResolveCurrentProjectStaticWebAssetsInputsDependsOn>
  CompileTypeScript;
  $(ResolveCurrentProjectStaticWebAssetsInputs)
</ResolveCurrentProjectStaticWebAssetsInputsDependsOn>
```

Consume content from a referenced RCL

The files included in the *wwwroot* folder of the RCL are exposed to either the RCL or the consuming app under the prefix `_content/{LIBRARY NAME}/`. For example, a library named *Razor.Class.Lib* results in a path to static content at `_content/Razor.Class.Lib/`. When producing a NuGet package and the assembly name isn't the same as the package ID, use the package ID for `{LIBRARY NAME}`.

The consuming app references static assets provided by the library with `<script>`, `<style>`, `<img>`, and other HTML tags. The consuming app must have static file support enabled in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    ...

    app.UseStaticFiles();

    ...
}
```

When running the consuming app from build output (`dotnet run`), static web assets are enabled by default in the Development environment. To support assets in other environments when running from build output, call `UseStaticWebAssets` on the host builder in *Program.cs*:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStaticWebAssets();
                webBuilder.UseStartup<Startup>();
            });
}
```

Calling `UseStaticWebAssets` isn't required when running an app from published output (`dotnet publish`).

Multi-project development flow

When the consuming app runs:

- The assets in the RCL stay in their original folders. The assets aren't moved to the consuming app.
- Any change within the RCL's *wwwroot* folder is reflected in the consuming app after the RCL is rebuilt and without rebuilding the consuming app.

When the RCL is built, a manifest is produced that describes the static web asset locations. The consuming app reads the manifest at runtime to consume the assets from referenced projects and packages. When a new asset is added to an RCL, the RCL must be rebuilt to update its manifest before a consuming app can access the new

asset.

Publish

When the app is published, the companion assets from all referenced projects and packages are copied into the *wwwroot* folder of the published app under `_content/{LIBRARY NAME}/`.

Razor views, pages, controllers, page models, Razor components, View components, and data models can be built into a Razor class library (RCL). The RCL can be packaged and reused. Applications can include the RCL and override the views and pages it contains. When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (*.cshtml* file) in the web app takes precedence.

View or download sample code (how to download)

Create a class library containing Razor UI

- Visual Studio
- .NET Core CLI

- From the Visual Studio **File** menu, select **New** > **Project**.
- Select **ASP.NET Core Web Application**.
- Name the library (for example, "RazorClassLib") > **OK**. To avoid a file name collision with the generated view library, ensure the library name doesn't end in `.Views`.
- Verify **ASP.NET Core 2.1** or later is selected.
- Select **Razor Class Library** > **OK**.

An RCL has the following project file:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  </ItemGroup>

</Project>
```

Add Razor files to the RCL.

The ASP.NET Core templates assume the RCL content is in the *Areas* folder. See RCL Pages layout to create an RCL that exposes content in `~/Pages` rather than `~/Areas/Pages`.

Reference RCL content

The RCL can be referenced by:

- NuGet package. See Creating NuGet packages and dotnet add package and Create and publish a NuGet package.
- *{ProjectName}.csproj*. See dotnet-add reference.

Walkthrough: Create an RCL project and use from a Razor Pages project

You can download the complete project and test it rather than creating it. The sample download contains

additional code and links that make the project easy to test. You can leave feedback in this GitHub issue with your comments on download samples versus step-by-step instructions.

Test the download app

If you haven't downloaded the completed app and would rather create the walkthrough project, skip to the next section.

- Visual Studio
- .NET Core CLI

Open the *.sln* file in Visual Studio. Run the app.

Follow the instructions in Test WebApp1

Create an RCL

In this section, an RCL is created. Razor files are added to the RCL.

- Visual Studio
- .NET Core CLI

Create the RCL project:

- From the Visual Studio **File** menu, select **New** > **Project**.
- Select **ASP.NET Core Web Application**.
- Name the app **RazorUIClassLib** > **OK**.
- Verify **ASP.NET Core 2.1** or later is selected.
- Select **Razor Class Library** > **OK**.
- Add a Razor partial view file named *RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml*.

Add Razor files and folders to the project

- Replace the markup in *RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml* with the following code:

```
<h3>_Message.cshtml partial view.</h3>

<p>RazorUIClassLib\Areas\MyFeature\Pages\Shared\_Message.cshtml</p>
```

- Replace the markup in *RazorUIClassLib/Areas/MyFeature/Pages/Page1.cshtml* with the following code:

```
@page
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h2>Hello from a Razor UI class library!</h2>
<p> From  RazorUIClassLib\Areas\MyFeature\Pages\Page1.cshtml</p>

<partial name="_Message" />
```

`@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers` is required to use the partial view (`<partial name="_Message" />`). Rather than including the `@addTagHelper` directive, you can add a *_ViewImports.cshtml* file. For example:

```
dotnet new viewimports -o RazorUIClassLib/Areas/MyFeature/Pages
```

For more information on *_ViewImports.cshtml*, see Importing Shared Directives

- Build the class library to verify there are no compiler errors:

```
dotnet build RazorUIClassLib
```

The build output contains *RazorUIClassLib.dll* and *RazorUIClassLib.Views.dll*. *RazorUIClassLib.Views.dll* contains the compiled Razor content.

Use the Razor UI library from a Razor Pages project

- Visual Studio
- .NET Core CLI

Create the Razor Pages web app:

- From **Solution Explorer**, right-click the solution > **Add** > **New Project**.
- Select **ASP.NET Core Web Application**.
- Name the app **WebApp1**.
- Verify **ASP.NET Core 2.1** or later is selected.
- Select **Web Application** > **OK**.
- From **Solution Explorer**, right-click on **WebApp1** and select **Set as StartUp Project**.
- From **Solution Explorer**, right-click on **WebApp1** and select **Build Dependencies** > **Project Dependencies**.
- Check **RazorUIClassLib** as a dependency of **WebApp1**.
- From **Solution Explorer**, right-click on **WebApp1** and select **Add** > **Reference**.
- In the **Reference Manager** dialog, check **RazorUIClassLib** > **OK**.

Run the app.

Test WebApp1

Browse to `/MyFeature/Page1` to verify that the Razor UI class library is in use.

Override views, partial views, and pages

When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (*.cshtml* file) in the web app takes precedence. For example, add *WebApp1/Areas/MyFeature/Pages/Page1.cshtml* to WebApp1, and Page1 in the WebApp1 will take precedence over Page1 in the RCL.

In the sample download, rename *WebApp1/Areas/MyFeature2* to *WebApp1/Areas/MyFeature* to test precedence.

Copy the *RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml* partial view to *WebApp1/Areas/MyFeature/Pages/Shared/_Message.cshtml*. Update the markup to indicate the new location. Build and run the app to verify the app's version of the partial is being used.

RCL Pages layout

To reference RCL content as though it is part of the web app's *Pages* folder, create the RCL project with the following file structure:

- *RazorUIClassLib/Pages*
- *RazorUIClassLib/Pages/Shared*

Suppose *RazorUIClassLib/Pages/Shared* contains two partial files: *_Header.cshtml* and *_Footer.cshtml*. The `<partial>` tags could be added to *_Layout.cshtml* file:

```
<body>
  <partial name="_Header">
  @RenderBody()
  <partial name="_Footer">
</body>
```

Additional resources

- ASP.NET Core Razor components class libraries

ASP.NET Core built-in Tag Helpers

9/22/2020 • 2 minutes to read • Edit Online

By Peter Kellner

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

There are built-in Tag Helpers which aren't listed in this document. The unlisted Tag Helpers are used internally by the Razor view engine. The Tag Helper for the `~` (tilde) character is unlisted. The tilde Tag Helper expands to the root path of the website.

Built-in ASP.NET Core Tag Helpers

Anchor Tag Helper

Cache Tag Helper

Component Tag Helper

Distributed Cache Tag Helper

Environment Tag Helper

Form Tag Helper

Form Action Tag Helper

Image Tag Helper

Input Tag Helper

Label Tag Helper

Link Tag Helper

Partial Tag Helper

Script Tag Helper

Select Tag Helper

Textarea Tag Helper

Validation Message Tag Helper

Validation Summary Tag Helper

Additional resources

- Tag Helpers in ASP.NET Core
- Tag Helper Components in ASP.NET Core

Tag Helpers in ASP.NET Core

9/22/2020 • 12 minutes to read • Edit Online

By Rick Anderson

What are Tag Helpers

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. For example, the built-in `ImageTagHelper` can append a version number to the image name. Whenever the image changes, the server generates a new unique version for the image, so clients are guaranteed to get the current image (instead of a stale cached image). There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LabelTagHelper` can target the HTML `<label>` element when the `LabelTagHelper` attributes are applied. If you're familiar with HTML Helpers, Tag Helpers reduce the explicit transitions between HTML and C# in Razor views. In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. Tag Helpers compared to HTML Helpers explains the differences in more detail.

What Tag Helpers provide

An HTML-friendly development experience For the most part, Razor markup using Tag Helpers looks like standard HTML. Front-end designers conversant with HTML/CSS/JavaScript can edit Razor without learning C# Razor syntax.

A rich IntelliSense environment for creating HTML and Razor markup This is in sharp contrast to HTML Helpers, the previous approach to server-side creation of markup in Razor views. Tag Helpers compared to HTML Helpers explains the differences in more detail. IntelliSense support for Tag Helpers explains the IntelliSense environment. Even developers experienced with Razor C# syntax are more productive using Tag Helpers than writing C# Razor markup.

A way to make you more productive and able to produce more robust, reliable, and maintainable code using information only available on the server For example, historically the mantra on updating images was to change the name of the image when you change the image. Images should be aggressively cached for performance reasons, and unless you change the name of an image, you risk clients getting a stale copy. Historically, after an image was edited, the name had to be changed and each reference to the image in the web app needed to be updated. Not only is this very labor intensive, it's also error prone (you could miss a reference, accidentally enter the wrong string, etc.) The built-in `ImageTagHelper` can do this for you automatically. The `ImageTagHelper` can append a version number to the image name, so whenever the image changes, the server automatically generates a new unique version for the image. Clients are guaranteed to get the current image. This robustness and labor savings comes essentially free by using the `ImageTagHelper`.

Most built-in Tag Helpers target standard HTML elements and provide server-side attributes

for the element. For example, the `<input>` element used in many views in the *Views/Account* folder contains the `asp-for` attribute. This attribute extracts the name of the specified model property into the rendered HTML. Consider a Razor view with the following model:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

The following Razor markup:

```
<label asp-for="Movie.Title"></label>
```

Generates the following HTML:

```
<label for="Movie_Title">Title</label>
```

The `asp-for` attribute is made available by the `For` property in the LabelTagHelper. See Author Tag Helpers for more information.

Managing Tag Helper scope

Tag Helpers scope is controlled by a combination of `@addTagHelper`, `@removeTagHelper`, and the "!" opt-out character.

`@addTagHelper` makes Tag Helpers available

If you create a new ASP.NET Core web app named *AuthoringTagHelpers*, the following *Views/_ViewImports.cshtml* file will be added to your project:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

The `@addTagHelper` directive makes Tag Helpers available to the view. In this case, the view file is *Pages/_ViewImports.cshtml*, which by default is inherited by all files in the *Pages* folder and subfolders; making Tag Helpers available. The code above uses the wildcard syntax ("*") to specify that all Tag Helpers in the specified assembly (*Microsoft.AspNetCore.Mvc.TagHelpers*) will be available to every view file in the *Views* directory or subdirectory. The first parameter after `@addTagHelper` specifies the Tag Helpers to load (we are using "*" for all Tag Helpers), and the second parameter "Microsoft.AspNetCore.Mvc.TagHelpers" specifies the assembly containing the Tag Helpers. *Microsoft.AspNetCore.Mvc.TagHelpers* is the assembly for the built-in ASP.NET Core Tag Helpers.

To expose all of the Tag Helpers in this project (which creates an assembly named *AuthoringTagHelpers*), you would use the following:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

If your project contains an `EmailTagHelper` with the default namespace (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), you can provide the fully qualified name (FQN) of the Tag Helper:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

To add a Tag Helper to a view using an FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (*AuthoringTagHelpers*). Most developers prefer to use the "*" wildcard syntax. The wildcard syntax allows you to insert the wildcard character "*" as the suffix in an FQN. For example, any of the following directives will bring in the `EmailTagHelper` :

```
@addTagHelper AuthoringTagHelpers.TagHelpers.E*, AuthoringTagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.Email*, AuthoringTagHelpers
```

As mentioned previously, adding the `@addTagHelper` directive to the *Views/_ViewImports.cshtml* file makes the Tag Helper available to all view files in the *Views* directory and subdirectories. You can use the `@addTagHelper` directive in specific view files if you want to opt-in to exposing the Tag Helper to only those views.

`@removeTagHelper` removes Tag Helpers

The `@removeTagHelper` has the same two parameters as `@addTagHelper` , and it removes a Tag Helper that was previously added. For example, `@removeTagHelper` applied to a specific view removes the specified Tag Helper from the view. Using `@removeTagHelper` in a *Views/Folder/_ViewImports.cshtml* file removes the specified Tag Helper from all of the views in *Folder*.

Controlling Tag Helper scope with the *_ViewImports.cshtml* file

You can add a *_ViewImports.cshtml* to any view folder, and the view engine applies the directives from both that file and the *Views/_ViewImports.cshtml* file. If you added an empty *Views/Home/_ViewImports.cshtml* file for the *Home* views, there would be no change because the *_ViewImports.cshtml* file is additive. Any `@addTagHelper` directives you add to the *Views/Home/_ViewImports.cshtml* file (that are not in the default *Views/_ViewImports.cshtml* file) would expose those Tag Helpers to views only in the *Home* folder.

Opting out of individual elements

You can disable a Tag Helper at the element level with the Tag Helper opt-out character ("!"). For example, `Email` validation is disabled in the `<span>` with the Tag Helper opt-out character:

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

You must apply the Tag Helper opt-out character to the opening and closing tag. (The Visual Studio editor automatically adds the opt-out character to the closing tag when you add one

to the opening tag). After you add the opt-out character, the element and Tag Helper attributes are no longer displayed in a distinctive font.

Using `@tagHelperPrefix` to make Tag Helper usage explicit

The `@tagHelperPrefix` directive allows you to specify a tag prefix string to enable Tag Helper support and to make Tag Helper usage explicit. For example, you could add the following markup to the *Views/_ViewImports.cshtml* file:

```
@tagHelperPrefix th:
```

In the code image below, the Tag Helper prefix is set to `th:`, so only those elements using the prefix `th:` support Tag Helpers (Tag Helper-enabled elements have a distinctive font). The `<label>` and `<input>` elements have the Tag Helper prefix and are Tag Helper-enabled, while the `<span>` element doesn't.

```
<div class="form-group">
    <th:label asp-for="Password" class="col-md-2"></th:label>
    <div class="col-md-10">
        <th:input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>
</div>
```

The same hierarchy rules that apply to `@addTagHelper` also apply to `@tagHelperPrefix`.

Self-closing Tag Helpers

Many Tag Helpers can't be used as self-closing tags. Some Tag Helpers are designed to be self-closing tags. Using a Tag Helper that was not designed to be self-closing suppresses the rendered output. Self-closing a Tag Helper results in a self-closing tag in the rendered output. For more information, see this note in Authoring Tag Helpers.

C# in Tag Helpers attribute/declaration

Tag Helpers do not allow C# in the element's attribute or tag declaration area. For example, the following code is not valid:

```
<input asp-for="LastName"
       @(Model?.LicenseId == null ? "disabled" : string.Empty) />
```

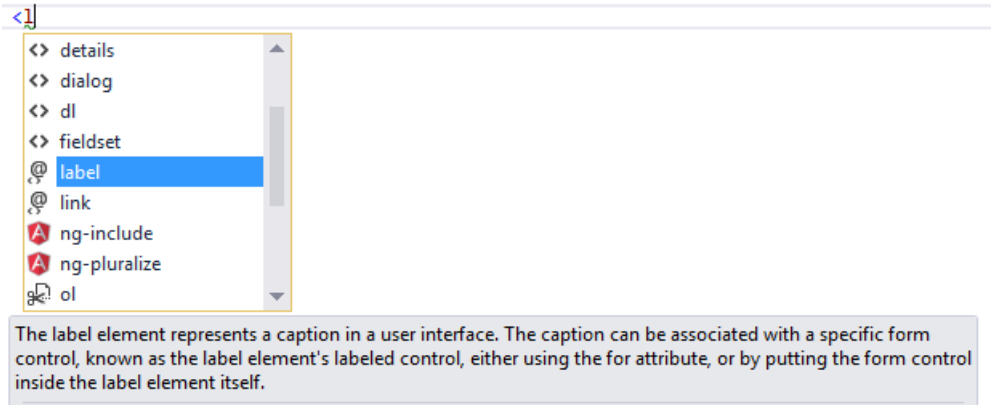
The preceding code can be written as:

```
<input asp-for="LastName"
       disabled="@(Model?.LicenseId == null)" />
```

IntelliSense support for Tag Helpers

When you create a new ASP.NET Core web app in Visual Studio, it adds the NuGet package "Microsoft.AspNetCore.Razor.Tools". This is the package that adds Tag Helper tooling.

Consider writing an HTML `<label>` element. As soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



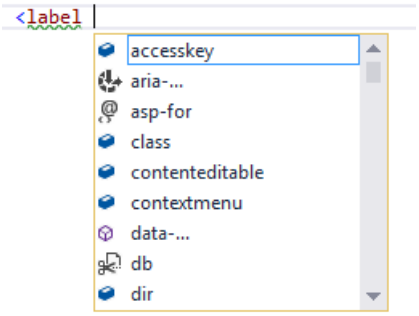
Not only do you get HTML help, but the icon (the "@" symbol with "<>" under it).

identifies the element as targeted by Tag Helpers. Pure HTML elements (such as the `fieldset`) display the "<>" icon.

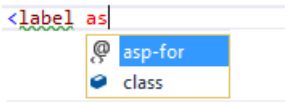
A pure HTML `<label>` tag displays the HTML tag (with the default Visual Studio color theme) in a brown font, the attributes in red, and the attribute values in blue.

```
<label class="col-md-2">Email</label>
```

After you enter `<label`, IntelliSense lists the available HTML/CSS attributes and the Tag Helper-targeted attributes:



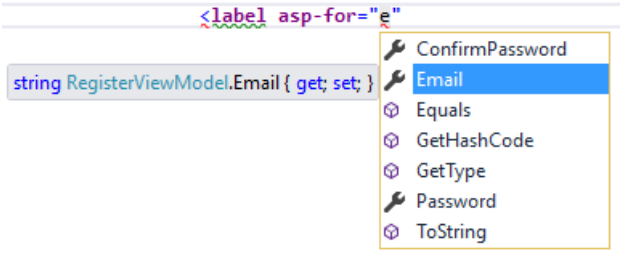
IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:



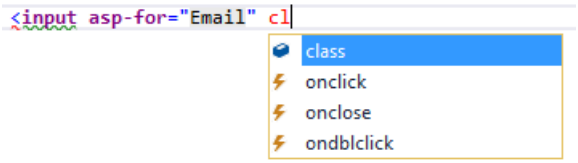
As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio "Blue" or "Light" color theme, the font is bold purple. If you're using the "Dark" theme the font is bold teal. The images in this document were taken using the default theme.

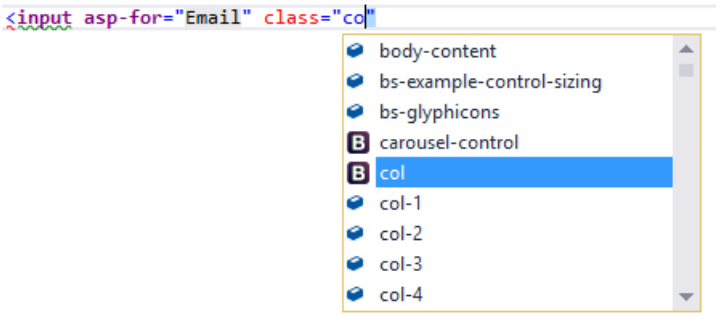
```
<label asp-for
```

You can enter the Visual Studio *CompleteWord* shortcut (Ctrl +spacebar is the default inside the double quotes (""), and you are now in C#, just like you would be in a C# class. IntelliSense displays all the methods and properties on the page model. The methods and properties are available because the property type is `ModelExpression`. In the image below, I'm editing the `Register` view, so the `RegisterViewModel` is available.



IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:





Tag Helpers compared to HTML Helpers

Tag Helpers attach to HTML elements in Razor views, while HTML Helpers are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class "caption":

```
@Html.Label("FirstName", "First Name:", new {@class="caption"})
```

The at (`@`) symbol tells Razor this is the start of code. The next two parameters ("FirstName" and "First Name:") are strings, so IntelliSense can't help. The last argument:

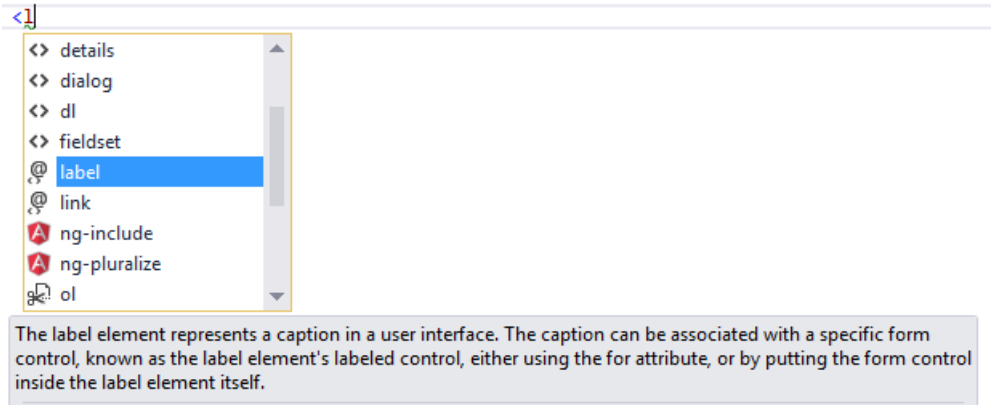
```
new {@class="caption"}
```

Is an anonymous object used to represent attributes. Because `class` is a reserved keyword in C#, you use the `@` symbol to force C# to interpret `@class=` as a symbol (property name). To a front-end designer (someone familiar with HTML/CSS/JavaScript and other client technologies but not familiar with C# and Razor), most of the line is foreign. The entire line must be authored with no help from IntelliSense.

Using the `LabelTagHelper`, the same markup can be written as:

```
<label class="caption" asp-for="FirstName"></label>
```

With the Tag Helper version, as soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



IntelliSense helps you write the entire line.

The following code image shows the Form portion of the *Views/Account/Register.cshtml* Razor view generated from the ASP.NET 4.5.x MVC template included with Visual Studio.

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

The Visual Studio editor displays C# code with a grey background. For example, the `AntiForgeryToken` HTML Helper:

```
@Html.AntiForgeryToken()
```

is displayed with a grey background. Most of the markup in the Register view is C#. Compare that to the equivalent approach using Tag Helpers:

```
<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>
```

The markup is much cleaner and easier to read, edit, and maintain than the HTML Helpers approach. The C# code is reduced to the minimum that the server needs to know about. The Visual Studio editor displays markup targeted by a Tag Helper in a distinctive font.

Consider the *Email* group:

```
<div class="form-group">
    <label asp-for="Email" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
</div>
```

Each of the "asp-" attributes has a value of "Email", but "Email" isn't a string. In this context, "Email" is the C# model expression property for the `RegisterViewModel`.

The Visual Studio editor helps you write **all** of the markup in the Tag Helper approach of the register form, while Visual Studio provides no help for most of the code in the HTML Helpers approach. IntelliSense support for Tag Helpers goes into detail on working with Tag Helpers in the Visual Studio editor.

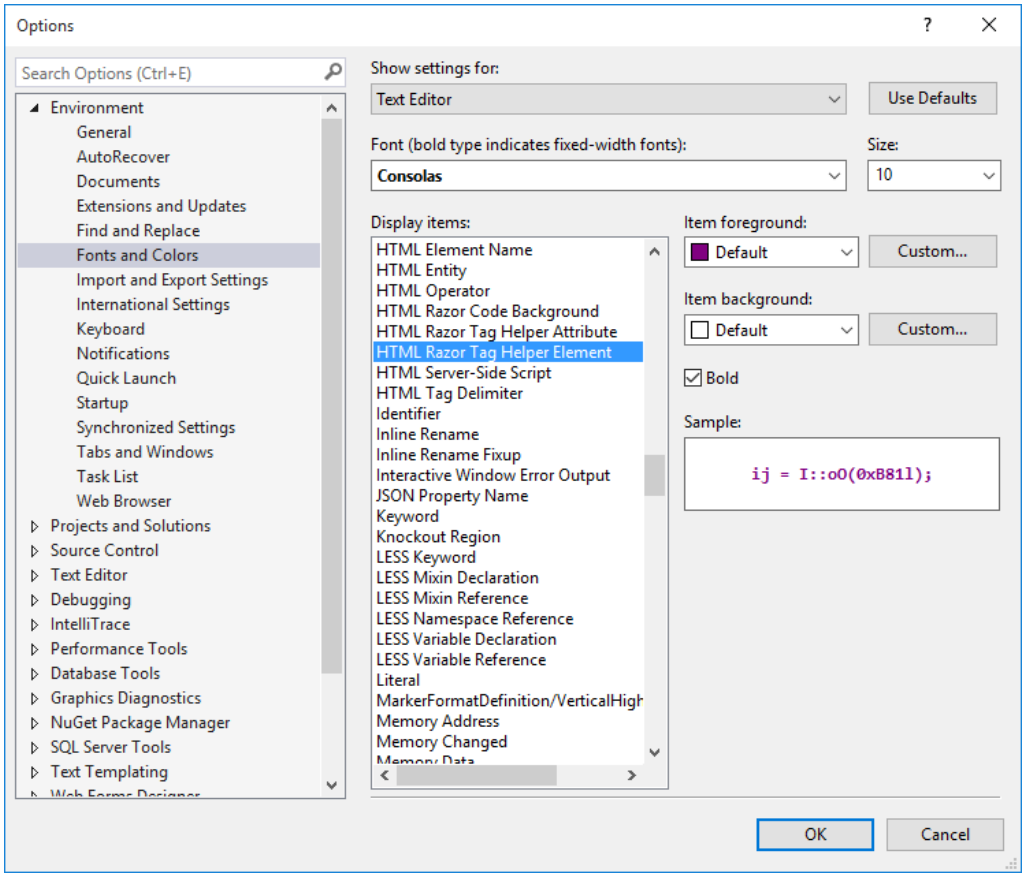
Tag Helpers compared to Web Server Controls

- Tag Helpers don't own the element they're associated with; they simply participate in the rendering of the element and content. ASP.NET /previous-versions/dotnet/netframework-3.0/7698y1f0(v=vs.85) are declared and invoked on a page.
- /previous-versions/zsyt68f1(v=vs.140) have a non-trivial lifecycle that can make developing and debugging difficult.
- Web Server controls allow you to add functionality to the client Document Object Model (DOM) elements by using a client control. Tag Helpers have no DOM.

- Web Server controls include automatic browser detection. Tag Helpers have no knowledge of the browser.
- Multiple Tag Helpers can act on the same element (see Avoiding Tag Helper conflicts) while you typically can't compose Web Server controls.
- Tag Helpers can modify the tag and content of HTML elements that they're scoped to, but don't directly modify anything else on a page. Web Server controls have a less specific scope and can perform actions that affect other parts of your page; enabling unintended side effects.
- Web Server controls use type converters to convert strings into objects. With Tag Helpers, you work natively in C#, so you don't need to do type conversion.
- Web Server controls use System.ComponentModel to implement the run-time and design-time behavior of components and controls. `System.ComponentModel` includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components. Contrast that to Tag Helpers, which typically derive from `TagHelper`, and the `TagHelper` base class exposes only two methods, `Process` and `ProcessAsync`.

Customizing the Tag Helper element font

You can customize the font and colorization from **Tools** > **Options** > **Environment** > **Fonts and Colors**:



Built-in ASP.NET Core Tag Helpers

Anchor Tag Helper

Cache Tag Helper

Component Tag Helper

Distributed Cache Tag Helper

Environment Tag Helper

Form Tag Helper

Form Action Tag Helper

Image Tag Helper

Input Tag Helper

Label Tag Helper

Link Tag Helper

Partial Tag Helper

Script Tag Helper

Select Tag Helper

Textarea Tag Helper

Validation Message Tag Helper

Validation Summary Tag Helper

Additional resources

- Author Tag Helpers
- Working with Forms
- TagHelperSamples on GitHub contains Tag Helper samples for working with Bootstrap.

Author Tag Helpers in ASP.NET Core

9/22/2020 • 16 minutes to read • Edit Online

By Rick Anderson

View or download sample code (how to download)

Get started with Tag Helpers

This tutorial provides an introduction to programming Tag Helpers. Introduction to Tag Helpers describes the benefits that Tag Helpers provide.

A tag helper is any class that implements the `ITagHelper` interface. However, when you author a tag helper, you generally derive from `TagHelper`, doing so gives you access to the `Process` method.

1. Create a new ASP.NET Core project called **AuthoringTagHelpers**. You won't need authentication for this project.
2. Create a folder to hold the Tag Helpers called *TagHelpers*. The *TagHelpers* folder is *not* required, but it's a reasonable convention. Now let's get started writing some simple tag helpers.

A minimal Tag Helper

In this section, you write a tag helper that updates an email tag. For example:

```
<email>Support</email>
```

The server will use our email tag helper to convert that markup into the following:

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

That is, an anchor tag that makes this an email link. You might want to do this if you are writing a blog engine and need it to send email for marketing, support, and other contacts, all to the same domain.

1. Add the following `EmailTagHelper` class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

namespace AuthoringTagHelpers.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.TagName = "a";    // Replaces <email> with <a> tag
        }
    }
}
```

- Tag helpers use a naming convention that targets elements of the root class name (minus the *TagHelper* portion of the class name). In this example, the root name of **EmailTagHelper** is *email*,

so the `<email>` tag will be targeted. This naming convention should work for most tag helpers, later on I'll show how to override it.

- The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides methods and properties for writing Tag Helpers.
- The overridden `Process` method controls what the tag helper does when executed. The `TagHelper` class also provides an asynchronous version (`ProcessAsync`) with the same parameters.
- The context parameter to `Process` (and `ProcessAsync`) contains information associated with the execution of the current HTML tag.
- The output parameter to `Process` (and `ProcessAsync`) contains a stateful HTML element representative of the original source used to generate an HTML tag and content.
- Our class name has a suffix of **TagHelper**, which is *not* required, but it's considered a best practice convention. You could declare the class as:

```
public class Email : TagHelper
```

2. To make the `EmailTagHelper` class available to all our Razor views, add the `addTagHelper` directive to the *Views/_ViewImports.cshtml* file:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

The code above uses the wildcard syntax to specify all the tag helpers in our assembly will be available. The first string after `@addTagHelper` specifies the tag helper to load (Use "*" for all tag helpers), and the second string "AuthoringTagHelpers" specifies the assembly the tag helper is in. Also, note that the second line brings in the ASP.NET Core MVC tag helpers using the wildcard syntax (those helpers are discussed in Introduction to Tag Helpers.) It's the `@addTagHelper` directive that makes the tag helper available to the Razor view. Alternatively, you can provide the fully qualified name (FQN) of a tag helper as shown below:

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper, AuthoringTagHelpers
```

To add a tag helper to a view using a FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the **assembly name** (*AuthoringTagHelpers*, not necessarily the `namespace`). Most developers will prefer to use the wildcard syntax. Introduction to Tag Helpers goes into detail on tag helper adding, removing, hierarchy, and wildcard syntax.

1. Update the markup in the *Views/Home/Contact.cshtml* file with these changes:

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

2. Run the app and use your favorite browser to view the HTML source so you can verify that the email tags are replaced with anchor markup (For example, `<a>Support</a>`). *Support* and *Marketing* are rendered as a links, but they don't have an `href` attribute to make them functional. We'll fix that in the next section.

SetAttribute and SetContent

In this section, we'll update the `EmailTagHelper` so that it will create a valid anchor tag for email. We'll update it to take information from a Razor view (in the form of a `mail-to` attribute) and use that in generating the anchor.

Update the `EmailTagHelper` class with the following:

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // PascalCase gets translated into kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + address);
        output.Content.SetContent(address);
    }
}
```

- Pascal-cased class and property names for tag helpers are translated into their kebab case. Therefore, to use the `MailTo` attribute, you'll use `<email mail-to="value"/>` equivalent.
- The last line sets the completed content for our minimally functional tag helper.
- The highlighted line shows the syntax for adding attributes:

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "a";    // Replaces <email> with <a> tag

    var address = MailTo + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
```

That approach works for the attribute "href" as long as it doesn't currently exist in the attributes collection. You can also use the `output.Attributes.Add` method to add a tag helper attribute to the end of the collection of tag attributes.

1. Update the markup in the *Views/Home/Contact.cshtml* file with these changes:

```
@{
    ViewData["Title"] = "Contact Copy";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way Copy Version <br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email mail-to="Support"></email><br />
    <strong>Marketing:</strong><email mail-to="Marketing"></email>
</address>
```

2. Run the app and verify that it generates the correct links.

NOTE

If you were to write the email tag self-closing (`<email mail-to="Rick" />`), the final output would also be self-closing. To enable the ability to write the tag with only a start tag (`<email mail-to="Rick">`) you must mark the class with the following:

```
[HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
public class EmailVoidTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    // Code removed for brevity
```

With a self-closing email tag helper, the output would be `<a href="mailto:Rick@contoso.com" />`. Self-closing anchor tags are not valid HTML, so you wouldn't want to create one, but you might want to create a tag helper that's self-closing. Tag helpers set the type of the `TagMode` property after reading a tag.

ProcessAsync

In this section, we'll write an asynchronous email helper.

1. Replace the `EmailTagHelper` class with the following code:

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "a";                                   // Replaces <email> with <a> tag
        var content = await output.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + target);
        output.Content.SetContent(target);
    }
}
```

Notes:

- This version uses the asynchronous `ProcessAsync` method. The asynchronous `GetChildContentAsync` returns a `Task` containing the `TagHelperContent`.
- Use the `output` parameter to get contents of the HTML element.

2. Make the following change to the *Views/Home/Contact.cshtml* file so the tag helper can get the target email.

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

3. Run the app and verify that it generates valid email links.

RemoveAll, PreContent.SetHtmlContent and PostContent.SetHtmlContent

1. Add the following `BoldTagHelper` class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = "bold")]
    public class BoldTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            output.Attributes.RemoveAll("bold");
            output.PreContent.SetHtmlContent("<strong>");
            output.PostContent.SetHtmlContent("</strong>");
        }
    }
}
```

- The `[HtmlTargetElement]` attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named "bold" will match, and the `Process` override method in the class will run. In our sample, the `Process` method removes the "bold" attribute and surrounds the containing markup with `<strong></strong>`.
- Because you don't want to replace the existing tag content, you must write the opening `<strong>` tag with the `PreContent.SetHtmlContent` method and the closing `</strong>` tag with the `PostContent.SetHtmlContent` method.

2. Modify the *About.cshtml* view to contain a `bold` attribute value. The completed code is shown below.

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>
```

3. Run the app. You can use your favorite browser to inspect the source and verify the markup.

   The `[HtmlTargetElement]` attribute above only targets HTML markup that provides an attribute name of "bold". The `<bold>` element wasn't modified by the tag helper.

4. Comment out the `[HtmlTargetElement]` attribute line and it will default to targeting `<bold>` tags, that is, HTML markup of the form `<bold>`. Remember, the default naming convention will match the class name **Bold**TagHelper to `<bold>` tags.

5. Run the app and verify that the `<bold>` tag is processed by the tag helper.

Decorating a class with multiple `[HtmlTargetElement]` attributes results in a logical-OR of the targets. For example, using the code below, a bold tag or a bold attribute will match.

```
[HtmlTargetElement("bold")]
[HtmlTargetElement(Attributes = "bold")]
public class BoldTagHelper : TagHelper
{
    public override void Process(TagHelperContext context, TagHelperOutput output)
    {
        output.Attributes.RemoveAll("bold");
        output.PreContent.SetHtmlContent("<strong>");
        output.PostContent.SetHtmlContent("</strong>");
    }
}
```

When multiple attributes are added to the same statement, the runtime treats them as a logical-AND. For example, in the code below, an HTML element must be named "bold" with an attribute named "bold" (`<bold bold />`) to match.

```
[HtmlTargetElement("bold", Attributes = "bold")]
```

You can also use the `[HtmlTargetElement]` to change the name of the targeted element. For example if you wanted the `BoldTagHelper` to target `<MyBold>` tags, you would use the following attribute:

```
[HtmlTargetElement("MyBold")]
```

Pass a model to a Tag Helper

1. Add a *Models* folder.
2. Add the following `WebsiteContext` class to the *Models* folder:

```
using System;

namespace AuthoringTagHelpers.Models
{
    public class WebsiteContext
    {
        public Version Version { get; set; }
        public int CopyrightYear { get; set; }
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

3. Add the following `WebsiteInformationTagHelper` class to the *TagHelpers* folder.

```
using System;
using AuthoringTagHelpers.Models;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
        public WebsiteContext Info { get; set; }

      public override void Process(TagHelperContext context, TagHelperOutput output)
      {
         output.TagName = "section";
         output.Content.SetHtmlContent(
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
<li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
<li><strong>Approved:</strong> {Info.Approved}</li>
<li><strong>Number of tags to show:</strong> {Info.TagsToShow}</li></ul>");
         output.TagMode = TagMode.StartTagAndEndTag;
      }
   }
}
```

- As mentioned previously, tag helpers translates Pascal-cased C# class names and properties for tag helpers into kebab case. Therefore, to use the `WebsiteInformationTagHelper` in Razor, you'll write `<website-information />`.
- You are not explicitly identifying the target element with the `[HtmlTargetElement]` attribute, so the default of `website-information` will be targeted. If you applied the following attribute (note it's not kebab case but matches the class name):

```
[HtmlTargetElement("WebsiteInformation")]
```

The kebab case tag `<website-information />` wouldn't match. If you want use the `[HtmlTargetElement]` attribute, you would use kebab case as shown below:

```
[HtmlTargetElement("Website-Information")]
```

- Elements that are self-closing have no content. For this example, the Razor markup will use a self-closing tag, but the tag helper will be creating a section element (which isn't self-closing and you are writing content inside the `section` element). Therefore, you need to set `TagMode` to `StartTagAndEndTag` to write output. Alternatively, you can comment out the line setting `TagMode` and write markup with a closing tag. (Example markup is provided later in this tutorial.)
- The `$` (dollar sign) in the following line uses an interpolated string:

```
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

4. Add the following markup to the *About.cshtml* view. The highlighted markup displays the web site information.

```
@using AuthoringTagHelpers.Models
@{
    ViewData["Title"] = "About";
    WebsiteContext webContext = new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 };
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="webContext" />
```

> **NOTE**
>
> In the Razor markup shown below:
>
> ```
> <website-information info="webContext" />
> ```
>
> Razor knows the `info` attribute is a class, not a string, and you want to write C# code. Any non-string tag helper attribute should be written without the `@` character.

5. Run the app, and navigate to the About view to see the web site information.

> **NOTE**
>
> You can use the following markup with a closing tag and remove the line with `TagMode.StartTagAndEndTag` in the tag helper:
>
> ```
> <website-information info="webContext" >
> </website-information>
> ```

Condition Tag Helper

The condition tag helper renders output when passed a true value.

1. Add the following `ConditionTagHelper` class to the *TagHelpers* folder.

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = nameof(Condition))]
    public class ConditionTagHelper : TagHelper
    {
        public bool Condition { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput output)
        {
            if (!Condition)
            {
                output.SuppressOutput();
            }
        }
    }
}
```

2. Replace the contents of the *Views/Home/Index.cshtml* file with the following markup:

```
@using AuthoringTagHelpers.Models
@model WebsiteContext

@{
    ViewData["Title"] = "Home Page";
}

<div>
    <h3>Information about our website (outdated):</h3>
    <Website-InforMation info="Model" />
    <div condition="Model.Approved">
        <p>
            This website has <strong surround="em">@Model.Approved</strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>
```

3. Replace the `Index` method in the `Home` controller with the following code:

```
public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}
```

4. Run the app and browse to the home page. The markup in the conditional `div` won't be rendered. Append the query string `?approved=true` to the URL (for example, `http://localhost:1235/Home/Index?approved=true`). `approved` is set to true and the conditional markup will

be displayed.

> **NOTE**
>
> Use the nameof operator to specify the attribute to target rather than specifying a string as you did with the bold tag helper:

```
[HtmlTargetElement(Attributes = nameof(Condition))]
 //   [HtmlTargetElement(Attributes = "condition")]
 public class ConditionTagHelper : TagHelper
{
   public bool Condition { get; set; }

   public override void Process(TagHelperContext context, TagHelperOutput output)
   {
      if (!Condition)
      {
         output.SuppressOutput();
      }
   }
}
```

> The nameof operator will protect the code should it ever be refactored (we might want to change the name to `RedCondition`).

Avoid Tag Helper conflicts

In this section, you write a pair of auto-linking tag helpers. The first will replace markup containing a URL starting with HTTP to an HTML anchor tag containing the same URL (and thus yielding a link to the URL). The second will do the same for a URL starting with WWW.

Because these two helpers are closely related and you may refactor them in the future, we'll keep them in the same file.

1. Add the following `AutoLinkerHttpTagHelper` class to the *TagHelpers* folder.

```
[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent.GetContent(),
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
    }
}
```

> **NOTE**
>
> The `AutoLinkerHttpTagHelper` class targets `p` elements and uses Regex to create the anchor.

2. Add the following markup to the end of the *Views/Home/Contact.cshtml* file:

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

<p>Visit us at http://docs.asp.net or at www.microsoft.com</p>
```

3. Run the app and verify that the tag helper renders the anchor correctly.

4. Update the `AutoLinker` class to include the `AutoLinkerWwwTagHelper` which will convert www text to an anchor tag that also contains the original www text. The updated code is highlighted below:

```
    [HtmlTargetElement("p")]
    public class AutoLinkerHttpTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                 childContent.GetContent(),
                 @"\b(?:https?://)(\S+)\b",
                  "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
        }
    }

    [HtmlTargetElement("p")]
    public class AutoLinkerWwwTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                childContent.GetContent(),
                 @"\b(www\.)(\S+)\b",
                 "<a target=\"_blank\" href=\"http://$0\">$0</a>"));  // www version
        }
    }
}
```

5. Run the app. Notice the www text is rendered as a link but the HTTP text isn't. If you put a break point in both classes, you can see that the HTTP tag helper class runs first. The problem is that the tag helper output is cached, and when the WWW tag helper is run, it overwrites the cached output from the HTTP tag helper. Later in the tutorial we'll see how to control the order that tag helpers run in. We'll fix the code with the following:

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
    }
}

[HtmlTargetElement("p")]
public class AutoLinkerWwwTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(www\.)(\S+)\b",
             "<a target=\"_blank\" href=\"http://$0\">$0</a>"));  // www version
    }
}
```

NOTE

In the first edition of the auto-linking tag helpers, you got the content of the target with the following code:

```
var childContent = await output.GetChildContentAsync();
```

That is, you call `GetChildContentAsync` using the `TagHelperOutput` passed into the `ProcessAsync` method. As mentioned previously, because the output is cached, the last tag helper to run wins. You fixed that problem with the following code:

```
var childContent = output.Content.IsModified ? output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

The code above checks to see if the content has been modified, and if it has, it gets the content from the output buffer.

6. Run the app and verify that the two links work as expected. While it might appear our auto linker tag helper is correct and complete, it has a subtle problem. If the WWW tag helper runs first, the www links won't be correct. Update the code by adding the `Order` overload to control the order that the tag runs in. The `Order` property determines the execution order relative to other tag helpers targeting the same element. The default order value is zero and instances with lower values are executed first.

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    // This filter must run before the AutoLinkerWwwTagHelper as it searches and replaces http and
    // the AutoLinkerWwwTagHelper adds http to the markup.
    public override int Order
    {
        get  {  return int.MinValue;   }
    }
```

The preceding code guarantees that the HTTP tag helper runs before the WWW tag helper. Change `Order` to `MaxValue` and verify that the markup generated for the WWW tag is incorrect.

Inspect and retrieve child content

The tag helpers provide several properties to retrieve content.

- The result of `GetChildContentAsync` can be appended to `output.Content`.
- You can inspect the result of `GetChildContentAsync` with `GetContent`.
- If you modify `output.Content`, the TagHelper body won't be executed or rendered unless you call `GetChildContentAsync` as in our auto-linker sample:

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ? output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link version}
    }
}
```

- Multiple calls to `GetChildContentAsync` returns the same value and doesn't re-execute the `TagHelper` body unless you pass in a false parameter indicating not to use the cached result.

Load minified partial view TagHelper

In production environments, performance can be improved by loading minified partial views. To take advantage of minified partial view in production:

- Create/set up a pre-build process that minifies partial views.
- Use the following code to load minified partial views in non-development environments.

```
public class MinifiedVersionPartialTagHelper : PartialTagHelper
    {
        public MinifiedVersionPartialTagHelper(ICompositeViewEngine viewEngine,
                                IViewBufferScope viewBufferScope)
                               : base(viewEngine, viewBufferScope)
        {

        }

        public override Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
        {
            // Append ".min" to load the minified partial view.
            if (!IsDevelopment())
            {
                Name += ".min";
            }

            return base.ProcessAsync(context, output);
        }

        private bool IsDevelopment()
        {
            return Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")
                                                == EnvironmentName.Development;
        }
    }
```

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

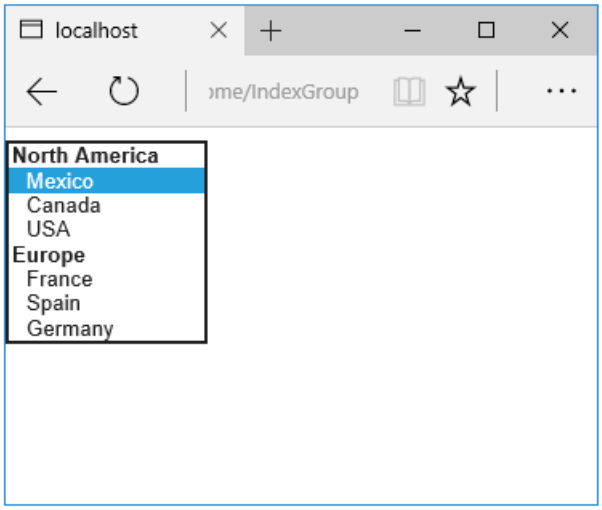
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Tag Helper Components in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

By Scott Addie and Fiyaz Bin Hasan

A Tag Helper Component is a Tag Helper that allows you to conditionally modify or add HTML elements from server-side code. This feature is available in ASP.NET Core 2.0 or later.

ASP.NET Core includes two built-in Tag Helper Components: `head` and `body`. They're located in the Microsoft.AspNetCore.Mvc.Razor.TagHelpers namespace and can be used in both MVC and Razor Pages. Tag Helper Components don't require registration with the app in *_ViewImports.cshtml*.

View or download sample code (how to download)

Use cases

Two common use cases of Tag Helper Components include:

1. Injecting a `<link>` into the `<head>`.
2. Injecting a `<script>` into the `<body>`.

The following sections describe these use cases.

Inject into HTML head element

Inside the HTML `<head>` element, CSS files are commonly imported with the HTML `<link>` element. The following code injects a `<link>` element into the `<head>` element using the `head` Tag Helper Component:

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressStyleTagHelperComponent : TagHelperComponent
    {
        private readonly string _style =
            @"<link rel=""stylesheet"" href=""/css/address.css"" />";

        public override int Order => 1;

        public override Task ProcessAsync(TagHelperContext context,
                                          TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "head",
                              StringComparison.OrdinalIgnoreCase))
            {
                output.PostContent.AppendHtml(_style);
            }

            return Task.CompletedTask;
        }
    }
}
```

In the preceding code:

- `AddressStyleTagHelperComponent` implements TagHelperComponent. The abstraction:

    - Allows initialization of the class with a TagHelperContext.
    - Enables the use of Tag Helper Components to add or modify HTML elements.
- The Order property defines the order in which the Components are rendered. `Order` is necessary when there are multiple usages of Tag Helper Components in an app.
- ProcessAsync compares the execution context's TagName property value to `head`. If the comparison evaluates to true, the content of the `_style` field is injected into the HTML `<head>` element.

Inject into HTML body element

The `body` Tag Helper Component can inject a `<script>` element into the `<body>` element. The following code demonstrates this technique:

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressScriptTagHelperComponent : TagHelperComponent
    {
        public override int Order => 2;

        public override async Task ProcessAsync(TagHelperContext context,
                                                TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "body",
                              StringComparison.OrdinalIgnoreCase))
            {
                var script = await File.ReadAllTextAsync(
                    "TagHelpers/Templates/AddressToolTipScript.html");
                output.PostContent.AppendHtml(script);
            }
        }
    }
}
```

A separate HTML file is used to store the `<script>` element. The HTML file makes the code cleaner and more maintainable. The preceding code reads the contents of *TagHelpers/Templates/AddressToolTipScript.html* and appends it with the Tag Helper output. The *AddressToolTipScript.html* file includes the following markup:

```
<script>
$("address[printable]").hover(function() {
    $(this).attr({
        "data-toggle": "tooltip",
        "data-placement": "right",
        "title": "Home of Microsoft!"
    });
});
</script>
```

The preceding code binds a Bootstrap tooltip widget to any `<address>` element that includes a `printable` attribute. The effect is visible when a mouse pointer hovers over the element.

Register a Component

A Tag Helper Component must be added to the app's Tag Helper Components collection. There are three ways to add to the collection:

- Registration via services container

- Registration via Razor file
- Registration via Page Model or controller

Registration via services container

If the Tag Helper Component class isn't managed with ITagHelperComponentManager, it must be registered with the dependency injection (DI) system. The following `Startup.ConfigureServices` code registers the `AddressStyleTagHelperComponent` and `AddressScriptTagHelperComponent` classes with a transient lifetime:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddTransient<ITagHelperComponent,
        AddressScriptTagHelperComponent>();
    services.AddTransient<ITagHelperComponent,
        AddressStyleTagHelperComponent>();
}
```

Registration via Razor file

If the Tag Helper Component isn't registered with DI, it can be registered from a Razor Pages page or an MVC view. This technique is used for controlling the injected markup and the component execution order from a Razor file.

`ITagHelperComponentManager` is used to add Tag Helper Components or remove them from the app. The following code demonstrates this technique with `AddressTagHelperComponent`:

```
@using RazorPagesSample.TagHelpers;
@using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
@inject ITagHelperComponentManager manager;

@{
    string markup;

    if (Model.IsWeekend)
    {
        markup = "<em class='text-warning'>Office closed today!</em>";
    }
    else
    {
        markup = "<em class='text-info'>Office open today!</em>";
    }

    manager.Components.Add(new AddressTagHelperComponent(markup, 1));
}
```

In the preceding code:

- The `@inject` directive provides an instance of `ITagHelperComponentManager`. The instance is assigned to a variable named `manager` for access downstream in the Razor file.
- An instance of `AddressTagHelperComponent` is added to the app's Tag Helper Components collection.

`AddressTagHelperComponent` is modified to accommodate a constructor that accepts the `markup` and `order` parameters:

```
private readonly string _markup;

public override int Order { get; }

public AddressTagHelperComponent(string markup = "", int order = 1)
{
    _markup = markup;
    Order = order;
}
```

The provided `markup` parameter is used in `ProcessAsync` as follows:

```
public override async Task ProcessAsync(TagHelperContext context,
                                        TagHelperOutput output)
{
    if (string.Equals(context.TagName, "address",
            StringComparison.OrdinalIgnoreCase) &&
        output.Attributes.ContainsName("printable"))
    {
        TagHelperContent childContent = await output.GetChildContentAsync();
        string content = childContent.GetContent();
        output.Content.SetHtmlContent(
            $"<div>{content}<br>{_markup}</div>{_printableButton}");
    }
}
```

Registration via Page Model or controller

If the Tag Helper Component isn't registered with DI, it can be registered from a Razor Pages page model or an MVC controller. This technique is useful for separating C# logic from Razor files.

Constructor injection is used to access an instance of `ITagHelperComponentManager`. The Tag Helper Component is added to the instance's Tag Helper Components collection. The following Razor Pages page model demonstrates this technique with `AddressTagHelperComponent`:

```
using System;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesSample.TagHelpers;

public class IndexModel : PageModel
{
    private readonly ITagHelperComponentManager _tagHelperComponentManager;

    public bool IsWeekend
    {
        get
        {
            var dayOfWeek = DateTime.Now.DayOfWeek;

            return dayOfWeek == DayOfWeek.Saturday ||
                   dayOfWeek == DayOfWeek.Sunday;
        }
    }

    public IndexModel(ITagHelperComponentManager tagHelperComponentManager)
    {
        _tagHelperComponentManager = tagHelperComponentManager;
    }

    public void OnGet()
    {
        string markup;

        if (IsWeekend)
        {
            markup = "<em class='text-warning'>Office closed today!</em>";
        }
        else
        {
            markup = "<em class='text-info'>Office open today!</em>";
        }

        _tagHelperComponentManager.Components.Add(
            new AddressTagHelperComponent(markup, 1));
    }
}
```

In the preceding code:

- Constructor injection is used to access an instance of `ITagHelperComponentManager`.
- An instance of `AddressTagHelperComponent` is added to the app's Tag Helper Components collection.

Create a Component

To create a custom Tag Helper Component:

- Create a public class deriving from TagHelperComponentTagHelper.
- Apply an `[HtmlTargetElement]` attribute to the class. Specify the name of the target HTML element.
- *Optional*: Apply an `[EditorBrowsable(EditorBrowsableState.Never)]` attribute to the class to suppress the type's display in IntelliSense.

The following code creates a custom Tag Helper Component that targets the `<address>` HTML element:

```
using System.ComponentModel;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Microsoft.Extensions.Logging;

namespace RazorPagesSample.TagHelpers
{
    [HtmlTargetElement("address")]
    [EditorBrowsable(EditorBrowsableState.Never)]
    public class AddressTagHelperComponentTagHelper : TagHelperComponentTagHelper
    {
        public AddressTagHelperComponentTagHelper(
            ITagHelperComponentManager componentManager,
            ILoggerFactory loggerFactory) : base(componentManager, loggerFactory)
        {
        }
    }
}
```

Use the custom `address` Tag Helper Component to inject HTML markup as follows:

```
public class AddressTagHelperComponent : TagHelperComponent
{
    private readonly string _printableButton =
        "<button type='button' class='btn btn-info' onclick=\"window.open(" +
        "'https://binged.it/2AXRRYw')\">" +
        "<span class='glyphicon glyphicon-road' aria-hidden='true'></span>" +
        "</button>";

    public override int Order => 3;

    public override async Task ProcessAsync(TagHelperContext context,
                                            TagHelperOutput output)
    {
        if (string.Equals(context.TagName, "address",
                StringComparison.OrdinalIgnoreCase) &&
            output.Attributes.ContainsName("printable"))
        {
            var content = await output.GetChildContentAsync();
            output.Content.SetHtmlContent(
                $"<div>{content.GetContent()}</div>{_printableButton}");
        }
    }
}
```

The preceding `ProcessAsync` method injects the HTML provided to SetHtmlContent into the matching `<address>` element. The injection occurs when:

- The execution context's `TagName` property value equals `address`.
- The corresponding `<address>` element has a `printable` attribute.

For example, the `if` statement evaluates to true when processing the following `<address>` element:

```
<address printable>
    One Microsoft Way<br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>
```

Additional resources

- Dependency injection in ASP.NET Core
- Dependency injection into views in ASP.NET Core
- ASP.NET Core built-in Tag Helpers

Anchor Tag Helper in ASP.NET Core

9/22/2020 • 6 minutes to read • Edit Online

By Peter Kellner and Scott Addie

The Anchor Tag Helper enhances the standard HTML anchor (`<a ... ></a>`) tag by adding new attributes. By convention, the attribute names are prefixed with `asp-`. The rendered anchor element's `href` attribute value is determined by the values of the `asp-` attributes.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

View or download sample code (how to download)

SpeakerController is used in samples throughout this document:

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;

public class SpeakerController : Controller
{
    private List<Speaker> Speakers =
        new List<Speaker>
        {
            new Speaker {SpeakerId = 10},
            new Speaker {SpeakerId = 11},
            new Speaker {SpeakerId = 12}
        };

    [Route("Speaker/{id:int}")]
    public IActionResult Detail(int id) =>
        View(Speakers.FirstOrDefault(a => a.SpeakerId == id));

    [Route("/Speaker/Evaluations",
           Name = "speakerevals")]
    public IActionResult Evaluations() => View();

    [Route("/Speaker/EvaluationsCurrent",
           Name = "speakerevalscurrent")]
    public IActionResult Evaluations(
        int speakerId,
        bool currentYear) => View();

    public IActionResult Index() => View(Speakers);
}

public class Speaker
{
    public int SpeakerId { get; set; }
}
```

Anchor Tag Helper attributes

asp-controller

The asp-controller attribute assigns the controller used for generating the URL. The following markup lists all speakers:

```
<a asp-controller="Speaker"
   asp-action="Index">All Speakers</a>
```

The generated HTML:

```
<a href="/Speaker">All Speakers</a>
```

If the `asp-controller` attribute is specified and `asp-action` isn't, the default `asp-action` value is the controller action associated with the currently executing view. If `asp-action` is omitted from the preceding markup, and the Anchor Tag Helper is used in *HomeController*'s *Index* view (*/Home*), the generated HTML is:

```
<a href="/Home">All Speakers</a>
```

asp-action

The asp-action attribute value represents the controller action name included in the generated `href` attribute. The following markup sets the generated `href` attribute value to the speaker evaluations page:

```
<a asp-controller="Speaker"
   asp-action="Evaluations">Speaker Evaluations</a>
```

The generated HTML:

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

If no `asp-controller` attribute is specified, the default controller calling the view executing the current view is used.

If the `asp-action` attribute value is `Index`, then no action is appended to the URL, leading to the invocation of the default `Index` action. The action specified (or defaulted), must exist in the controller referenced in `asp-controller`.

asp-route-{value}

The asp-route-{value} attribute enables a wildcard route prefix. Any value occupying the `{value}` placeholder is interpreted as a potential route parameter. If a default route isn't found, this route prefix is appended to the generated `href` attribute as a request parameter and value. Otherwise, it's substituted in the route template.

Consider the following controller action:

```
public IActionResult AnchorTagHelper(int id)
{
    var speaker = new Speaker
    {
        SpeakerId = id
    };

    return View(speaker);
}
```

With a default route template defined in *Startup.Configure*:

```
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
                    template: "{area:exists}/{controller=Home}/{action=Index}");

    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

The MVC view uses the model, provided by the action, as follows:

```
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-id="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
</body>
</html>
```

The default route's `{id?}` placeholder was matched. The generated HTML:

```
<a href="/Speaker/Detail/12">SpeakerId: 12</a>
```

Assume the route prefix isn't part of the matching routing template, as with the following MVC view:

```
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-speakerid="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
<body>
</html>
```

The following HTML is generated because `speakerid` wasn't found in the matching route:

```
<a href="/Speaker/Detail?speakerid=12">SpeakerId: 12</a>
```

If either `asp-controller` or `asp-action` aren't specified, then the same default processing is followed as is in the `asp-route` attribute.

asp-route

The asp-route attribute is used for creating a URL linking directly to a named route. Using routing attributes, a route can be named as shown in the `SpeakerController` and used in its `Evaluations` action:

```
[Route("/Speaker/Evaluations",
       Name = "speakerevals")]
public IActionResult Evaluations() => View();
```

In the following markup, the `asp-route` attribute references the named route:

```
<a asp-route="speakerevals">Speaker Evaluations</a>
```

The Anchor Tag Helper generates a route directly to that controller action using the URL */Speaker/Evaluations*. The generated HTML:

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

If `asp-controller` or `asp-action` is specified in addition to `asp-route`, the route generated may not be what you expect. To avoid a route conflict, `asp-route` shouldn't be used with the `asp-controller` and `asp-action` attributes.

asp-all-route-data

The asp-all-route-data attribute supports the creation of a dictionary of key-value pairs. The key is the parameter name, and the value is the parameter value.

In the following example, a dictionary is initialized and passed to a Razor view. Alternatively, the data could be passed in with your model.

```
@{
var parms = new Dictionary<string, string>
            {
                { "speakerId", "11" },
                { "currentYear", "true" }
            };
}

<a asp-route="speakerevalscurrent"
   asp-all-route-data="parms">Speaker Evaluations</a>
```

The preceding code generates the following HTML:

```
<a href="/Speaker/EvaluationsCurrent?speakerId=11&currentYear=true">Speaker Evaluations</a>
```

The `asp-all-route-data` dictionary is flattened to produce a querystring meeting the requirements of the overloaded `Evaluations` action:

```
[Route("/Speaker/EvaluationsCurrent",
       Name = "speakerevalscurrent")]
public IActionResult Evaluations(
    int speakerId,
    bool currentYear) => View();
```

If any keys in the dictionary match route parameters, those values are substituted in the route as appropriate. The other non-matching values are generated as request parameters.

asp-fragment

The asp-fragment attribute defines a URL fragment to append to the URL. The Anchor Tag Helper adds the hash character (#). Consider the following markup:

```
<a asp-controller="Speaker"
   asp-action="Evaluations"
   asp-fragment="SpeakerEvaluations">Speaker Evaluations</a>
```

The generated HTML:

```
<a href="/Speaker/Evaluations#SpeakerEvaluations">Speaker Evaluations</a>
```

Hash tags are useful when building client-side apps. They can be used for easy marking and searching in JavaScript, for example.

asp-area

The asp-area attribute sets the area name used to set the appropriate route. The following examples depict how the `asp-area` attribute causes a remapping of routes.

Usage in Razor Pages

Razor Pages areas are supported in ASP.NET Core 2.1 or later.

Consider the following directory hierarchy:

- {Project name}
  - wwwroot
  - Areas
    - Sessions
      - Pages
        - *_ViewStart.cshtml*
        - *Index.cshtml*
        - *Index.cshtml.cs*
  - Pages

The markup to reference the *Sessions* area *Index* Razor Page is:

```
<a asp-area="Sessions"
   asp-page="/Index">View Sessions</a>
```

The generated HTML:

```
<a href="/Sessions">View Sessions</a>
```

TIP

To support areas in a Razor Pages app, do one of the following in `Startup.ConfigureServices`:

- Set the compatibility version to 2.1 or later.
- Set the RazorPagesOptions.AllowAreas property to `true`:

```
services.AddMvc()
        .AddRazorPagesOptions(options => options.AllowAreas = true);
```

Usage in MVC

Consider the following directory hierarchy:

- {Project name}
  - wwwroot
  - Areas

- Blogs
  - Controllers
    - *HomeController.cs*
  - Views
    - Home
      - *AboutBlog.cshtml*
      - *Index.cshtml*
    - *_ViewStart.cshtml*
- Controllers

Setting `asp-area` to "Blogs" prefixes the directory *Areas/Blogs* to the routes of the associated controllers and views for this anchor tag. The markup to reference the *AboutBlog* view is:

```
<a asp-area="Blogs"
   asp-controller="Home"
   asp-action="AboutBlog">About Blog</a>
```

The generated HTML:

```
<a href="/Blogs/Home/AboutBlog">About Blog</a>
```

TIP

To support areas in an MVC app, the route template must include a reference to the area, if it exists. That template is represented by the second parameter of the `routes.MapRoute` method call in *Startup.Configure*:

```
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
                    template: "{area:exists}/{controller=Home}/{action=Index}");

    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

asp-protocol

The asp-protocol attribute is for specifying a protocol (such as `https`) in your URL. For example:

```
<a asp-protocol="https"
   asp-controller="Home"
   asp-action="About">About</a>
```

The generated HTML:

```
<a href="https://localhost/Home/About">About</a>
```

The host name in the example is localhost. The Anchor Tag Helper uses the website's public domain when generating the URL.

asp-host

The asp-host attribute is for specifying a host name in your URL. For example:

```
<a asp-protocol="https"
   asp-host="microsoft.com"
   asp-controller="Home"
   asp-action="About">About</a>
```

The generated HTML:

```
<a href="https://microsoft.com/Home/About">About</a>
```

asp-page

The asp-page attribute is used with Razor Pages. Use it to set an anchor tag's `href` attribute value to a specific page. Prefixing the page name with a forward slash ("/") creates the URL.

The following sample points to the attendee Razor Page:

```
<a asp-page="/Attendee">All Attendees</a>
```

The generated HTML:

```
<a href="/Attendee">All Attendees</a>
```

The `asp-page` attribute is mutually exclusive with the `asp-route`, `asp-controller`, and `asp-action` attributes. However, `asp-page` can be used with `asp-route-{value}` to control routing, as the following markup demonstrates:

```
<a asp-page="/Attendee"
   asp-route-attendeeid="10">View Attendee</a>
```

The generated HTML:

```
<a href="/Attendee?attendeeid=10">View Attendee</a>
```

asp-page-handler

The asp-page-handler attribute is used with Razor Pages. It's intended for linking to specific page handlers.

Consider the following page handler:

```
public void OnGetProfile(int attendeeId)
{
    ViewData["AttendeeId"] = attendeeId;

    // code omitted for brevity
}
```

The page model's associated markup links to the `OnGetProfile` page handler. Note the `On<Verb>` prefix of the page handler method name is omitted in the `asp-page-handler` attribute value. When the method is asynchronous, the `Async` suffix is omitted, too.

```
<a asp-page="/Attendee"
   asp-page-handler="Profile"
   asp-route-attendeeid="12">Attendee Profile</a>
```

The generated HTML:

```
<a href="/Attendee?attendeeid=12&handler=Profile">Attendee Profile</a>
```

Additional resources

- Areas in ASP.NET Core
- Introduction to Razor Pages in ASP.NET Core
- Compatibility version for ASP.NET Core MVC

Cache Tag Helper in ASP.NET Core MVC

9/22/2020 • 4 minutes to read • Edit Online

By Peter Kellner

The Cache Tag Helper provides the ability to improve the performance of your ASP.NET Core app by caching its content to the internal ASP.NET Core cache provider.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

The following Razor markup caches the current date:

```
<cache>@DateTime.Now</cache>
```

The first request to the page that contains the Tag Helper displays the current date. Additional requests show the cached value until the cache expires (default 20 minutes) or until the cached date is evicted from the cache.

Cache Tag Helper Attributes

enabled

ATTRIBUTE TYPE	EXAMPLES	DEFAULT
Boolean	`true`, `false`	`true`

`enabled` determines if the content enclosed by the Cache Tag Helper is cached. The default is `true`. If set to `false`, the rendered output is **not** cached.

Example:

```
<cache enabled="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-on

ATTRIBUTE TYPE	EXAMPLE
`DateTimeOffset`	`@new DateTime(2025,1,29,17,02,0)`

`expires-on` sets an absolute expiration date for the cached item.

The following example caches the contents of the Cache Tag Helper until 5:02 PM on January 29, 2025:

```
<cache expires-on="@new DateTime(2025,1,29,17,02,0)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-after

ATTRIBUTE TYPE	EXAMPLE	DEFAULT
`TimeSpan`	`@TimeSpan.FromSeconds(120)`	20 minutes

`expires-after` sets the length of time from the first request time to cache the contents.

Example:

```
<cache expires-after="@TimeSpan.FromSeconds(120)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

The Razor View Engine sets the default `expires-after` value to twenty minutes.

expires-sliding

ATTRIBUTE TYPE	EXAMPLE
`TimeSpan`	`@TimeSpan.FromSeconds(60)`

Sets the time that a cache entry should be evicted if its value hasn't been accessed.

Example:

```
<cache expires-sliding="@TimeSpan.FromSeconds(60)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-header

ATTRIBUTE TYPE	EXAMPLES
String	`User-Agent`, `User-Agent,content-encoding`

`vary-by-header` accepts a comma-delimited list of header values that trigger a cache refresh when they change.

The following example monitors the header value `User-Agent`. The example caches the content for every different `User-Agent` presented to the web server:

```
<cache vary-by-header="User-Agent">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-query

ATTRIBUTE TYPE	EXAMPLES
String	`Make`, `Make,Model`

`vary-by-query` accepts a comma-delimited list of Keys in a query string (Query) that trigger a cache refresh when the value of any listed key changes.

The following example monitors the values of `Make` and `Model`. The example caches the content for every different `Make` and `Model` presented to the web server:

```
<cache vary-by-query="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-route

ATTRIBUTE TYPE	EXAMPLES
String	`Make`, `Make,Model`

`vary-by-route` accepts a comma-delimited list of route parameter names that trigger a cache refresh when the route data parameter value changes.

Example:

Startup.cs:

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{Make?}/{Model?}");
```

Index.cshtml:

```
<cache vary-by-route="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-cookie

ATTRIBUTE TYPE	EXAMPLES
String	`.AspNetCore.Identity.Application`, `.AspNetCore.Identity.Application,HairColor`

`vary-by-cookie` accepts a comma-delimited list of cookie names that trigger a cache refresh when the cookie values change.

The following example monitors the cookie associated with ASP.NET Core Identity. When a user is authenticated, a change in the Identity cookie triggers a cache refresh:

```
<cache vary-by-cookie=".AspNetCore.Identity.Application">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-user

ATTRIBUTE TYPE	EXAMPLES	DEFAULT
Boolean	`true`, `false`	`true`

`vary-by-user` specifies whether or not the cache resets when the signed-in user (or Context Principal) changes. The current user is also known as the Request Context Principal and can be viewed in a Razor view by referencing `@User.Identity.Name`.

The following example monitors the current logged in user to trigger a cache refresh:

```
<cache vary-by-user="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

Using this attribute maintains the contents in cache through a sign-in and sign-out cycle. When the value is set to `true`, an authentication cycle invalidates the cache for the authenticated user. The cache is invalidated because a new unique cookie value is generated when a user is authenticated. Cache is maintained for the anonymous state when no cookie is present or the cookie has expired. If the user is **not** authenticated, the cache is maintained.

vary-by

ATTRIBUTE TYPE	EXAMPLE
String	`@Model`

`vary-by` allows for customization of what data is cached. When the object referenced by the attribute's string value changes, the content of the Cache Tag Helper is updated. Often, a string-concatenation of model values are assigned to this attribute. Effectively, this results in a scenario where an update to any of the concatenated values invalidates the cache.

The following example assumes the controller method rendering the view sums the integer value of the two route parameters, `myParam1` and `myParam2`, and returns the sum as the single model property. When this sum changes, the content of the Cache Tag Helper is rendered and cached again.

Action:

```
public IActionResult Index(string myParam1, string myParam2, string myParam3)
{
    int num1;
    int num2;
    int.TryParse(myParam1, out num1);
    int.TryParse(myParam2, out num2);
    return View(viewName, num1 + num2);
}
```

Index.cshtml:

```
<cache vary-by="@Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

priority

ATTRIBUTE TYPE	EXAMPLES	DEFAULT
`CacheItemPriority`	`High`, `Low`, `NeverRemove`, `Normal`	`Normal`

`priority` provides cache eviction guidance to the built-in cache provider. The web server evicts `Low` cache entries first when it's under memory pressure.

Example:

```
<cache priority="High">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

The `priority` attribute doesn't guarantee a specific level of cache retention. `CacheItemPriority` is only a suggestion. Setting this attribute to `NeverRemove` doesn't guarantee that cached items are always retained. See the topics in the Additional Resources section for more information.

The Cache Tag Helper is dependent on the memory cache service. The Cache Tag Helper adds the service if it hasn't been added.

Additional resources

- Cache in-memory in ASP.NET Core
- Introduction to Identity on ASP.NET Core

Component Tag Helper in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Daniel Roth and Luke Latham

To render a component from a page or view, use the Component Tag Helper.

Prerequisites

Follow the guidance in the *Prepare the app to use components in pages and views* section of the Integrate ASP.NET Core Razor components into Razor Pages and MVC apps article.

Component Tag Helper

The following Component Tag Helper renders the `Counter` component in a page or view:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}.Pages

...

<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

The preceding example assumes that the `Counter` component is in the app's *Pages* folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample.Pages`).

The Component Tag Helper can also pass parameters to components. Consider the following `ColorfulCheckbox` component that sets the check box label's color and size:

```
<label style="font-size:@(Size)px;color:@Color">
    <input @bind="Value"
           id="survey"
           name="blazor"
           type="checkbox" />
    Enjoying Blazor?
</label>

@code {
    [Parameter]
    public bool Value { get; set; }

    [Parameter]
    public int Size { get; set; } = 8;

    [Parameter]
    public string Color { get; set; }

    protected override void OnInitialized()
    {
        Size += 10;
    }
}
```

The `Size` (`int`) and `Color` (`string`) component parameters can be set by the Component Tag Helper:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}.Shared

...

<component type="typeof(ColorfulCheckbox)" render-mode="ServerPrerendered"
    param-Size="14" param-Color="@("blue")" />
```

The preceding example assumes that the `ColorfulCheckbox` component is in the app's *Shared* folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample.Shared`).

The following HTML is rendered in the page or view:

```
<label style="font-size:24px;color:blue">
    <input id="survey" name="blazor" type="checkbox">
    Enjoying Blazor?
</label>
```

Passing a quoted string requires an explicit Razor expression, as shown for `param-Color` in the preceding example. The Razor parsing behavior for a `string` type value doesn't apply to a `param-*` attribute because the attribute is an `object` type.

The parameter type must be JSON serializable, which typically means that the type must have a default constructor and settable properties. For example, you can specify a value for `Size` and `Color` in the preceding example because the types of `Size` and `Color` are primitive types (`int` and `string`), which are supported by the JSON serializer.

In the following example, a class object is passed to the component:

MyClass.cs:

```
public class MyClass
{
    public MyClass()
    {
    }

    public int MyInt { get; set; } = 999;
    public string MyString { get; set; } = "Initial value";
}
```

The class must have a public parameterless constructor.

Shared/MyComponent.razor:

```
<h2>MyComponent</h2>

<p>Int: @MyObject.MyInt</p>
<p>String: @MyObject.MyString</p>

@code
{
    [Parameter]
    public MyClass MyObject { get; set; }
}
```

Pages/MyPage.cshtml:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}
@using {APP ASSEMBLY}.Shared

...

@{
    var myObject = new MyClass();
    myObject.MyInt = 7;
    myObject.MyString = "Set by MyPage";
}

<component type="typeof(MyComponent)" render-mode="ServerPrerendered"
    param-MyObject="@myObject" />
```

The preceding example assumes that the `MyComponent` component is in the app's *Shared* folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample` and `@using BlazorSample.Shared`). `MyClass` is in the app's namespace.

RenderMode configures whether the component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

RENDER MODE	DESCRIPTION
ServerPrerendered	Renders the component into static HTML and includes a marker for a Blazor Server app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a Blazor Server app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the component into static HTML.

While pages and views can use components, the converse isn't true. Components can't use view- and page-specific features, such as partial views and sections. To use logic from a partial view in a component, factor out the partial view logic into a component.

Rendering server components from a static HTML page isn't supported.

Additional resources

- ComponentTagHelper
- Tag Helpers in ASP.NET Core
- Create and use ASP.NET Core Razor components

Distributed Cache Tag Helper in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Peter Kellner

The Distributed Cache Tag Helper provides the ability to dramatically improve the performance of your ASP.NET Core app by caching its content to a distributed cache source.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

The Distributed Cache Tag Helper inherits from the same base class as the Cache Tag Helper. All of the Cache Tag Helper attributes are available to the Distributed Tag Helper.

The Distributed Cache Tag Helper uses constructor injection. The IDistributedCache interface is passed into the Distributed Cache Tag Helper's constructor. If no concrete implementation of `IDistributedCache` is created in `Startup.ConfigureServices` (*Startup.cs*), the Distributed Cache Tag Helper uses the same in-memory provider for storing cached data as the Cache Tag Helper.

Distributed Cache Tag Helper Attributes

Attributes shared with the Cache Tag Helper

- `enabled`
- `expires-on`
- `expires-after`
- `expires-sliding`
- `vary-by-header`
- `vary-by-query`
- `vary-by-route`
- `vary-by-cookie`
- `vary-by-user`
- `vary-by priority`

The Distributed Cache Tag Helper inherits from the same class as Cache Tag Helper. For descriptions of these attributes, see the Cache Tag Helper.

name

ATTRIBUTE TYPE	EXAMPLE
String	`my-distributed-cache-unique-key-101`

`name` is required. The `name` attribute is used as a key for each stored cache instance. Unlike the Cache Tag Helper that assigns a cache key to each instance based on the Razor page name and location in the Razor page, the Distributed Cache Tag Helper only bases its key on the attribute `name`.

Example:

```
<distributed-cache name="my-distributed-cache-unique-key-101">
    Time Inside Cache Tag Helper: @DateTime.Now
</distributed-cache>
```

Distributed Cache Tag Helper IDistributedCache implementations

There are two implementations of IDistributedCache built in to ASP.NET Core. One is based on SQL Server, and the other is based on Redis. Third-party implementations are also available, such as NCache. Details of these implementations can be found at Distributed caching in ASP.NET Core. Both implementations involve setting an instance of `IDistributedCache` in `Startup`.

There are no tag attributes specifically associated with using any specific implementation of `IDistributedCache`.

Additional resources

- Cache Tag Helper in ASP.NET Core MVC
- Dependency injection in ASP.NET Core
- Distributed caching in ASP.NET Core
- Cache in-memory in ASP.NET Core
- Introduction to Identity on ASP.NET Core

Environment Tag Helper in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Peter Kellner and Hisham Bin Ateya

The Environment Tag Helper conditionally renders its enclosed content based on the current hosting environment. The Environment Tag Helper's single attribute, `names`, is a comma-separated list of environment names. If any of the provided environment names match the current environment, the enclosed content is rendered.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

Environment Tag Helper Attributes

names

`names` accepts a single hosting environment name or a comma-separated list of hosting environment names that trigger the rendering of the enclosed content.

Environment values are compared to the current value returned by IHostingEnvironment.EnvironmentName. The comparison ignores case.

The following example uses an Environment Tag Helper. The content is rendered if the hosting environment is Staging or Production:

```
<environment names="Staging,Production">
    <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

include and exclude attributes

`include` & `exclude` attributes control rendering the enclosed content based on the included or excluded hosting environment names.

include

The `include` property exhibits similar behavior to the `names` attribute. An environment listed in the `include` attribute value must match the app's hosting environment (IHostingEnvironment.EnvironmentName) to render the content of the `<environment>` tag.

```
<environment include="Staging,Production">
    <strong>HostingEnvironment.EnvironmentName is Staging or Production</strong>
</environment>
```

exclude

In contrast to the `include` attribute, the content of the `<environment>` tag is rendered when the hosting environment doesn't match an environment listed in the `exclude` attribute value.

```
<environment exclude="Development">
    <strong>HostingEnvironment.EnvironmentName is not Development</strong>
</environment>
```

Additional resources

- Use multiple environments in ASP.NET Core

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

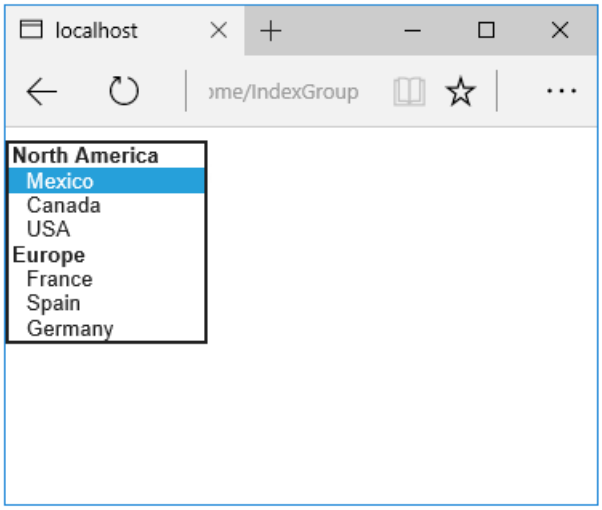
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

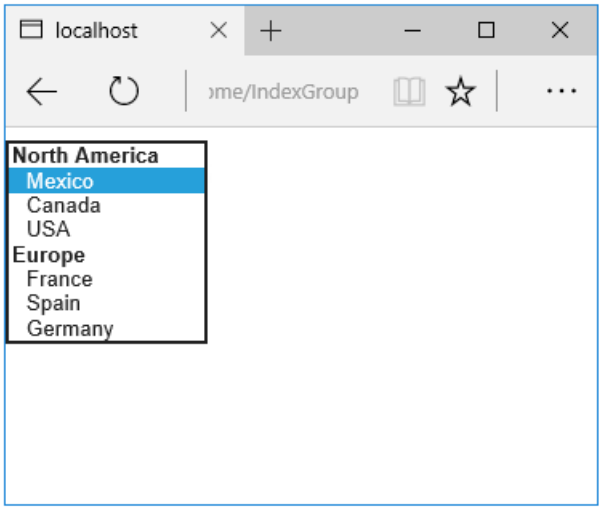
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Image Tag Helper in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Peter Kellner

The Image Tag Helper enhances the `<img>` tag to provide cache-busting behavior for static image files.

A cache-busting string is a unique value representing the hash of the static image file appended to the asset's URL. The unique string prompts clients (and some proxies) to reload the image from the host web server and not from the client's cache.

If the image source (`src`) is a static file on the host web server:

- A unique cache-busting string is appended as a query parameter to the image source.
- If the file on the host web server changes, a unique request URL is generated that includes the updated request parameter.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

Image Tag Helper Attributes

src

To activate the Image Tag Helper, the `src` attribute is required on the `<img>` element.

The image source (`src`) must point to a physical static file on the server. If the `src` is a remote URI, the cache-busting query string parameter isn't generated.

asp-append-version

When `asp-append-version` is specified with a `true` value along with a `src` attribute, the Image Tag Helper is invoked.

The following example uses an Image Tag Helper:

```
<img src="~/images/asplogo.png" asp-append-version="true">
```

If the static file exists in the directory */wwwroot/images/*, the generated HTML is similar to the following (the hash will be different):

```
<img src="/images/asplogo.png?v=Kl_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM">
```

The value assigned to the parameter `v` is the hash value of the *asplogo.png* file on disk. If the web server is unable to obtain read access to the static file, no `v` parameter is added to the `src` attribute in the rendered markup.

Hash caching behavior

The Image Tag Helper uses the cache provider on the local web server to store the calculated `Sha512` hash of a given file. If the file is requested multiple times, the hash isn't recalculated. The cache is invalidated by a file watcher that's attached to the file when the file's `Sha512` hash is calculated. When the file changes on disk, a new hash is calculated and cached.

Additional resources

- Cache in-memory in ASP.NET Core

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

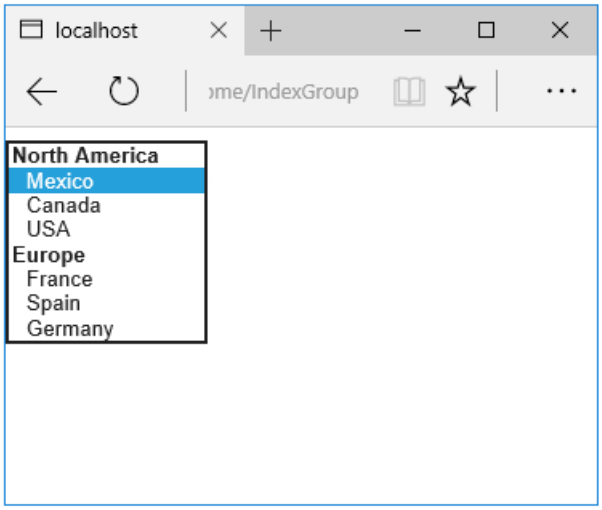
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

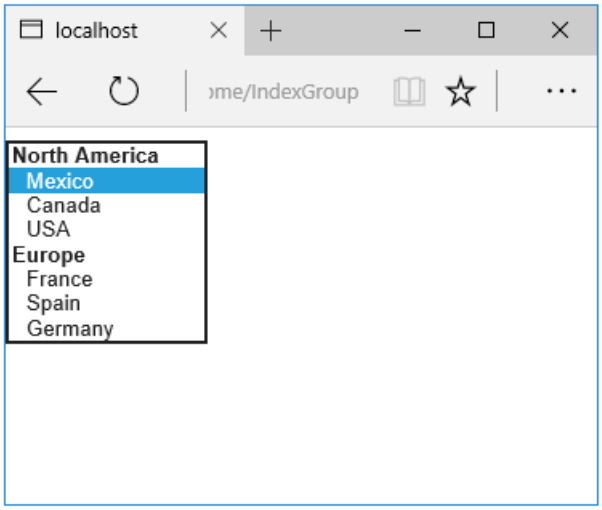
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Link Tag Helper in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Rick Anderson

The Link Tag Helper generates a link to a primary or fall back CSS file. Typically the primary CSS file is on a Content Delivery Network (CDN).

A CDN:

- Provides several performance advantages vs hosting the asset with the web app.
- Should not be relied on as the only source for the asset. CDNs are not always available, therefore a reliable fallback should be used. Typically the fallback is the site hosting the web app.

The Link Tag Helper allows you to specify a CDN for the CSS file and a fallback when the CDN is not available. The Link Tag Helper provides the performance advantage of a CDN with the robustness of local hosting.

The following Razor markup shows the `head` element of a layout file created with the ASP.NET Core web app template:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebLinkTH</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
              href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position"
              asp-fallback-test-value="absolute"
              crossorigin="anonymous"
              integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE=" />
    </environment>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
```

The following is rendered HTML from the preceding code (in a non-Development environment):

```
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Home page - WebLinkTH</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.min.css"
          crossorigin="anonymous" integrity="sha256-eS<snip>BE=" />
    <meta name="x-stylesheet-fallback-test" content="" class="sr-only" />
    <script>
        !function (a, b, c, d) {
            var e, f = document,
                g = f.getElementsByTagName("SCRIPT"),
                h = g[g.length - 1].previousElementSibling,
                i = f.defaultView && f.defaultView.getComputedStyle ? f.defaultView.getComputedStyle(h) :
h.currentStyle;
            if (i && i[a] !== b) for (e = 0; e < c.length; e++)
                f.write('<link href="' + c[e] + '" ' + d + "/>")
        }
            ("position", "absolute", ["\/lib\/bootstrap\/dist\/css\/bootstrap.min.css"],
                "rel=\u0022stylesheet\u0022 crossorigin=\u0022anonymous\u0022 integrity=\abc<snip>BE=\u0022
");
    </script>

    <link rel="stylesheet" href="/css/site.css" />
</head>
```

In the preceding code, the Link Tag Helper generated the `<meta name="x-stylesheet-fallback-test" content="" class="sr-only" />` element and the following JavaScript which is used to verify the requested *bootstrap.min.css* file is available on the CDN. In this case, the CSS file was available so the Tag Helper generated the `<link />` element with the CDN CSS file.

Commonly used Link Tag Helper attributes

See Link Tag Helper for all the Link Tag Helper attributes, properties, and methods.

href

Preferred address of the linked resource. The address is passed thought to the generated HTML in all cases.

asp-fallback-href

The URL of a CSS stylesheet to fallback to in the case the primary URL fails.

asp-fallback-test-class

The class name defined in the stylesheet to use for the fallback test. For more information, see FallbackTestClass.

asp-fallback-test-property

The CSS property name to use for the fallback test. For more information, see FallbackTestProperty.

asp-fallback-test-value

The CSS property value to use for the fallback test. For more information, see FallbackTestValue.

Additional resources

- Tag Helpers in ASP.NET Core
- Areas in ASP.NET Core
- Introduction to Razor Pages in ASP.NET Core
- Compatibility version for ASP.NET Core MVC

Partial Tag Helper in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Scott Addie

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

View or download sample code (how to download)

Overview

The Partial Tag Helper is used for rendering a partial view in Razor Pages and MVC apps. Consider that it:

- Requires ASP.NET Core 2.1 or later.
- Is an alternative to HTML Helper syntax.
- Renders the partial view asynchronously.

The HTML Helper options for rendering a partial view include:

- `@await Html.PartialAsync`
- `@await Html.RenderPartialAsync`
- `@Html.Partial`
- `@Html.RenderPartial`

The *Product* model is used in samples throughout this document:

```
namespace TagHelpersBuiltIn.Models
{
    public class Product
    {
        public int Number { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }
    }
}
```

An inventory of the Partial Tag Helper attributes follows.

name

The `name` attribute is required. It indicates the name or the path of the partial view to be rendered. When a partial view name is provided, the view discovery process is initiated. That process is bypassed when an explicit path is provided. For all acceptable `name` values, see Partial view discovery.

The following markup uses an explicit path, indicating that *_ProductPartial.cshtml* is to be loaded from the *Shared* folder. Using the for attribute, a model is passed to the partial view for binding.

```
<partial name="Shared/_ProductPartial.cshtml" for="Product">
```

for

The `for` attribute assigns a ModelExpression to be evaluated against the current model. A `ModelExpression` infers the `@Model.` syntax. For example, `for="Product"` can be used instead of `for="@Model.Product"`. This default inference behavior is overridden by using the `@` symbol to define an inline expression.

The following markup loads *_ProductPartial.cshtml*:

```
<partial name="_ProductPartial" for="Product">
```

The partial view is bound to the associated page model's `Product` property:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using TagHelpersBuiltIn.Models;

namespace TagHelpersBuiltIn.Pages
{
    public class ProductModel : PageModel
    {
        public Product Product { get; set; }

        public void OnGet()
        {
            Product = new Product
            {
                Number = 1,
                Name = "Test product",
                Description = "This is a test product"
            };
        }
    }
}
```

model

The `model` attribute assigns a model instance to pass to the partial view. The `model` attribute can't be used with the for attribute.

In the following markup, a new `Product` object is instantiated and passed to the `model` attribute for binding:

```
<partial name="_ProductPartial"
         model='new Product { Number = 1, Name = "Test product", Description = "This is a test" }'>
```

view-data

The `view-data` attribute assigns a ViewDataDictionary to pass to the partial view. The following markup makes the entire ViewData collection accessible to the partial view:

```
@{
    ViewData["IsNumberReadOnly"] = true;
}

<partial name="_ProductViewDataPartial" for="Product" view-data="ViewData">
```

In the preceding code, the `IsNumberReadOnly` key value is set to `true` and added to the ViewData collection. Consequently, `ViewData["IsNumberReadOnly"]` is made accessible within the following partial view:

```
@model TagHelpersBuiltIn.Models.Product

<div class="form-group">
    <label asp-for="Number"></label>
    @if ((bool)ViewData["IsNumberReadOnly"])
    {
        <input asp-for="Number" type="number" class="form-control" readonly />
    }
    else
    {
        <input asp-for="Number" type="number" class="form-control" />
    }
</div>
<div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" type="text" class="form-control" />
</div>
<div class="form-group">
    <label asp-for="Description"></label>
    <textarea asp-for="Description" rows="4" cols="50" class="form-control"></textarea>
</div>
```

In this example, the value of `ViewData["IsNumberReadOnly"]` determines whether the *Number* field is displayed as read only.

Migrate from an HTML Helper

Consider the following asynchronous HTML Helper example. A collection of products is iterated and displayed. Per the `PartialAsync` method's first parameter, the *_ProductPartial.cshtml* partial view is loaded. An instance of the `Product` model is passed to the partial view for binding.

```
@foreach (var product in Model.Products)
{
    @await Html.PartialAsync("_ProductPartial", product)
}
```

The following Partial Tag Helper achieves the same asynchronous rendering behavior as the `PartialAsync` HTML Helper. The `model` attribute is assigned a `Product` model instance for binding to the partial view.

```
@foreach (var product in Model.Products)
{
    <partial name="_ProductPartial" model="product" />
}
```

Additional resources

- Partial views in ASP.NET Core
- Views in ASP.NET Core MVC

Script Tag Helper in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Rick Anderson

The Script Tag Helper generates a link to a primary or fall back script file. Typically the primary script file is on a Content Delivery Network (CDN).

A CDN:

- Provides several performance advantages vs hosting the asset with the web app.
- Should not be relied on as the only source for the asset. CDNs are not always available, therefore a reliable fallback should be used. Typically the fallback is the site hosting the web app.

The Script Tag Helper allows you to specify a CDN for the script file and a fallback when the CDN is not available. The Script Tag Helper provides the performance advantage of a CDN with the robustness of local hosting.

The following Razor markup shows a `script` element with a fallback:

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">
</script>
```

Don't use the `<script>` element's defer attribute to defer loading the CDN script. The Script Tag Helper renders JavaScript that immediately executes the asp-fallback-test expression. The expression fails if loading the CDN script is deferred.

Commonly used Script Tag Helper attributes

See Script Tag Helper for all the Script Tag Helper attributes, properties, and methods.

asp-fallback-test

The script method defined in the primary script to use for the fallback test. For more information, see FallbackTestExpression.

asp-fallback-src

The URL of a Script tag to fallback to in the case the primary one fails. For more information, see FallbackSrc.

Additional resources

- Tag Helpers in ASP.NET Core
- Areas in ASP.NET Core
- Introduction to Razor Pages in ASP.NET Core
- Compatibility version for ASP.NET Core MVC

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

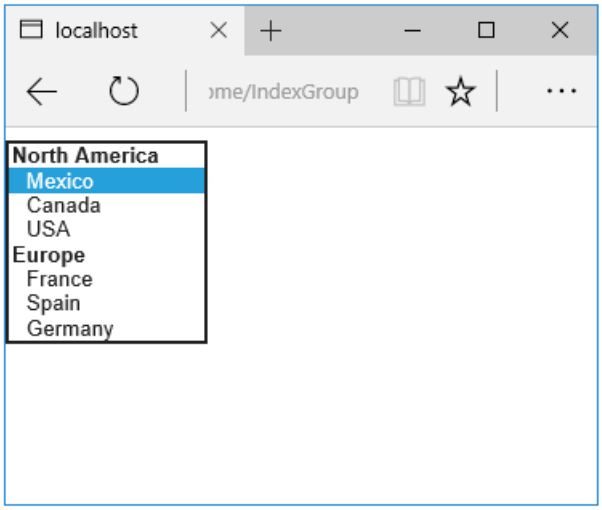
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

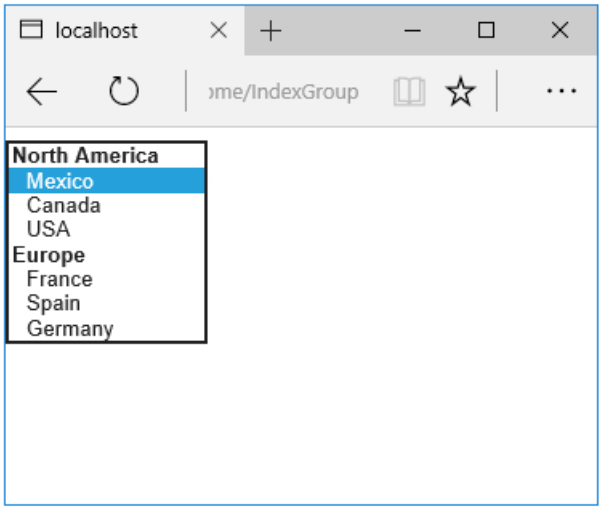
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

NOTE

With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"` .
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
  <form method="post" action="/Demo/RegisterInput">
      Email:
      <input type="email" data-val="true"
             data-val-email="The Email Address field is not a valid email address."
             data-val-required="The Email Address field is required."
             id="Email" name="Email" value=""><br>
      Password:
      <input type="password" data-val="true"
             data-val-required="The Password field is required."
             id="Password" name="Password"><br>
      <button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"`.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length of
&#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length of
&#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side

validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
<form method="post" action="/Home/IndexEnum">
       <select data-val="true" data-val-required="The EnumCountry field is required."
               id="EnumCountry" name="EnumCountry">
           <option value="0">United Mexican States</option>
           <option value="1">United States of America</option>
           <option value="2">Canada</option>
           <option value="3">France</option>
           <option value="4">Germany</option>
           <option selected="selected" value="5">Spain</option>
       </select>
       <br /><button type="submit">Register</button>
       <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
  </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

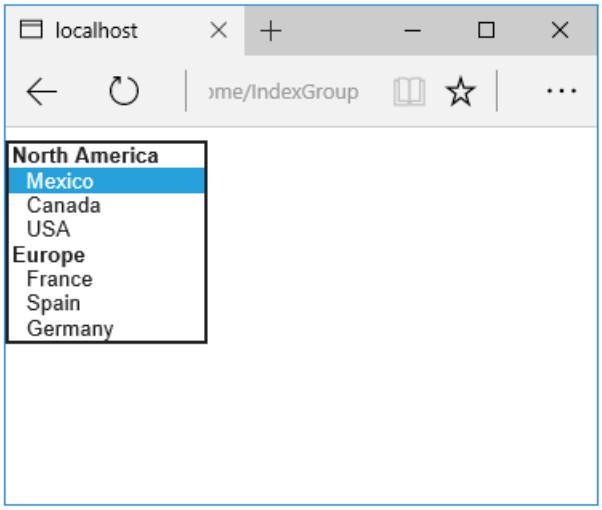
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

9/22/2020 • 18 minutes to read • Edit Online

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app

with a route named `register` could use the following markup for the registration page:

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **NOTE**
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

ATTRIBUTE	DESCRIPTION
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or

button are selected:

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the **HTML Helper alternatives to Input Tag Helper** section for details.
- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to process
this request. Please review the following specific error details and modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type 'RegisterViewModel'
could be found (are you missing a using directive or an assembly reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

.NET TYPE	INPUT TYPE
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"

.NET TYPE	INPUT TYPE
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

ATTRIBUTE	INPUT TYPE
[EmailAddress]	type="email"
[Url]	type="url"
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
           data-val-email="The Email Address field is not a valid email address."
           data-val-required="The Email Address field is required."
           id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
           data-val-required="The Password field is required."
           id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="@Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    Email:  <input asp-for="Email" /> <br />
    Password: <input asp-for="Password" /><br />
    Address: <input asp-for="Address.AddressLine1" /><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1" value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The *Views/Shared/EditorTemplates/String.cshtml* template:

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The *Views/Shared/EditorTemplates/ToDoItem.cshtml* template:

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
</td>

@*
    This template replaces the following Razor which evaluates the indexer three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> **NOTE**
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.

- HTML Helper alternative: `Html.TextAreaFor`

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type with a maximum length
of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type with a minimum length
of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.
- Less markup in source code
- Strong typing with the model property.

Sample:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on a HTML span element.

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> **NOTE**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

ASP-VALIDATION-SUMMARY	VALIDATION MESSAGES DISPLAYED
ValidationSummary.All	Property and model level
ValidationSummary.ModelOnly	Model
ValidationSummary.None	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    Email:  <input asp-for="Email" /> <br />
    <span asp-validation-for="Email"></span><br />
    Password: <input asp-for="Password" /><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

```
<form action="/DemoReg/Register" method="post">
  <div class="validation-summary-valid" data-valmsg-summary="true">
  <ul><li style="display:none"></li></ul></div>
  Email:  <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true"><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"  },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
   </form>
```

> **NOTE**
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

```
    public class CountryEnumViewModel
    {
        public CountryEnum EnumCountry { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

```
  <form method="post" action="/Home/IndexEnum">
        <select data-val="true" data-val-required="The EnumCountry field is required."
                id="EnumCountry" name="EnumCountry">
            <option value="0">United Mexican States</option>
            <option value="1">United States of America</option>
            <option value="2">Canada</option>
            <option value="3">France</option>
            <option value="4">Germany</option>
            <option selected="selected" value="5">Spain</option>
        </select>
        <br /><button type="submit">Register</button>
        <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
    </form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

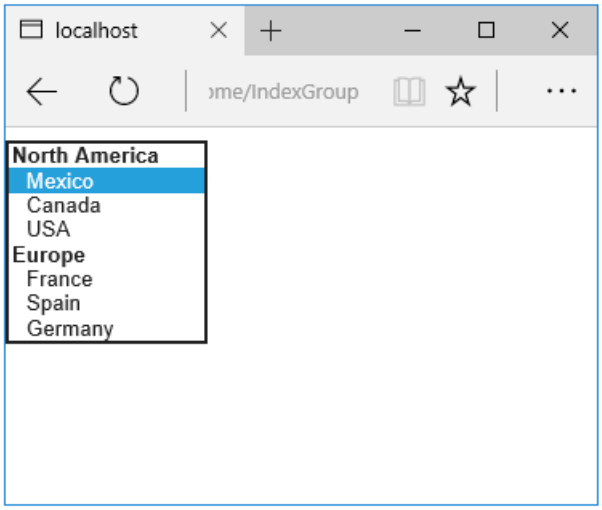
```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America" };
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }

    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

```
<form method="post" action="/Home/IndexGroup">
     <select id="Country" name="Country">
         <optgroup label="North America">
             <option value="MEX">Mexico</option>
             <option value="CAN">Canada</option>
             <option value="US">USA</option>
         </optgroup>
         <optgroup label="Europe">
             <option value="FR">France</option>
             <option value="ES">Spain</option>
             <option value="DE">Germany</option>
         </optgroup>
     </select>
     <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The *Views/Shared/EditorTemplates/CountryViewModel.cshtml* template:

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface
- Code snippets for this document

Share controllers, views, Razor Pages and more with Application Parts

9/22/2020 • 7 minutes to read • Edit Online

By Rick Anderson

View or download sample code (how to download)

An *Application Part* is an abstraction over the resources of an app. Application Parts allow ASP.NET Core to discover controllers, view components, tag helpers, Razor Pages, razor compilation sources, and more. AssemblyPart is an Application part. `AssemblyPart` encapsulates an assembly reference and exposes types and compilation references.

Feature providers work with application parts to populate the features of an ASP.NET Core app. The main use case for application parts is to configure an app to discover (or avoid loading) ASP.NET Core features from an assembly. For example, you might want to share common functionality between multiple apps. Using Application Parts, you can share an assembly (DLL) containing controllers, views, Razor Pages, razor compilation sources, Tag Helpers, and more with multiple apps. Sharing an assembly is preferred to duplicating code in multiple projects.

ASP.NET Core apps load features from ApplicationPart. The AssemblyPart class represents an application part that's backed by an assembly.

Load ASP.NET Core features

Use the Microsoft.AspNetCore.Mvc.ApplicationParts and AssemblyPart classes to discover and load ASP.NET Core features (controllers, view components, etc.). The ApplicationPartManager tracks the application parts and feature providers available. `ApplicationPartManager` is configured in `Startup.ConfigureServices`:

```
// Requires using System.Reflection;
public void ConfigureServices(IServiceCollection services)
{
    var assembly = typeof(MySharedController).Assembly;
    services.AddControllersWithViews()
        .AddApplicationPart(assembly)
        .AddRazorRuntimeCompilation();

    services.Configure<MvcRazorRuntimeCompilationOptions>(options =>
    { options.FileProviders.Add(new EmbeddedFileProvider(assembly)); });
}
```

The following code provides an alternative approach to configuring `ApplicationPartManager` using `AssemblyPart`:

```
// Requires using System.Reflection;
// Requires using Microsoft.AspNetCore.Mvc.ApplicationParts;
public void ConfigureServices(IServiceCollection services)
{
    var assembly = typeof(MySharedController).GetTypeInfo().Assembly;
    // This creates an AssemblyPart, but does not create any related parts for items such as views.
    var part = new AssemblyPart(assembly);
    services.AddControllersWithViews()
        .ConfigureApplicationPartManager(apm => apm.ApplicationParts.Add(part));
}
```

The preceding two code samples load the `SharedController` from an assembly. The `SharedController` is not in the

app's project. See the WebAppParts solution sample download.

Include views

Use a Razor class library to include views in the assembly.

Prevent loading resources

Application parts can be used to *avoid* loading resources in a particular assembly or location. Add or remove members of the Microsoft.AspNetCore.Mvc.ApplicationParts collection to hide or make available resources. The order of the entries in the `ApplicationParts` collection isn't important. Configure the `ApplicationPartManager` before using it to configure services in the container. For example, configure the `ApplicationPartManager` before invoking `AddControllersAsServices`. Call `Remove` on the `ApplicationParts` collection to remove a resource.

The `ApplicationPartManager` includes parts for:

- The app's assembly and dependent assemblies.
- `Microsoft.AspNetCore.Mvc.ApplicationParts.CompiledRazorAssemblyPart`
- `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`
- `Microsoft.AspNetCore.Mvc.TagHelpers`.
- `Microsoft.AspNetCore.Mvc.Razor`.

Feature providers

Application feature providers examine application parts and provide features for those parts. There are built-in feature providers for the following ASP.NET Core features:

- ControllerFeatureProvider
- TagHelperFeatureProvider
- MetadataReferenceFeatureProvider
- ViewsFeatureProvider
- `internal class` RazorCompiledItemFeatureProvider

Feature providers inherit from IApplicationFeatureProvider<TFeature>, where `T` is the type of the feature. Feature providers can be implemented for any of the previously listed feature types. The order of feature providers in the `ApplicationPartManager.FeatureProviders` can impact run time behavior. Later added providers can react to actions taken by earlier added providers.

Display available features

The features available to an app can be enumerated by requesting an `ApplicationPartManager` through dependency injection:

```
using AppPartsSample.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AppPartsSample.Controllers
{
    public class FeaturesController : Controller
    {
        private readonly ApplicationPartManager _partManager;

        public FeaturesController(ApplicationPartManager partManager)
        {
            _partManager = partManager;
        }

        public IActionResult Index()
        {
            var viewModel = new FeaturesViewModel();

            var controllerFeature = new ControllerFeature();
            _partManager.PopulateFeature(controllerFeature);
            viewModel.Controllers = controllerFeature.Controllers.ToList();

            var tagHelperFeature = new TagHelperFeature();
            _partManager.PopulateFeature(tagHelperFeature);
            viewModel.TagHelpers = tagHelperFeature.TagHelpers.ToList();

            var viewComponentFeature = new ViewComponentFeature();
            _partManager.PopulateFeature(viewComponentFeature);
            viewModel.ViewComponents = viewComponentFeature.ViewComponents.ToList();

            return View(viewModel);
        }
    }
}
```

The download sample uses the preceding code to display the app features:

```
Controllers:
    - FeaturesController
    - HomeController
    - HelloController
    - GenericController`1
    - GenericController`1
Tag Helpers:
    - PrerenderTagHelper
    - AnchorTagHelper
    - CacheTagHelper
    - DistributedCacheTagHelper
    - EnvironmentTagHelper
    - Additional Tag Helpers omitted for brevity.
View Components:
    - SampleViewComponent
```

Discovery in application parts

HTTP 404 errors are not uncommon when developing with application parts. These errors are typically caused by missing an essential requirement for how applications parts are discovered. If your app returns an HTTP 404 error,

verify the following requirements have been met:

- The `applicationName` setting needs to be set to the root assembly used for discovery. The root assembly used for discovery is normally the entry point assembly.
- The root assembly needs to have a reference to the parts used for discovery. The reference can be direct or transitive.
- The root assembly needs to reference the Web SDK. The framework has logic that stamps attributes into the root assembly that are used for discovery.

By Rick Anderson

View or download sample code (how to download)

An *Application Part* is an abstraction over the resources of an app. Application Parts allow ASP.NET Core to discover controllers, view components, tag helpers, Razor Pages, razor compilation sources, and more. AssemblyPart is an Application part. `AssemblyPart` encapsulates an assembly reference and exposes types and compilation references.

Feature providers work with application parts to populate the features of an ASP.NET Core app. The main use case for application parts is to configure an app to discover (or avoid loading) ASP.NET Core features from an assembly. For example, you might want to share common functionality between multiple apps. Using Application Parts, you can share an assembly (DLL) containing controllers, views, Razor Pages, razor compilation sources, Tag Helpers, and more with multiple apps. Sharing an assembly is preferred to duplicating code in multiple projects.

ASP.NET Core apps load features from ApplicationPart. The AssemblyPart class represents an application part that's backed by an assembly.

Load ASP.NET Core features

Use the `ApplicationPart` and `AssemblyPart` classes to discover and load ASP.NET Core features (controllers, view components, etc.). The ApplicationPartManager tracks the application parts and feature providers available. `ApplicationPartManager` is configured in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    // Requires using System.Reflection;
    var assembly = typeof(MySharedController).GetTypeInfo().Assembly;
    services.AddMvc()
        .AddApplicationPart(assembly)
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

The following code provides an alternative approach to configuring `ApplicationPartManager` using `AssemblyPart`:

```
public void ConfigureServices(IServiceCollection services)
{
    // Requires using System.Reflection;
    // Requires using Microsoft.AspNetCore.Mvc.ApplicationParts;
    var assembly = typeof(MySharedController).GetTypeInfo().Assembly;
    var part = new AssemblyPart(assembly);
    services.AddMvc()
        .ConfigureApplicationPartManager(apm => apm.ApplicationParts.Add(part))
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

The preceding two code samples load the `SharedController` from an assembly. The `SharedController` is not in the application's project. See the WebAppParts solution sample download.

Include views

Use a Razor class library to include views in the assembly.

Prevent loading resources

Application parts can be used to *avoid* loading resources in a particular assembly or location. Add or remove members of the Microsoft.AspNetCore.Mvc.ApplicationParts collection to hide or make available resources. The order of the entries in the `ApplicationParts` collection isn't important. Configure the `ApplicationPartManager` before using it to configure services in the container. For example, configure the `ApplicationPartManager` before invoking `AddControllersAsServices`. Call `Remove` on the `ApplicationParts` collection to remove a resource.

The following code uses Microsoft.AspNetCore.Mvc.ApplicationParts to remove `MyDependentLibrary` from the app:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
            .ConfigureApplicationPartManager(apm =>
            {
                var dependentLibrary = apm.ApplicationParts
                    .FirstOrDefault(part => part.Name == "MyDependentLibrary");

                if (dependentLibrary != null)
                {
                    apm.ApplicationParts.Remove(dependentLibrary);
                }
            });
}
```

The `ApplicationPartManager` includes parts for:

- The app's assembly and dependent assemblies.
- `Microsoft.AspNetCore.Mvc.TagHelpers`.
- `Microsoft.AspNetCore.Mvc.Razor`.

Application feature providers

Application feature providers examine application parts and provide features for those parts. There are built-in feature providers for the following ASP.NET Core features:

- Controllers
- Tag Helpers
- View Components

Feature providers inherit from IApplicationFeatureProvider<TFeature>, where `T` is the type of the feature. Feature providers can be implemented for any of the previously listed feature types. The order of feature providers in the `ApplicationPartManager.FeatureProviders` can impact run time behavior. Later added providers can react to actions taken by earlier added providers.

Display available features

The features available to an app can be enumerated by requesting an `ApplicationPartManager` through dependency injection:

```
using AppPartsSample.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AppPartsSample.Controllers
{
    public class FeaturesController : Controller
    {
        private readonly ApplicationPartManager _partManager;

        public FeaturesController(ApplicationPartManager partManager)
        {
            _partManager = partManager;
        }

        public IActionResult Index()
        {
            var viewModel = new FeaturesViewModel();

            var controllerFeature = new ControllerFeature();
            _partManager.PopulateFeature(controllerFeature);
            viewModel.Controllers = controllerFeature.Controllers.ToList();

            var tagHelperFeature = new TagHelperFeature();
            _partManager.PopulateFeature(tagHelperFeature);
            viewModel.TagHelpers = tagHelperFeature.TagHelpers.ToList();

            var viewComponentFeature = new ViewComponentFeature();
            _partManager.PopulateFeature(viewComponentFeature);
            viewModel.ViewComponents = viewComponentFeature.ViewComponents.ToList();

            return View(viewModel);
        }
    }
}
```

The download sample uses the preceding code to display the app features:

```
Controllers:
    - FeaturesController
    - HomeController
    - HelloController
    - GenericController`1
    - GenericController`1
Tag Helpers:
    - PrerenderTagHelper
    - AnchorTagHelper
    - CacheTagHelper
    - DistributedCacheTagHelper
    - EnvironmentTagHelper
    - Additional Tag Helpers omitted for brevity.
View Components:
    - SampleViewComponent
```

Discovery in application parts

HTTP 404 errors are not uncommon when developing with application parts. These errors are typically caused by missing an essential requirement for how applications parts are discovered. If your app returns an HTTP 404 error,

verify the following requirements have been met:

- The `applicationName` setting needs to be set to the root assembly used for discovery. The root assembly used for discovery is normally the entry point assembly.
- The root assembly needs to have a reference to the parts used for discovery. The reference can be direct or transitive.
- The root assembly needs to reference the Web SDK.
  - The ASP.NET Core framework has custom build logic that stamps attributes into the root assembly that are used for discovery.

Work with the application model in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Steve Smith

ASP.NET Core MVC defines an *application model* representing the components of an MVC app. You can read and manipulate this model to modify how MVC elements behave. By default, MVC follows certain conventions to determine which classes are considered to be controllers, which methods on those classes are actions, and how parameters and routing behave. You can customize this behavior to suit your app's needs by creating your own conventions and applying them globally or as attributes.

Models and Providers

The ASP.NET Core MVC application model include both abstract interfaces and concrete implementation classes that describe an MVC application. This model is the result of MVC discovering the app's controllers, actions, action parameters, routes, and filters according to default conventions. By working with the application model, you can modify your app to follow different conventions from the default MVC behavior. The parameters, names, routes, and filters are all used as configuration data for actions and controllers.

The ASP.NET Core MVC Application Model has the following structure:

- ApplicationModel
  - Controllers (ControllerModel)
    - Actions (ActionModel)
      - Parameters (ParameterModel)

Each level of the model has access to a common `Properties` collection, and lower levels can access and overwrite property values set by higher levels in the hierarchy. The properties are persisted to the `ActionDescriptor.Properties` when the actions are created. Then when a request is being handled, any properties a convention added or modified can be accessed through `ActionContext.ActionDescriptor.Properties`. Using properties is a great way to configure your filters, model binders, etc. on a per-action basis.

> **NOTE**
> The `ActionDescriptor.Properties` collection isn't thread safe (for writes) once app startup has finished. Conventions are the best way to safely add data to this collection.

IApplicationModelProvider

ASP.NET Core MVC loads the application model using a provider pattern, defined by the IApplicationModelProvider interface. This section covers some of the internal implementation details of how this provider functions. This is an advanced topic - most apps that leverage the application model should do so by working with conventions.

Implementations of the `IApplicationModelProvider` interface "wrap" one another, with each implementation calling `OnProvidersExecuting` in ascending order based on its `Order` property. The `OnProvidersExecuted` method is then called in reverse order. The framework defines several providers:

First (`Order=-1000`):

- `DefaultApplicationModelProvider`

Then (`Order=-990`):

- `AuthorizationApplicationModelProvider`
- `CorsApplicationModelProvider`

> **NOTE**
> The order in which two providers with the same value for `Order` are called is undefined, and therefore shouldn't be relied upon.

> **NOTE**
> `IApplicationModelProvider` is an advanced concept for framework authors to extend. In general, apps should use conventions and frameworks should use providers. The key distinction is that providers always run before conventions.

The `DefaultApplicationModelProvider` establishes many of the default behaviors used by ASP.NET Core MVC. Its responsibilities include:

- Adding global filters to the context
- Adding controllers to the context
- Adding public controller methods as actions
- Adding action method parameters to the context
- Applying route and other attributes

Some built-in behaviors are implemented by the `DefaultApplicationModelProvider`. This provider is responsible for constructing the `ControllerModel`, which in turn references `ActionModel`, `PropertyModel`, and `ParameterModel` instances. The `DefaultApplicationModelProvider` class is an internal framework implementation detail that can and will change in the future.

The `AuthorizationApplicationModelProvider` is responsible for applying the behavior associated with the `AuthorizeFilter` and `AllowAnonymousFilter` attributes. Learn more about these attributes.

The `CorsApplicationModelProvider` implements behavior associated with the `IEnableCorsAttribute` and `IDisableCorsAttribute`, and the `DisableCorsAuthorizationFilter`. Learn more about CORS.

Conventions

The application model defines convention abstractions that provide a simpler way to customize the behavior of the models than overriding the entire model or provider. These abstractions are the recommended way to modify your app's behavior. Conventions provide a way for you to write code that will dynamically apply customizations. While filters provide a means of modifying the framework's behavior, customizations let you control how the whole app works together.

The following conventions are available:

- `IApplicationModelConvention`
- `IControllerModelConvention`
- `IActionModelConvention`
- `IParameterModelConvention`

Conventions are applied by adding them to MVC options or by implementing `Attribute`s and applying them to controllers, actions, or action parameters (similar to `Filters`). Unlike filters, conventions are only executed when the app is starting, not as part of each request.

Sample: Modifying the ApplicationModel

The following convention is used to add a property to the application model.

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ApplicationDescription : IApplicationModelConvention
    {
        private readonly string _description;

        public ApplicationDescription(string description)
        {
            _description = description;
        }

        public void Apply(ApplicationModel application)
        {
            application.Properties["description"] = _description;
        }
    }
}
```

Application model conventions are applied as options when MVC is added in `ConfigureServices` in `Startup`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}
```

Properties are accessible from the `ActionDescriptor` properties collection within controller actions:

```
public class AppModelController : Controller
{
    public string Description()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

Sample: Modifying the ControllerModel Description

As in the previous example, the controller model can also be modified to include custom properties. These will override existing properties with the same name specified in the application model. The following convention attribute adds a description at the controller level:

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ControllerDescriptionAttribute : Attribute, IControllerModelConvention
    {
        private readonly string _description;

        public ControllerDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ControllerModel controllerModel)
        {
            controllerModel.Properties["description"] = _description;
        }
    }
}
```

This convention is applied as an attribute on a controller.

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
```

The "description" property is accessed in the same manner as in previous examples.

Sample: Modifying the ActionModel Description

A separate attribute convention can be applied to individual actions, overriding behavior already applied at the application or controller level.

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ActionDescriptionAttribute : Attribute, IActionModelConvention
    {
        private readonly string _description;

        public ActionDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ActionModel actionModel)
        {
            actionModel.Properties["description"] = _description;
        }
    }
}
```

Applying this to an action within the previous example's controller demonstrates how it overrides the controller-level convention:

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }

    [ActionDescription("Action Description")]
    public string UseActionDescriptionAttribute()
    {
        return "Description: " + ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

Sample: Modifying the ParameterModel

The following convention can be applied to action parameters to modify their `BindingInfo`. The following convention requires that the parameter be a route parameter; other potential binding sources (such as query string values) are ignored.

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace AppModelSample.Conventions
{
    public class MustBeInRouteParameterModelConvention : Attribute, IParameterModelConvention
    {
        public void Apply(ParameterModel model)
        {
            if (model.BindingInfo == null)
            {
                model.BindingInfo = new BindingInfo();
            }
            model.BindingInfo.BindingSource = BindingSource.Path;
        }
    }
}
```

The attribute may be applied to any action parameter:

```
public class ParameterModelController : Controller
{
    // Will bind:  /ParameterModel/GetById/123
    // WON'T bind: /ParameterModel/GetById?id=123
    public string GetById([MustBeInRouteParameterModelConvention]int id)
    {
        return $"Bound to id: {id}";
    }
}
```

Sample: Modifying the ActionModel Name

The following convention modifies the `ActionModel` to update the *name* of the action to which it's applied. The new name is provided as a parameter to the attribute. This new name is used by routing, so it will affect the route used to reach this action method.

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class CustomActionNameAttribute : Attribute, IActionModelConvention
    {
        private readonly string _actionName;

        public CustomActionNameAttribute(string actionName)
        {
            _actionName = actionName;
        }

        public void Apply(ActionModel actionModel)
        {
            // this name will be used by routing
            actionModel.ActionName = _actionName;
        }
    }
}
```

This attribute is applied to an action method in the `HomeController`:

```
// Route: /Home/MyCoolAction
[CustomActionName("MyCoolAction")]
public string SomeName()
{
    return ControllerContext.ActionDescriptor.ActionName;
}
```

Even though the method name is `SomeName`, the attribute overrides the MVC convention of using the method name and replaces the action name with `MyCoolAction`. Thus, the route used to reach this action is `/Home/MyCoolAction`.

> **NOTE**
>
> This example is essentially the same as using the built-in ActionName attribute.

Sample: Custom Routing Convention

You can use an `IApplicationModelConvention` to customize how routing works. For example, the following convention will incorporate Controllers' namespaces into their routes, replacing `.` in the namespace with `/` in the route:

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace AppModelSample.Conventions
{
    public class NamespaceRoutingConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            foreach (var controller in application.Controllers)
            {
                var hasAttributeRouteModels = controller.Selectors
                    .Any(selector => selector.AttributeRouteModel != null);

                if (!hasAttributeRouteModels
                    && controller.ControllerName.Contains("Namespace")) // affect one controller in this sample
                {
                    // Replace the . in the namespace with a / to create the attribute route
                    // Ex: MySite.Admin namespace will correspond to MySite/Admin attribute route
                    // Then attach [controller], [action] and optional {id?} token.
                    // [Controller] and [action] is replaced with the controller and action
                    // name to generate the final template
                    controller.Selectors[0].AttributeRouteModel = new AttributeRouteModel()
                    {
                        Template = controller.ControllerType.Namespace.Replace('.', '/') +
"/[controller]/[action]/{id?}"
                    };
                }
            }

            // You can continue to put attribute route templates for the controller actions depending on the
way you want them to behave
        }
    }
}
```

The convention is added as an option in Startup.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
        //options.Conventions.Add(new IdsMustBeInRouteParameterModelConvention());
    });
}
```

TIP

You can add conventions to your middleware by accessing `MvcOptions` using `services.Configure<MvcOptions>(c => c.Conventions.Add(YOURCONVENTION));`

This sample applies this convention to routes that are not using attribute routing where the controller has "Namespace" in its name. The following controller demonstrates this convention:

```
using Microsoft.AspNetCore.Mvc;

namespace AppModelSample.Controllers
{
    public class NamespaceRoutingController : Controller
    {
        // using NamespaceRoutingConvention
        // route: /AppModelSample/Controllers/NamespaceRouting/Index
        public string Index()
        {
            return "This demonstrates namespace routing.";
        }
    }
}
```

Application Model Usage in WebApiCompatShim

ASP.NET Core MVC uses a different set of conventions from ASP.NET Web API 2. Using custom conventions, you can modify an ASP.NET Core MVC app's behavior to be consistent with that of a Web API app. Microsoft ships the WebApiCompatShim specifically for this purpose.

> **NOTE**
> Learn more about migration from ASP.NET Web API.

To use the Web API Compatibility Shim, you need to add the package to your project and then add the conventions to MVC by calling `AddWebApiConventions` in `Startup`:

```
services.AddMvc().AddWebApiConventions();
```

The conventions provided by the shim are only applied to parts of the app that have had certain attributes applied to them. The following four attributes are used to control which controllers should have their conventions modified by the shim's conventions:

- UseWebApiActionConventionsAttribute
- UseWebApiOverloadingAttribute
- UseWebApiParameterConventionsAttribute
- UseWebApiRoutesAttribute

Action Conventions

The `UseWebApiActionConventionsAttribute` is used to map the HTTP method to actions based on their name (for instance, `Get` would map to `HttpGet`). It only applies to actions that don't use attribute routing.

Overloading

The `UseWebApiOverloadingAttribute` is used to apply the `WebApiOverloadingApplicationModelConvention` convention. This convention adds an `OverloadActionConstraint` to the action selection process, which limits candidate actions to those for which the request satisfies all non-optional parameters.

Parameter Conventions

The `UseWebApiParameterConventionsAttribute` is used to apply the `WebApiParameterConventionsApplicationModelConvention` action convention. This convention specifies that simple types used as action parameters are bound from the URI by default, while complex types are bound from the request body.

Routes

The `UseWebApiRoutesAttribute` controls whether the `WebApiApplicationModelConvention` controller convention is applied. When enabled, this convention is used to add support for areas to the route.

In addition to a set of conventions, the compatibility package includes a `System.Web.Http.ApiController` base class that replaces the one provided by Web API. This allows your controllers written for Web API and inheriting from its `ApiController` to work as they were designed, while running on ASP.NET Core MVC. All of the `UseWebApi*` attributes listed earlier are applied to the base controller class. The `ApiController` exposes properties, methods, and result types that are compatible with those found in Web API.

Using ApiExplorer to Document Your App

The application model exposes an `ApiExplorer` property at each level that can be used to traverse the app's structure. This can be used to generate help pages for your Web APIs using tools like Swagger. The `ApiExplorer` property exposes an `IsVisible` property that can be set to specify which parts of your app's model should be exposed. You can configure this setting using a convention:

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class EnableApiExplorerApplicationConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            application.ApiExplorer.IsVisible = true;
        }
    }
}
```

Using this approach (and additional conventions if required), you can enable or disable API visibility at any level within your app.

Areas in ASP.NET Core

9/22/2020 • 14 minutes to read • Edit Online

By Dhananjay Kumar and Rick Anderson

Areas are an ASP.NET feature used to organize related functionality into a group as a separate:

- Namespace for routing.
- Folder structure for views and Razor Pages.

Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area`, to `controller` and `action` or a Razor Page `page`.

Areas provide a way to partition an ASP.NET Core Web app into smaller functional groups, each with its own set of Razor Pages, controllers, views, and models. An area is effectively a structure inside an app. In an ASP.NET Core web project, logical components like Pages, Model, Controller, and View are kept in different folders. The ASP.NET Core runtime uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search. Each of these units have their own area to contain views, controllers, Razor Pages, and models.

Consider using Areas in a project when:

- The app is made of multiple high-level functional components that can be logically separated.
- You want to partition the app so that each functional area can be worked on independently.

View or download sample code (how to download). The download sample provides a basic app for testing areas.

If you're using Razor Pages, see Areas with Razor Pages in this document.

Areas for controllers with views

A typical ASP.NET Core web app using areas, controllers, and views contains the following:

- An Area folder structure.
- Controllers with the `[Area]` attribute to associate the controller with the area:

```
[Area("Products")]
public class ManageController : Controller
{
```

- The area route added to startup:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "MyArea",
        pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Area folder structure

Consider an app that has two logical groups, *Products* and *Services*. Using areas, the folder structure would be similar to the following:

- Project name
  - Areas
    - Products
      - Controllers
        - HomeController.cs
        - ManageController.cs
      - Views
        - Home
          - Index.cshtml
        - Manage
          - Index.cshtml
          - About.cshtml
    - Services
      - Controllers
        - HomeController.cs
      - Views
        - Home
          - Index.cshtml

While the preceding layout is typical when using Areas, only the view files are required to use this folder structure. View discovery searches for a matching area view file in the following order:

```
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml
/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml
/Views/Shared/<Action-Name>.cshtml
/Pages/Shared/<Action-Name>.cshtml
```

Associate the controller with an Area

Area controllers are designated with the [Area] attribute:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Docs.Samples;

namespace MVCareas.Areas.Products.Controllers
{
    [Area("Products")]
    public class ManageController : Controller
    {
        public IActionResult Index()
        {
            ViewData["routeInfo"] = ControllerContext.MyDisplayRouteInfo();
            return View();
        }

        public IActionResult About()
        {
            ViewData["routeInfo"] = ControllerContext.MyDisplayRouteInfo();
            return View();
        }
    }
}
```

Add Area route

Area routes typically use conventional routing rather than attribute routing. Conventional routing is order-dependent. In general, routes with areas should be placed earlier in the route table as they're more specific than routes without an area.

`{area:...}` can be used as a token in route templates if url space is uniform across all areas:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "MyArea",
            pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

In the preceding code, `exists` applies a constraint that the route must match an area. Using `{area:...}` with `MapControllerRoute`:

- Is the least complicated mechanism to adding routing to areas.

- Matches all controllers with the `[Area("Area name")]` attribute.

The following code uses MapAreaControllerRoute to create two named area routes:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapAreaControllerRoute(
            name: "MyAreaProducts",
            areaName: "Products",
            pattern: "Products/{controller=Home}/{action=Index}/{id?}");

        endpoints.MapAreaControllerRoute(
            name: "MyAreaServices",
            areaName: "Services",
            pattern: "Services/{controller=Home}/{action=Index}/{id?}");

        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

For more information, see Area routing.

Link generation with MVC areas

The following code from the sample download shows link generation with the area specified:

```
<li>Anchor Tag Helper links</li>
<ul>
    <li>
        <a asp-area="Products" asp-controller="Home" asp-action="About">
            Products/Home/About
        </a>
    </li>
    <li>
        <a asp-area="Services" asp-controller="Home" asp-action="About">
            Services About
        </a>
    </li>
    <li>
        <a asp-area="" asp-controller="Home" asp-action="About">
            /Home/About
        </a>
    </li>
</ul>
<li>Html.ActionLink generated links</li>
<ul>
    <li>
        @Html.ActionLink("Product/Manage/About", "About", "Manage",
                                                 new { area = "Products" })
    </li>
</ul>
<li>Url.Action generated links</li>
<ul>
    <li>
        <a href='@Url.Action("About", "Manage", new { area = "Products" })'>
            Products/Manage/About
        </a>
    </li>
</ul>
```

The sample download includes a partial view that contains:

- The preceding links.
- Links similar to the preceding except `area` is not specified.

The partial view is referenced in the layout file, so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area and controller.

When the area or controller is not specified, routing depends on the ambient values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links with the markup that doesn't specify the area.

For more information, see Routing to controller actions.

Shared layout for Areas using the _ViewStart.cshtml file

To share a common layout for the entire app, keep the *_ViewStart.cshtml* in the application root folder. For more information, see Layout in ASP.NET Core

Application root folder

The application root folder is the folder containing *Startup.cs* in web app created with the ASP.NET Core templates.

_ViewImports.cshtml

/Views/_ViewImports.cshtml, for MVC, and */Pages/_ViewImports.cshtml* for Razor Pages, is not imported to views in areas. Use one of the following approaches to provide view imports to all views:

- Add *_ViewImports.cshtml* to the application root folder. A *_ViewImports.cshtml* in the application root folder will apply to all views in the app.
- Copy the *_ViewImports.cshtml* file to the appropriate view folder under areas.

The *_ViewImports.cshtml* file typically contains Tag Helpers imports, `@using`, and `@inject` statements. For more information, see Importing Shared Directives.

Change default area folder where views are stored

The following code changes the default area folder from `"Areas"` to `"MyAreas"`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<RazorViewEngineOptions>(options =>
    {
        options.AreaViewLocationFormats.Clear();
        options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/{1}/{0}.cshtml");
        options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/Shared/{0}.cshtml");
        options.AreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
    });

    services.AddControllersWithViews();
}
```

Areas with Razor Pages

Areas with Razor Pages require an `Areas/<area name>/Pages` folder in the root of the app. The following folder structure is used with the sample app:

- Project name
  - Areas
    - Products
      - Pages
        - _ViewImports
        - About
        - Index
    - Services
      - Pages
        - Manage
          - About
          - Index

Link generation with Razor Pages and areas

The following code from the sample download shows link generation with the area specified (for example, `asp-area="Products"`):

```
<li>Anchor Tag Helper links</li>
<ul>
    <li>
        <a asp-area="Products" asp-page="/About">
            Products/About
        </a>
    </li>
    <li>
        <a asp-area="Services" asp-page="/Manage/About">
            Services/Manage/About
        </a>
    </li>
    <li>
        <a asp-area="" asp-page="/About">
            /About
        </a>
    </li>
</ul>
<li>Url.Page generated links</li>
<ul>
    <li>
        <a href='@Url.Page("/Manage/About", new { area = "Services" })'>
            Services/Manage/About
        </a>
    </li>
    <li>
        <a href='@Url.Page("/About", new { area = "Products" })'>
            Products/About
        </a>
    </li>
</ul>
```

The sample download includes a partial view that contains the preceding links and the same links without specifying the area. The partial view is referenced in the layout file, so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area.

When the area is not specified, routing depends on the *ambient* values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links. For example, consider the links generated from the following code:

```
<li>
    <a asp-page="/Manage/About">
        Services/Manage/About
    </a>
</li>
<li>
    <a asp-page="/About">
        /About
    </a>
</li>
```

For the preceding code:

- The link generated from `<a asp-page="/Manage/About">` is correct only when the last request was for a page in `Services` area. For example, `/Services/Manage/`, `/Services/Manage/Index`, or `/Services/Manage/About`.
- The link generated from `<a asp-page="/About">` is correct only when the last request was for a page in `/Home`.

- The code is from the sample download.

Import namespace and Tag Helpers with _ViewImports file

A *_ViewImports.cshtml* file can be added to each area *Pages* folder to import the namespace and Tag Helpers to each Razor Page in the folder.

Consider the *Services* area of the sample code, which doesn't contain a *_ViewImports.cshtml* file. The following markup shows the */Services/Manage/About* Razor Page:

```
@page
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model RPareas.Areas.Services.Pages.Manage.AboutModel
@{
    ViewData["Title"] = "Srv Mng About";
}

<a asp-area="Products" asp-page="/Index">
    Products/Index
</a>
```

In the preceding markup:

- The fully qualified domain name must be used to specify the model (`@model RPareas.Areas.Services.Pages.Manage.AboutModel`).
- Tag Helpers are enabled by `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`

In the sample download, the Products area contains the following *_ViewImports.cshtml* file:

```
@namespace RPareas.Areas.Products.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The following markup shows the */Products/About* Razor Page:

```
@page
@model AboutModel
@{
    ViewData["Title"] = "Prod About";
}
```

In the preceding file, the namespace and `@addTagHelper` directive is imported to the file by the *Areas/Products/Pages/_ViewImports.cshtml* file.

For more information, see Managing Tag Helper scope and Importing Shared Directives.

Shared layout for Razor Pages Areas

To share a common layout for the entire app, move the *_ViewStart.cshtml* to the application root folder.

Publishing Areas

All *.cshtml files and files within the *wwwroot* directory are published to output when `<Project Sdk="Microsoft.NET.Sdk.Web">` is included in the *.csproj file.

Areas are an ASP.NET feature used to organize related functionality into a group as a separate namespace (for routing) and folder structure (for views). Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area`, to `controller` and `action` or a Razor Page `page`.

Areas provide a way to partition an ASP.NET Core Web app into smaller functional groups, each with its own set of Razor Pages, controllers, views, and models. An area is effectively a structure inside an app. In an

ASP.NET Core web project, logical components like Pages, Model, Controller, and View are kept in different folders. The ASP.NET Core runtime uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search. Each of these units have their own area to contain views, controllers, Razor Pages, and models.

Consider using Areas in a project when:

- The app is made of multiple high-level functional components that can be logically separated.
- You want to partition the app so that each functional area can be worked on independently.

View or download sample code (how to download). The download sample provides a basic app for testing areas.

If you're using Razor Pages, see Areas with Razor Pages in this document.

Areas for controllers with views

A typical ASP.NET Core web app using areas, controllers, and views contains the following:

- An Area folder structure.
- Controllers with the `[Area]` attribute to associate the controller with the area:

```
[Area("Products")]
public class ManageController : Controller
{
```

- The area route added to startup:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
      name: "MyArea",
      template: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

    routes.MapRoute(
       name: "default",
       template: "{controller=Home}/{action=Index}/{id?}");
});
```

Area folder structure

Consider an app that has two logical groups, *Products* and *Services*. Using areas, the folder structure would be similar to the following:

- Project name
  - Areas
    - Products
      - Controllers
        - HomeController.cs
        - ManageController.cs
      - Views
        - Home
          - Index.cshtml
        - Manage

        - Index.cshtml
        - About.cshtml
    - Services
        - Controllers
            - HomeController.cs
        - Views
            - Home
                - Index.cshtml

While the preceding layout is typical when using Areas, only the view files are required to use this folder structure. View discovery searches for a matching area view file in the following order:

```
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml
/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml
/Views/Shared/<Action-Name>.cshtml
/Pages/Shared/<Action-Name>.cshtml
```

Associate the controller with an Area

Area controllers are designated with the [Area] attribute:

```
using Microsoft.AspNetCore.Mvc;

namespace MVCareas.Areas.Products.Controllers
{
    [Area("Products")]
    public class ManageController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult About()
        {
            return View();
        }
    }
}
```

Add Area route

Area routes typically use conventional routing rather than attribute routing. Conventional routing is order-dependent. In general, routes with areas should be placed earlier in the route table as they're more specific than routes without an area.

`{area:...}` can be used as a token in route templates if url space is uniform across all areas:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
          name: "MyArea",
          template: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

        routes.MapRoute(
           name: "default",
           template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

In the preceding code, `exists` applies a constraint that the route must match an area. Using `{area:...}` is the least complicated mechanism to adding routing to areas.

The following code uses MapAreaRoute to create two named area routes:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapAreaRoute(
            name: "MyAreaProducts",
            areaName:"Products",
            template: "Products/{controller=Home}/{action=Index}/{id?}");

        routes.MapAreaRoute(
            name: "MyAreaServices",
            areaName: "Services",
            template: "Services/{controller=Home}/{action=Index}/{id?}");

        routes.MapRoute(
           name: "default",
           template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

When using `MapAreaRoute` with ASP.NET Core 2.2, see this GitHub issue.

For more information, see Area routing.

Link generation with MVC areas

The following code from the sample download shows link generation with the area specified:

```
<li>Anchor Tag Helper links</li>
<ul>
    <li>
        <a asp-area="Products" asp-controller="Home" asp-action="About">
            Products/Home/About
        </a>
    </li>
    <li>
        <a asp-area="Services" asp-controller="Home" asp-action="About">
            Services About
        </a>
    </li>
    <li>
        <a asp-area="" asp-controller="Home" asp-action="About">
            /Home/About
        </a>
    </li>
</ul>
<li>Html.ActionLink generated links</li>
<ul>
    <li>
        @Html.ActionLink("Product/Manage/About", "About", "Manage",
                                                 new { area = "Products" })
    </li>
</ul>
<li>Url.Action generated links</li>
<ul>
    <li>
        <a href='@Url.Action("About", "Manage", new { area = "Products" })'>
            Products/Manage/About
        </a>
    </li>
</ul>
```

The links generated with the preceding code are valid anywhere in the app.

The sample download includes a partial view that contains the preceding links and the same links without specifying the area. The partial view is referenced in the layout file, so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area and controller.

When the area or controller is not specified, routing depends on the *ambient* values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links.

For more information, see Routing to controller actions.

Shared layout for Areas using the _ViewStart.cshtml file

To share a common layout for the entire app, move the *_ViewStart.cshtml* to the application root folder.

_ViewImports.cshtml

In its standard location, */Views/_ViewImports.cshtml* doesn't apply to areas. To use common Tag Helpers, `@using`, or `@inject` in your area, ensure a proper *_ViewImports.cshtml* file applies to your area views. If you want the same behavior in all your views, move */Views/_ViewImports.cshtml* to the application root.

Change default area folder where views are stored

The following code changes the default area folder from `"Areas"` to `"MyAreas"`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<RazorViewEngineOptions>(options =>
    {
        options.AreaViewLocationFormats.Clear();
        options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/{1}/{0}.cshtml");
        options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/Shared/{0}.cshtml");
        options.AreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
    });

    services.AddMvc();
}
```

Areas with Razor Pages

Areas with Razor Pages require an `Areas/<area name>/Pages` folder in the root of the app. The following folder structure is used with the sample app:

- Project name
  - Areas
    - Products
      - Pages
        - _ViewImports
        - About
        - Index
    - Services
      - Pages
        - Manage
          - About
          - Index

Link generation with Razor Pages and areas

The following code from the sample download shows link generation with the area specified (for example, `asp-area="Products"`):

```
<li>Anchor Tag Helper links</li>
<ul>
    <li>
        <a asp-area="Products" asp-page="/About">
            Products/About
        </a>
    </li>
    <li>
        <a asp-area="Services" asp-page="/Manage/About">
            Services/Manage/About
        </a>
    </li>
    <li>
        <a asp-area="" asp-page="/About">
            /About
        </a>
    </li>
</ul>
<li>Url.Page generated links</li>
<ul>
    <li>
        <a href='@Url.Page("/Manage/About", new { area = "Services" })'>
            Services/Manage/About
        </a>
    </li>
    <li>
        <a href='@Url.Page("/About", new { area = "Products" })'>
            Products/About
        </a>
    </li>
</ul>
```

The links generated with the preceding code are valid anywhere in the app.

The sample download includes a partial view that contains the preceding links and the same links without specifying the area. The partial view is referenced in the layout file, so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area.

When the area is not specified, routing depends on the *ambient* values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links. For example, consider the links generated from the following code:

```
<li>
    <a asp-page="/Manage/About">
        Services/Manage/About
    </a>
</li>
<li>
    <a asp-page="/About">
        /About
    </a>
</li>
```

For the preceding code:

- The link generated from `<a asp-page="/Manage/About">` is correct only when the last request was for a page in `Services` area. For example, `/Services/Manage/`, `/Services/Manage/Index`, or `/Services/Manage/About`.
- The link generated from `<a asp-page="/About">` is correct only when the last request was for a page in

`/Home`.

- The code is from the sample download.

Import namespace and Tag Helpers with _ViewImports file

A *_ViewImports.cshtml* file can be added to each area *Pages* folder to import the namespace and Tag Helpers to each Razor Page in the folder.

Consider the *Services* area of the sample code, which doesn't contain a *_ViewImports.cshtml* file. The following markup shows the */Services/Manage/About* Razor Page:

```
@page
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model RPareas.Areas.Services.Pages.Manage.AboutModel
@{
    ViewData["Title"] = "Srv Mng About";
}

<h2>/Services/Manage/About</h2>

<a asp-area="Products" asp-page="/Index">
    Products/Index
</a>
```

In the preceding markup:

- The fully qualified domain name must be used to specify the model (`@model RPareas.Areas.Services.Pages.Manage.AboutModel`).
- Tag Helpers are enabled by `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`

In the sample download, the Products area contains the following *_ViewImports.cshtml* file:

```
@namespace RPareas.Areas.Products.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The following markup shows the */Products/About* Razor Page:

```
@page
@model AboutModel
@{
    ViewData["Title"] = "Prod About";
}

<h2>Products/About</h2>

<a asp-area="Services" asp-page="/Manage/About">
    Services/Manage/About
</a>
```

In the preceding file, the namespace and `@addTagHelper` directive is imported to the file by the *Areas/Products/Pages/_ViewImports.cshtml* file.

For more information, see Managing Tag Helper scope and Importing Shared Directives.

Shared layout for Razor Pages Areas

To share a common layout for the entire app, move the *_ViewStart.cshtml* to the application root folder.

Publishing Areas

All *.cshtml files and files within the *wwwroot* directory are published to output when

`<Project Sdk="Microsoft.NET.Sdk.Web">` is included in the *.csproj file.

Filters in ASP.NET Core

9/22/2020 • 44 minutes to read • Edit Online

By Kirk Larkin, Rick Anderson, Tom Dykstra, and Steve Smith

Filters in ASP.NET Core allow code to be run before or after specific stages in the request processing pipeline.

Built-in filters handle tasks such as:

- Authorization (preventing access to resources a user isn't authorized for).
- Response caching (short-circuiting the request pipeline to return a cached response).

Custom filters can be created to handle cross-cutting concerns. Examples of cross-cutting concerns include error handling, caching, configuration, authorization, and logging. Filters avoid duplicating code. For example, an error handling exception filter could consolidate error handling.

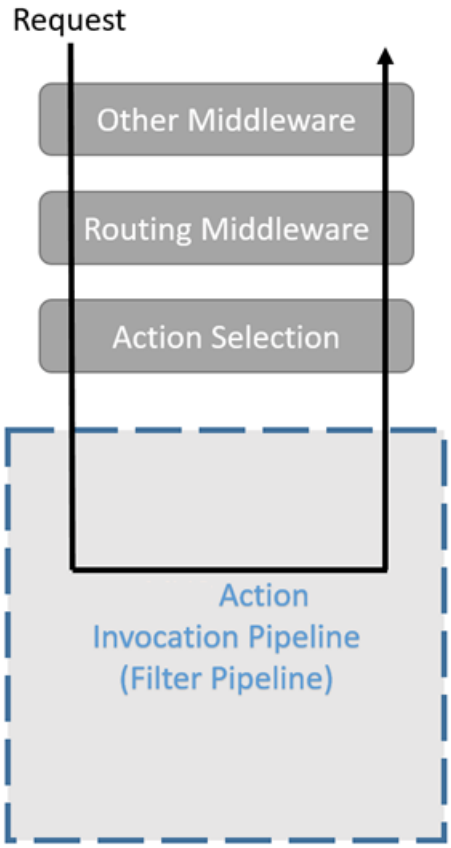
This document applies to Razor Pages, API controllers, and controllers with views. Filters don't work directly with Razor components. A filter can only indirectly affect a component when:

- The component is embedded in a page or view.
- The page or controller/view uses the filter.

View or download sample (how to download).

How filters work

Filters run within the *ASP.NET Core action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after ASP.NET Core selects the action to execute.

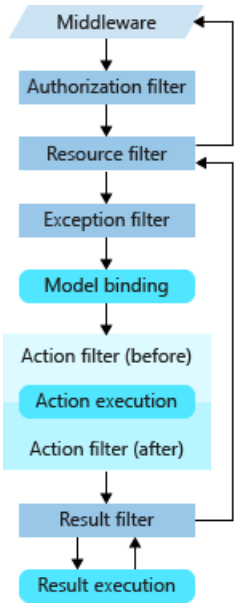


Filter types

Each filter type is executed at a different stage in the filter pipeline:

- Authorization filters run first and are used to determine whether the user is authorized for the request. Authorization filters short-circuit the pipeline if the request is not authorized.
- Resource filters:
  - Run after authorization.
  - OnResourceExecuting runs code before the rest of the filter pipeline. For example, `OnResourceExecuting` runs code before model binding.
  - OnResourceExecuted runs code after the rest of the pipeline has completed.
- Action filters:
  - Run code immediately before and after an action method is called.
  - Can change the arguments passed into an action.
  - Can change the result returned from the action.
  - Are **not** supported in Razor Pages.
- Exception filters apply global policies to unhandled exceptions that occur before the response body has been written to.
- Result filters run code immediately before and after the execution of action results. They run only when the action method has executed successfully. They are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline.



Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions.

Synchronous filters run code before and after their pipeline stage. For example, OnActionExecuting is called before the action method is called. OnActionExecuted is called after the action method returns.

```
public class MySampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }
}
```

In the preceding code, MyDebug is a utility function in the sample download.

Asynchronous filters define an `On-Stage-ExecutionAsync` method. For example, OnActionExecutionAsync:

```
public class SampleAsyncActionFilter : IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context,
        ActionExecutionDelegate next)
    {
        // Do something before the action executes.

        // next() calls the action method.
        var resultContext = await next();
        // resultContext.Result is set.
        // Do something after the action executes.
    }
}
```

In the preceding code, the `SampleAsyncActionFilter` has an ActionExecutionDelegate (`next`) that executes the action method.

Multiple filter stages

Interfaces for multiple filter stages can be implemented in a single class. For example, the ActionFilterAttribute class implements:

- Synchronous: IActionFilter and IResultFilter
- Asynchronous: IAsyncActionFilter and IAsyncResultFilter
- IOrderedFilter

Implement **either** the synchronous or the async version of a filter interface, **not** both. The runtime checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both asynchronous and synchronous interfaces are implemented in one class, only the async method is called. When using abstract classes like ActionFilterAttribute, override only the synchronous methods or the asynchronous methods for each filter type.

Built-in filter attributes

ASP.NET Core includes built-in attribute-based filters that can be subclassed and customized. For example, the following result filter adds a header to the response:

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string _value;

    public AddHeaderAttribute(string name, string value)
    {
        _name = name;
        _value = value;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add( _name, new string[] { _value });
        base.OnResultExecuting(context);
    }
}
```

Attributes allow filters to accept arguments, as shown in the preceding example. Apply the `AddHeaderAttribute` to a controller or action method and specify the name and value of the HTTP header:

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }
```

Use a tool such as the browser developer tools to examine the headers. Under **Response Headers**, `author: Rick Anderson` is displayed.

The following code implements an `ActionFilterAttribute` that:

- Reads the title and name from the configuration system. Unlike the previous sample, the following code doesn't require filter parameters to be added to the code.
- Adds the title and name to the response header.

```
public class MyActionFilterAttribute : ActionFilterAttribute
{
    private readonly PositionOptions _settings;

    public MyActionFilterAttribute(IOptions<PositionOptions> options)
    {
        _settings = options.Value;
        Order = 1;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_settings.Title,
                                                 new string[] { _settings.Name });
        base.OnResultExecuting(context);
    }
}
```

The configuration options are provided from the configuration system using the options pattern. For example, from the *appsettings.json* file:

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

In the `StartUp.ConfigureServices`:

- The `PositionOptions` class is added to the service container with the `"Position"` configuration area.
- The `MyActionFilterAttribute` is added to the service container.

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<PositionOptions>(
             Configuration.GetSection("Position"));
    services.AddScoped<MyActionFilterAttribute>();

    services.AddControllersWithViews();
}
```

The following code shows the `PositionOptions` class:

```
public class PositionOptions
{
    public string Title { get; set; }
    public string Name { get; set; }
}
```

The following code applies the `MyActionFilterAttribute` to the `Index2` method:

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }

    [ServiceFilter(typeof(MyActionFilterAttribute))]
    public IActionResult Index2()
    {
        return Content("Header values by configuration.");
    }
```

Under **Response Headers**, `author: Rick Anderson`, and `Editor: Joe Smith` is displayed when the `Sample/Index2` endpoint is called.

The following code applies the `MyActionFilterAttribute` and the `AddHeaderAttribute` to the Razor Page:

```
[AddHeader("Author", "Rick Anderson")]
[ServiceFilter(typeof(MyActionFilterAttribute))]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }
}
```

Filters cannot be applied to Razor Page handler methods. They can be applied either to the Razor Page model or globally.

Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

- ActionFilterAttribute
- ExceptionFilterAttribute
- ResultFilterAttribute
- FormatFilterAttribute
- ServiceFilterAttribute
- TypeFilterAttribute

Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*:

- Using an attribute on a controller action. Filter attributes cannot be applied to Razor Pages handler methods.
- Using an attribute on a controller or Razor Page.
- Globally for all controllers, actions, and Razor Pages as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
   {
        options.Filters.Add(typeof(MySampleActionFilter));
    });
}
```

Default order of execution

When there are multiple filters for a particular stage of the pipeline, scope determines the default order of filter execution. Global filters surround class filters, which in turn surround method filters.

As a result of filter nesting, the *after* code of filters runs in the reverse order of the *before* code. The filter sequence:

- The *before* code of global filters.
  - The *before* code of controller and Razor Page filters.
    - The *before* code of action method filters.
    - The *after* code of action method filters.
  - The *after* code of controller and Razor Page filters.
- The *after* code of global filters.

The following example that illustrates the order in which filter methods are called for synchronous action filters.

SEQUENCE	FILTER SCOPE	FILTER METHOD
1	Global	`OnActionExecuting`
2	Controller or Razor Page	`OnActionExecuting`
3	Method	`OnActionExecuting`
4	Method	`OnActionExecuted`
5	Controller or Razor Page	`OnActionExecuted`
6	Global	`OnActionExecuted`

Controller level filters

Every controller that inherits from the Controller base class includes Controller.OnActionExecuting, Controller.OnActionExecutionAsync, and Controller.OnActionExecuted `OnActionExecuted` methods. These methods:

- Wrap the filters that run for a given action.
- `OnActionExecuting` is called before any of the action's filters.
- `OnActionExecuted` is called after all of the action filters.
- `OnActionExecutionAsync` is called before any of the action's filters. Code in the filter after `next` runs after the action method.

For example, in the download sample, `MySampleActionFilter` is applied globally in startup.

The `TestController`:

- Applies the `SampleActionFilterAttribute` (`[SampleActionFilter]`) to the `FilterTest2` action.
- Overrides `OnActionExecuting` and `OnActionExecuted`.

```
public class TestController : Controller
{
    [SampleActionFilter(Order = int.MinValue)]
    public IActionResult FilterTest2()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuted(context);
    }
}
```

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

Navigating to `https://localhost:5001/Test2/FilterTest2` runs the following code:

- `TestController.OnActionExecuting`
  - `MySampleActionFilter.OnActionExecuting`
    - `SampleActionFilterAttribute.OnActionExecuting`
      - `TestController.FilterTest2`
    - `SampleActionFilterAttribute.OnActionExecuted`
  - `MySampleActionFilter.OnActionExecuted`
- `TestController.OnActionExecuted`

Controller level filters set the Order property to `int.MinValue`. Controller level filters can **not** be set to run after filters applied to methods. Order is explained in the next section.

For Razor Pages, see Implement Razor Page filters by overriding filter methods.

Overriding the default order

The default sequence of execution can be overridden by implementing IOrderedFilter. `IOrderedFilter` exposes the Order property that takes precedence over scope to determine the order of execution. A filter with a lower `Order` value:

- Runs the *before* code before that of a filter with a higher value of `Order`.
- Runs the *after* code after that of a filter with a higher `Order` value.

The `Order` property is set with a constructor parameter:

```
[SampleActionFilter(Order = int.MinValue)]
```

Consider the two action filters in the following controller:

```
[MyAction2Filter]
public class Test2Controller : Controller
{
    public IActionResult FilterTest2()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuted(context);
    }
}
```

A global filter is added in `StartUp.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
   {
        options.Filters.Add(typeof(MySampleActionFilter));
    });
}
```

The 3 filters run in the following order:

- `Test2Controller.OnActionExecuting`
  - `MySampleActionFilter.OnActionExecuting`
    - `MyAction2FilterAttribute.OnActionExecuting`
      - `Test2Controller.FilterTest2`
    - `MySampleActionFilter.OnActionExecuted`
  - `MyAction2FilterAttribute.OnResultExecuting`
- `Test2Controller.OnActionExecuted`

The `Order` property overrides scope when determining the order in which filters run. Filters are sorted first by order, then scope is used to break ties. All of the built-in filters implement `IOrderedFilter` and set the default `Order` value to 0. As mentioned previously, controller level filters set the Order property to `int.MinValue` For built-in filters, scope determines order unless `Order` is set to a non-zero value.

In the preceding code, `MySampleActionFilter` has global scope so it runs before `MyAction2FilterAttribute`, which has controller scope. To make `MyAction2FilterAttribute` run first, set the order to `int.MinValue`:

```
[MyAction2Filter(int.MinValue)]
public class Test2Controller : Controller
{
    public IActionResult FilterTest2()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuted(context);
    }
}
```

To make the global filter `MySampleActionFilter` run first, set `Order` to `int.MinValue`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
   {
        options.Filters.Add(typeof(MySampleActionFilter),
                            int.MinValue);
    });
}
```

Cancellation and short-circuiting

The filter pipeline can be short-circuited by setting the Result property on the ResourceExecutingContext parameter provided to the filter method. For instance, the following Resource filter prevents the rest of the pipeline from executing:

```
public class ShortCircuitingResourceFilterAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        context.Result = new ContentResult()
        {
            Content = "Resource unavailable - header not set."
        };
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

In the following code, both the `ShortCircuitingResourceFilter` and the `AddHeader` filter target the `SomeResource` action method. The `ShortCircuitingResourceFilter`:

- Runs first, because it's a Resource Filter and `AddHeader` is an Action Filter.
- Short-circuits the rest of the pipeline.

Therefore the `AddHeader` filter never runs for the `SomeResource` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilter` ran first. The `ShortCircuitingResourceFilter` runs first because of its filter type, or by explicit use of `Order` property.

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }
```

Dependency injection

Filters can be added by type or by instance. If an instance is added, that instance is used for every request. If a type is added, it's type-activated. A type-activated filter means:

- An instance is created for each request.
- Any constructor dependencies are populated by dependency injection (DI).

Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by dependency injection (DI). Constructor dependencies cannot be provided by DI because:

- Attributes must have their constructor parameters supplied where they're applied.
- This is a limitation of how attributes work.

The following filters support constructor dependencies provided from DI:

- ServiceFilterAttribute
- TypeFilterAttribute
- IFilterFactory implemented on the attribute.

The preceding filters can be applied to a controller or action method:

Loggers are available from DI. However, avoid creating and using filters purely for logging purposes. The built-in framework logging typically provides what's needed for logging. Logging added to filters:

- Should focus on business domain concerns or behavior specific to the filter.
- Should **not** log actions or other framework events. The built-in filters log actions and framework events.

ServiceFilterAttribute

Service filter implementation types are registered in `ConfigureServices`. A ServiceFilterAttribute retrieves an instance of the filter from DI.

The following code shows the `AddHeaderResultServiceFilter`:

```
public class AddHeaderResultServiceFilter : IResultFilter
{
    private ILogger _logger;
    public AddHeaderResultServiceFilter(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderResultServiceFilter>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation("Header added: {HeaderName}", headerName);
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has started.
        _logger.LogInformation("AddHeaderResultServiceFilter.OnResultExecuted");
    }
}
```

In the following code, `AddHeaderResultServiceFilter` is added to the DI container:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add service filters.
    services.AddScoped<AddHeaderResultServiceFilter>();
    services.AddScoped<SampleActionFilterAttribute>();

    services.AddControllersWithViews(options =>
   {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters"));         // An instance
        options.Filters.Add(typeof(MySampleActionFilter));         // By type
        options.Filters.Add(new SampleGlobalActionFilter());       // An instance
    });
}
```

In the following code, the `ServiceFilter` attribute retrieves an instance of the `AddHeaderResultServiceFilter` filter from DI:

```
[ServiceFilter(typeof(AddHeaderResultServiceFilter))]
public IActionResult Index()
{
    return View();
}
```

When using `ServiceFilterAttribute`, setting ServiceFilterAttribute.IsReusable:

- Provides a hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime doesn't guarantee:
  - That a single instance of the filter will be created.
  - The filter will not be re-requested from the DI container at some later point.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

ServiceFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from DI.

TypeFilterAttribute

TypeFilterAttribute is similar to ServiceFilterAttribute, but its type isn't resolved directly from the DI container. It instantiates the type by using Microsoft.Extensions.DependencyInjection.ObjectFactory.

Because `TypeFilterAttribute` types aren't resolved directly from the DI container:

- Types that are referenced using the `TypeFilterAttribute` don't need to be registered with the DI container. They do have their dependencies fulfilled by the DI container.
- `TypeFilterAttribute` can optionally accept constructor arguments for the type.

When using `TypeFilterAttribute`, setting TypeFilterAttribute.IsReusable:

- Provides hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime provides no guarantees that a single instance of the filter will be created.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

The following example shows how to pass arguments to a type using `TypeFilterAttribute`:

```
[TypeFilter(typeof(LogConstantFilter),
    Arguments = new object[] { "Method 'Hi' called" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

Authorization filters

Authorization filters:

- Are the first filters run in the filter pipeline.
- Control access to action methods.
- Have a before method, but no after method.

Custom authorization filters require a custom authorization framework. Prefer configuring the authorization policies or writing a custom authorization policy over writing a custom filter. The built-in authorization filter:

- Calls the authorization system.
- Does not authorize requests.

Do **not** throw exceptions within authorization filters:

- The exception will not be handled.
- Exception filters will not handle the exception.

Consider issuing a challenge when an exception occurs in an authorization filter.

Learn more about Authorization.

Resource filters

Resource filters:

- Implement either the IResourceFilter or IAsyncResourceFilter interface.
- Execution wraps most of the filter pipeline.
- Only Authorization filters run before resource filters.

Resource filters are useful to short-circuit most of the pipeline. For example, a caching filter can avoid the rest

of the pipeline on a cache hit.

Resource filter examples:

- The short-circuiting resource filter shown previously.
- DisableFormValueModelBindingAttribute:
  - Prevents model binding from accessing the form data.
  - Used for large file uploads to prevent the form data from being read into memory.

Action filters

Action filters do **not** apply to Razor Pages. Razor Pages supports IPageFilter and IAsyncPageFilter . For more information, see Filter methods for Razor Pages.

Action filters:

- Implement either the IActionFilter or IAsyncActionFilter interface.
- Their execution surrounds the execution of action methods.

The following code shows a sample action filter:

```
public class MySampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }
}
```

The ActionExecutingContext provides the following properties:

- ActionArguments - enables reading the inputs to an action method.
- Controller - enables manipulating the controller instance.
- Result - setting `Result` short-circuits execution of the action method and subsequent action filters.

Throwing an exception in an action method:

- Prevents running of subsequent filters.
- Unlike setting `Result`, is treated as a failure instead of a successful result.

The ActionExecutedContext provides `Controller` and `Result` plus the following properties:

- Canceled - True if the action execution was short-circuited by another filter.
- Exception - Non-null if the action or a previously run action filter threw an exception. Setting this property to null:
  - Effectively handles the exception.
  - `Result` is executed as if it was returned from the action method.

For an `IAsyncActionFilter`, a call to the ActionExecutionDelegate:

- Executes any subsequent action filters and the action method.
- Returns `ActionExecutedContext`.

To short-circuit, assign Microsoft.AspNetCore.Mvc.Filters.ActionExecutingContext.Result to a result instance and don't call `next` (the `ActionExecutionDelegate`).

The framework provides an abstract ActionFilterAttribute that can be subclassed.

The `OnActionExecuting` action filter can be used to:

- Validate model state.
- Return an error if the state is invalid.

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext
                                           context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(
                                                context.ModelState);
        }
    }
```

> **NOTE**
> Controllers annotated with the `[ApiController]` attribute automatically validate model state and return a 400 response. For more information, see Automatic HTTP 400 responses.

The `OnActionExecuted` method runs after the action method:

- And can see and manipulate the results of the action through the Result property.
- Canceled is set to true if the action execution was short-circuited by another filter.
- Exception is set to a non-null value if the action or a subsequent action filter threw an exception. Setting `Exception` to null:
  - Effectively handles an exception.
  - `ActionExecutedContext.Result` is executed as if it were returned normally from the action method.

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext
                                           context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(
                                                context.ModelState);
        }
    }


    public override void OnActionExecuted(ActionExecutedContext
                                          context)
    {
        var result = context.Result;
        // Do something with Result.
        if (context.Canceled == true)
        {
            // Action execution was short-circuited by another filter.
        }

        if(context.Exception != null)
        {
            // Exception thrown by action or action filter.
            // Set to null to handle the exception.
            context.Exception = null;
        }
        base.OnActionExecuted(context);
    }
}
```

Exception filters

Exception filters:

- Implement IExceptionFilter or IAsyncExceptionFilter.
- Can be used to implement common error handling policies.

The following sample exception filter uses a custom error view to display details about exceptions that occur when the app is in development:

```
public class CustomExceptionFilter : IExceptionFilter
{
    private readonly IWebHostEnvironment _hostingEnvironment;
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionFilter(
        IWebHostEnvironment hostingEnvironment,
        IModelMetadataProvider modelMetadataProvider)
    {
        _hostingEnvironment = hostingEnvironment;
        _modelMetadataProvider = modelMetadataProvider;
    }

    public void OnException(ExceptionContext context)
    {
        if (!_hostingEnvironment.IsDevelopment())
        {
            return;
        }
        var result = new ViewResult {ViewName = "CustomError"};
        result.ViewData = new ViewDataDictionary(_modelMetadataProvider,
                                                 context.ModelState);
        result.ViewData.Add("Exception", context.Exception);
        // TODO: Pass additional detailed data via ViewData
        context.Result = result;
    }
}
```

The following code tests the exception filter:

```
[TypeFilter(typeof(CustomExceptionFilter))]
public class FailingController : Controller
{
    [AddHeader("Failing Controller",
               "Won't appear when exception is handled")]
    public IActionResult Index()
    {
        throw new Exception("Testing custom exception filter.");
    }
}
```

Exception filters:

- Don't have before and after events.
- Implement OnException or OnExceptionAsync.
- Handle unhandled exceptions that occur in Razor Page or controller creation, model binding, action filters, or action methods.
- Do **not** catch exceptions that occur in resource filters, result filters, or MVC result execution.

To handle an exception, set the ExceptionHandled property to `true` or write a response. This stops propagation of the exception. An exception filter can't turn an exception into a "success". Only an action filter can do that.

Exception filters:

- Are good for trapping exceptions that occur within actions.
- Are not as flexible as error handling middleware.

Prefer middleware for exception handling. Use exception filters only where error handling *differs* based on which action method is called. For example, an app might have action methods for both API endpoints and for views/HTML. The API endpoints could return error information as JSON, while the view-based actions could

return an error page as HTML.

Result filters

Result filters:

- Implement an interface:
  - IResultFilter or IAsyncResultFilter
  - IAlwaysRunResultFilter or IAsyncAlwaysRunResultFilter
- Their execution surrounds the execution of action results.

IResultFilter and IAsyncResultFilter

The following code shows a result filter that adds an HTTP header:

```
public class AddHeaderResultServiceFilter : IResultFilter
{
    private ILogger _logger;
    public AddHeaderResultServiceFilter(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderResultServiceFilter>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation("Header added: {HeaderName}", headerName);
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has started.
        _logger.LogInformation("AddHeaderResultServiceFilter.OnResultExecuted");
    }
}
```

The kind of result being executed depends on the action. An action returning a view includes all razor processing as part of the ViewResult being executed. An API method might perform some serialization as part of the execution of the result. Learn more about action results.

Result filters are only executed when an action or action filter produces an action result. Result filters are not executed when:

- An authorization filter or resource filter short-circuits the pipeline.
- An exception filter handles an exception by producing an action result.

The Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuting method can short-circuit execution of the action result and subsequent result filters by setting Microsoft.AspNetCore.Mvc.Filters.ResultExecutingContext.Cancel to `true`. Write to the response object when short-circuiting to avoid generating an empty response. Throwing an exception in `IResultFilter.OnResultExecuting`:

- Prevents execution of the action result and subsequent filters.
- Is treated as a failure instead of a successful result.

When the Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuted method runs, the response has probably already been sent to the client. If the response has already been sent to the client, it cannot be changed.

`ResultExecutedContext.Canceled` is set to `true` if the action result execution was short-circuited by another filter.

`ResultExecutedContext.Exception` is set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `Exception` to null effectively handles an exception and prevents the exception from being thrown again later in the pipeline. There is no reliable way to write data to a response when handling an exception in a result filter. If the headers have been flushed to the client when an action result throws an exception, there's no reliable mechanism to send a failure code.

For an IAsyncResultFilter, a call to `await next` on the ResultExecutionDelegate executes any subsequent result filters and the action result. To short-circuit, set ResultExecutingContext.Cancel to `true` and don't call the `ResultExecutionDelegate`:

```
public class MyAsyncResponseFilter : IAsyncResultFilter
{
    public async Task OnResultExecutionAsync(ResultExecutingContext context,
                                             ResultExecutionDelegate next)
    {
        if (!(context.Result is EmptyResult))
        {
            await next();
        }
        else
        {
            context.Cancel = true;
        }

    }
}
```

The framework provides an abstract `ResultFilterAttribute` that can be subclassed. The AddHeaderAttribute class shown previously is an example of a result filter attribute.

IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter

The IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter interfaces declare an IResultFilter implementation that runs for all action results. This includes action results produced by:

- Authorization filters and resource filters that short-circuit.
- Exception filters.

For example, the following filter always runs and sets an action result (ObjectResult) with a *422 Unprocessable Entity* status code when content negotiation fails:

```
public class UnprocessableResultFilter : Attribute, IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        if (context.Result is StatusCodeResult statusCodeResult &&
            statusCodeResult.StatusCode == 415)
        {
            context.Result = new ObjectResult("Can't process this!")
            {
                StatusCode = 422,
            };
        }
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

IFilterFactory

IFilterFactory implements IFilterMetadata. Therefore, an `IFilterFactory` instance can be used as an `IFilterMetadata` instance anywhere in the filter pipeline. When the runtime prepares to invoke the filter, it attempts to cast it to an `IFilterFactory`. If that cast succeeds, the CreateInstance method is called to create the `IFilterMetadata` instance that is invoked. This provides a flexible design, since the precise filter pipeline doesn't need to be set explicitly when the app starts.

`IFilterFactory` can be implemented using custom attribute implementations as another approach to creating filters:

```
public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new InternalAddHeaderFilter();
    }

    private class InternalAddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "Internal", new string[] { "My header" });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

The filter is applied in the following code:

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }

    [ServiceFilter(typeof(MyActionFilterAttribute))]
    public IActionResult Index2()
    {
        return Content("Header values by configuration.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header is set.");
    }

    [AddHeaderWithFactory]
    public IActionResult HeaderWithFactory()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }
}
```

Test the preceding code by running the download sample:

- Invoke the F12 developer tools.
- Navigate to `https://localhost:5001/Sample/HeaderWithFactory`.

The F12 developer tools display the following response headers added by the sample code:

- **author:** `Rick Anderson`
- **globaladdheader:** `Result filter added to MvcOptions.Filters`
- **internal:** `My header`

The preceding code creates the **internal:** `My header` response header.

IFilterFactory implemented on an attribute

Filters that implement `IFilterFactory` are useful for filters that:

- Don't require passing parameters.
- Have constructor dependencies that need to be filled by DI.

TypeFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from the services container (DI).

```
public class SampleActionFilterAttribute : TypeFilterAttribute
{
    public SampleActionFilterAttribute()
                        :base(typeof(SampleActionFilterImpl))
    {
    }

    private class SampleActionFilterImpl : IActionFilter
    {
        private readonly ILogger _logger;
        public SampleActionFilterImpl(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
           _logger.LogInformation("SampleActionFilterAttribute.OnActionExecuting");
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            _logger.LogInformation("SampleActionFilterAttribute.OnActionExecuted");
        }
    }
}
```

The following code shows three approaches to applying the `[SampleActionFilter]`:

```
[SampleActionFilter]
public IActionResult FilterTest()
{
    return Content("From FilterTest");
}

[TypeFilter(typeof(SampleActionFilterAttribute))]
public IActionResult TypeFilterTest()
{
    return Content("From TypeFilterTest");
}

// ServiceFilter must be registered in ConfigureServices or
// System.InvalidOperationException: No service for type '<filter>'
// has been registered. Is thrown.
[ServiceFilter(typeof(SampleActionFilterAttribute))]
public IActionResult ServiceFilterTest()
{
    return Content("From ServiceFilterTest");
}
```

In the preceding code, decorating the method with `[SampleActionFilter]` is the preferred approach to applying the `SampleActionFilter`.

Using middleware in the filter pipeline

Resource filters work like middleware in that they surround the execution of everything that comes later in the pipeline. But filters differ from middleware in that they're part of the runtime, which means that they have access to context and constructs.

To use middleware as a filter, create a type with a `Configure` method that specifies the middleware to inject into the filter pipeline. The following example uses the localization middleware to establish the current culture for a request:

```
public class LocalizationPipeline
{
    public void Configure(IApplicationBuilder applicationBuilder)
    {
        var supportedCultures = new[]
        {
            new CultureInfo("en-US"),
            new CultureInfo("fr")
        };

        var options = new RequestLocalizationOptions
        {
            DefaultRequestCulture = new RequestCulture(
                                    culture: "en-US",
                                    uiCulture: "en-US"),
            SupportedCultures = supportedCultures,
            SupportedUICultures = supportedCultures
        };
        options.RequestCultureProviders = new[]
            { new RouteDataRequestCultureProvider() {
                Options = options } };

        applicationBuilder.UseRequestLocalization(options);
    }
}
```

Use the MiddlewareFilterAttribute to run the middleware:

```
[Route("{culture}/[controller]/[action]")]
[MiddlewareFilter(typeof(LocalizationPipeline))]
public IActionResult CultureFromRouteData()
{
    return Content(
          $"CurrentCulture:{CultureInfo.CurrentCulture.Name},"
        + $"CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");
}
```

Middleware filters run at the same stage of the filter pipeline as Resource filters, before model binding and after the rest of the pipeline.

Next actions

- See Filter methods for Razor Pages.
- To experiment with filters, download, test, and modify the GitHub sample.

By Kirk Larkin, Rick Anderson, Tom Dykstra, and Steve Smith

Filters in ASP.NET Core allow code to be run before or after specific stages in the request processing pipeline.

Built-in filters handle tasks such as:

- Authorization (preventing access to resources a user isn't authorized for).
- Response caching (short-circuiting the request pipeline to return a cached response).

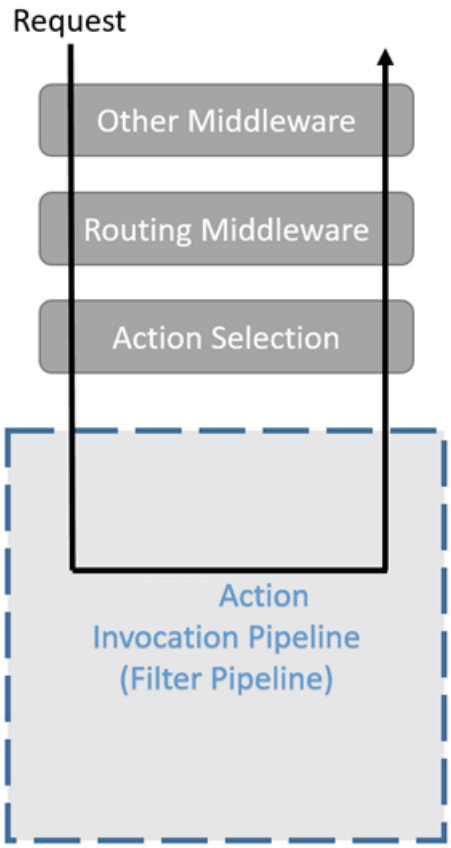
Custom filters can be created to handle cross-cutting concerns. Examples of cross-cutting concerns include error handling, caching, configuration, authorization, and logging. Filters avoid duplicating code. For example, an error handling exception filter could consolidate error handling.

This document applies to Razor Pages, API controllers, and controllers with views.

View or download sample (how to download).

How filters work

Filters run within the *ASP.NET Core action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after ASP.NET Core selects the action to execute.

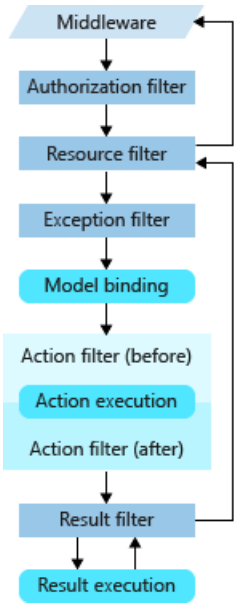


Filter types

Each filter type is executed at a different stage in the filter pipeline:

- Authorization filters run first and are used to determine whether the user is authorized for the request. Authorization filters short-circuit the pipeline if the request is unauthorized.
- Resource filters:
  - Run after authorization.
  - OnResourceExecuting can run code before the rest of the filter pipeline. For example, `OnResourceExecuting` can run code before model binding.
  - OnResourceExecuted can run code after the rest of the pipeline has completed.
- Action filters can run code immediately before and after an individual action method is called. They can be used to manipulate the arguments passed into an action and the result returned from the action. Action filters are **not** supported in Razor Pages.
- Exception filters are used to apply global policies to unhandled exceptions that occur before anything has been written to the response body.
- Result filters can run code immediately before and after the execution of individual action results. They run only when the action method has executed successfully. They are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline.



Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions.

Synchronous filters can run code before (`On-Stage-Executing`) and after (`On-Stage-Executed`) their pipeline stage. For example, `OnActionExecuting` is called before the action method is called. `OnActionExecuted` is called after the action method returns.

```
public class MySampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

Asynchronous filters define an `On-Stage-ExecutionAsync` method:

```
public class SampleAsyncActionFilter : IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context,
        ActionExecutionDelegate next)
    {
        // Do something before the action executes.

        // next() calls the action method.
        var resultContext = await next();
        // resultContext.Result is set.
        // Do something after the action executes.
    }
}
```

In the preceding code, the `SampleAsyncActionFilter` has an ActionExecutionDelegate (`next`) that executes the action method. Each of the `On-Stage-ExecutionAsync` methods take a `FilterType-ExecutionDelegate` that executes the filter's pipeline stage.

Multiple filter stages

Interfaces for multiple filter stages can be implemented in a single class. For example, the ActionFilterAttribute class implements `IActionFilter`, `IResultFilter`, and their async equivalents.

Implement **either** the synchronous or the async version of a filter interface, **not** both. The runtime checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both asynchronous and synchronous interfaces are implemented in one class, only the async method is called. When using abstract classes like ActionFilterAttribute override only the synchronous methods or the async method for each filter type.

Built-in filter attributes

ASP.NET Core includes built-in attribute-based filters that can be subclassed and customized. For example, the following result filter adds a header to the response:

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string _value;

    public AddHeaderAttribute(string name, string value)
    {
        _name = name;
        _value = value;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add( _name, new string[] { _value });
        base.OnResultExecuting(context);
    }
}
```

Attributes allow filters to accept arguments, as shown in the preceding example. Apply the `AddHeaderAttribute` to a controller or action method and specify the name and value of the HTTP header:

```
[AddHeader("Author", "Joe Smith")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header is set.");
    }
```

Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

- ActionFilterAttribute
- ExceptionFilterAttribute
- ResultFilterAttribute
- FormatFilterAttribute
- ServiceFilterAttribute

- TypeFilterAttribute

Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*:

- Using an attribute on an action.
- Using an attribute on a controller.
- Globally for all controllers and actions as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters"));         // An instance
        options.Filters.Add(typeof(MySampleActionFilter));         // By type
        options.Filters.Add(new SampleGlobalActionFilter());       // An instance
    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

The preceding code adds three filters globally using the MvcOptions.Filters collection.

Default order of execution

When there are multiple filters *of the same type*, scope determines the default order of filter execution. Global filters surround class filters. Class filters surround method filters.

As a result of filter nesting, the *after* code of filters runs in the reverse order of the *before* code. The filter sequence:

- The *before* code of global filters.
  - The *before* code of controller filters.
    - The *before* code of action method filters.
    - The *after* code of action method filters.
  - The *after* code of controller filters.
- The *after* code of global filters.

The following example that illustrates the order in which filter methods are called for synchronous action filters.

SEQUENCE	FILTER SCOPE	FILTER METHOD
1	Global	`OnActionExecuting`
2	Controller	`OnActionExecuting`
3	Method	`OnActionExecuting`
4	Method	`OnActionExecuted`
5	Controller	`OnActionExecuted`
6	Global	`OnActionExecuted`

This sequence shows:

- The method filter is nested within the controller filter.
- The controller filter is nested within the global filter.

Controller and Razor Page level filters

Every controller that inherits from the Controller base class includes Controller.OnActionExecuting, Controller.OnActionExecutionAsync, and Controller.OnActionExecuted `OnActionExecuted` methods. These methods:

- Wrap the filters that run for a given action.
- `OnActionExecuting` is called before any of the action's filters.
- `OnActionExecuted` is called after all of the action filters.
- `OnActionExecutionAsync` is called before any of the action's filters. Code in the filter after `next` runs after the action method.

For example, in the download sample, `MySampleActionFilter` is applied globally in startup.

The `TestController`:

- Applies the `SampleActionFilterAttribute` (`[SampleActionFilter]`) to the `FilterTest2` action.
- Overrides `OnActionExecuting` and `OnActionExecuted`.

```
public class TestController : Controller
{
    [SampleActionFilter]
    public IActionResult FilterTest2()
    {
        return Content($"From FilterTest2");
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        base.OnActionExecuted(context);
    }
}
```

Navigating to `https://localhost:5001/Test/FilterTest2` runs the following code:

- `TestController.OnActionExecuting`
  - `MySampleActionFilter.OnActionExecuting`
    - `SampleActionFilterAttribute.OnActionExecuting`
      - `TestController.FilterTest2`
    - `SampleActionFilterAttribute.OnActionExecuted`
  - `MySampleActionFilter.OnActionExecuted`
- `TestController.OnActionExecuted`

For Razor Pages, see Implement Razor Page filters by overriding filter methods.

Overriding the default order

The default sequence of execution can be overridden by implementing IOrderedFilter. `IOrderedFilter` exposes the Order property that takes precedence over scope to determine the order of execution. A filter with a lower

`Order` value:

- Runs the *before* code before that of a filter with a higher value of `Order`.
- Runs the *after* code after that of a filter with a higher `Order` value.

The `Order` property can be set with a constructor parameter:

```
[MyFilter(Name = "Controller Level Attribute", Order=1)]
```

Consider the same 3 action filters shown in the preceding example. If the `Order` property of the controller and global filters is set to 1 and 2 respectively, the order of execution is reversed.

SEQUENCE	FILTER SCOPE	`ORDER` PROPERTY	FILTER METHOD
1	Method	0	`OnActionExecuting`
2	Controller	1	`OnActionExecuting`
3	Global	2	`OnActionExecuting`
4	Global	2	`OnActionExecuted`
5	Controller	1	`OnActionExecuted`
6	Method	0	`OnActionExecuted`

The `Order` property overrides scope when determining the order in which filters run. Filters are sorted first by order, then scope is used to break ties. All of the built-in filters implement `IOrderedFilter` and set the default `Order` value to 0. For built-in filters, scope determines order unless `Order` is set to a non-zero value.

Cancellation and short-circuiting

The filter pipeline can be short-circuited by setting the Result property on the ResourceExecutingContext parameter provided to the filter method. For instance, the following Resource filter prevents the rest of the pipeline from executing:

```
public class ShortCircuitingResourceFilterAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        context.Result = new ContentResult()
        {
            Content = "Resource unavailable - header not set."
        };
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

In the following code, both the `ShortCircuitingResourceFilter` and the `AddHeader` filter target the `SomeResource` action method. The `ShortCircuitingResourceFilter`:

- Runs first, because it's a Resource Filter and `AddHeader` is an Action Filter.

- Short-circuits the rest of the pipeline.

Therefore the `AddHeader` filter never runs for the `SomeResource` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilter` ran first. The `ShortCircuitingResourceFilter` runs first because of its filter type, or by explicit use of `Order` property.

```
[AddHeader("Author", "Joe Smith")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header is set.");
    }
```

Dependency injection

Filters can be added by type or by instance. If an instance is added, that instance is used for every request. If a type is added, it's type-activated. A type-activated filter means:

- An instance is created for each request.
- Any constructor dependencies are populated by dependency injection (DI).

Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by dependency injection (DI). Constructor dependencies cannot be provided by DI because:

- Attributes must have their constructor parameters supplied where they're applied.
- This is a limitation of how attributes work.

The following filters support constructor dependencies provided from DI:

- ServiceFilterAttribute
- TypeFilterAttribute
- IFilterFactory implemented on the attribute.

The preceding filters can be applied to a controller or action method:

Loggers are available from DI. However, avoid creating and using filters purely for logging purposes. The built-in framework logging typically provides what's needed for logging. Logging added to filters:

- Should focus on business domain concerns or behavior specific to the filter.
- Should **not** log actions or other framework events. The built in filters log actions and framework events.

ServiceFilterAttribute

Service filter implementation types are registered in `ConfigureServices`. A ServiceFilterAttribute retrieves an instance of the filter from DI.

The following code shows the `AddHeaderResultServiceFilter`:

```
public class AddHeaderResultServiceFilter : IResultFilter
{
    private ILogger _logger;
    public AddHeaderResultServiceFilter(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderResultServiceFilter>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation("Header added: {HeaderName}", headerName);
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has started.
    }
}
```

In the following code, `AddHeaderResultServiceFilter` is added to the DI container:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add service filters.
    services.AddScoped<AddHeaderResultServiceFilter>();
    services.AddScoped<SampleActionFilterAttribute>();

    services.AddMvc(options =>
    {
        options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
            "Result filter added to MvcOptions.Filters"));         // An instance
        options.Filters.Add(typeof(MySampleActionFilter));         // By type
        options.Filters.Add(new SampleGlobalActionFilter());       // An instance
    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

In the following code, the `ServiceFilter` attribute retrieves an instance of the `AddHeaderResultServiceFilter` filter from DI:

```
[ServiceFilter(typeof(AddHeaderResultServiceFilter))]
public IActionResult Index()
{
    return View();
}
```

When using `ServiceFilterAttribute`, setting ServiceFilterAttribute.IsReusable:

- Provides a hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime doesn't guarantee:
  - That a single instance of the filter will be created.
  - The filter will not be re-requested from the DI container at some later point.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

ServiceFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from DI.

TypeFilterAttribute

TypeFilterAttribute is similar to ServiceFilterAttribute, but its type isn't resolved directly from the DI container. It instantiates the type by using Microsoft.Extensions.DependencyInjection.ObjectFactory.

Because `TypeFilterAttribute` types aren't resolved directly from the DI container:

- Types that are referenced using the `TypeFilterAttribute` don't need to be registered with the DI container. They do have their dependencies fulfilled by the DI container.
- `TypeFilterAttribute` can optionally accept constructor arguments for the type.

When using `TypeFilterAttribute`, setting TypeFilterAttribute.IsReusable:

- Provides hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime provides no guarantees that a single instance of the filter will be created.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

The following example shows how to pass arguments to a type using `TypeFilterAttribute`:

```
[TypeFilter(typeof(LogConstantFilter),
    Arguments = new object[] { "Method 'Hi' called" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

```
public class LogConstantFilter : IActionFilter
{
    private readonly string _value;
    private readonly ILogger<LogConstantFilter> _logger;

    public LogConstantFilter(string value, ILogger<LogConstantFilter> logger)
    {
        _logger = logger;
        _value = value;
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        _logger.LogInformation(_value);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    { }
}
```

Authorization filters

Authorization filters:

- Are the first filters run in the filter pipeline.
- Control access to action methods.
- Have a before method, but no after method.

Custom authorization filters require a custom authorization framework. Prefer configuring the authorization policies or writing a custom authorization policy over writing a custom filter. The built-in authorization filter:

- Calls the authorization system.
- Does not authorize requests.

Do **not** throw exceptions within authorization filters:

- The exception will not be handled.
- Exception filters will not handle the exception.

Consider issuing a challenge when an exception occurs in an authorization filter.

Learn more about Authorization.

Resource filters

Resource filters:

- Implement either the IResourceFilter or IAsyncResourceFilter interface.
- Execution wraps most of the filter pipeline.
- Only Authorization filters run before resource filters.

Resource filters are useful to short-circuit most of the pipeline. For example, a caching filter can avoid the rest of the pipeline on a cache hit.

Resource filter examples:

- The short-circuiting resource filter shown previously.
- DisableFormValueModelBindingAttribute:
  - Prevents model binding from accessing the form data.
  - Used for large file uploads to prevent the form data from being read into memory.

Action filters

> **IMPORTANT**
> Action filters do **not** apply to Razor Pages. Razor Pages supports IPageFilter and IAsyncPageFilter . For more information, see Filter methods for Razor Pages.

Action filters:

- Implement either the IActionFilter or IAsyncActionFilter interface.
- Their execution surrounds the execution of action methods.

The following code shows a sample action filter:

```
public class MySampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

The ActionExecutingContext provides the following properties:

- ActionArguments - enables the inputs to an action method be read.
- Controller - enables manipulating the controller instance.
- Result - setting `Result` short-circuits execution of the action method and subsequent action filters.

Throwing an exception in an action method:

- Prevents running of subsequent filters.
- Unlike setting `Result`, is treated as a failure instead of a successful result.

The ActionExecutedContext provides `Controller` and `Result` plus the following properties:

- Canceled - True if the action execution was short-circuited by another filter.
- Exception - Non-null if the action or a previously run action filter threw an exception. Setting this property to null:
  - Effectively handles the exception.
  - `Result` is executed as if it was returned from the action method.

For an `IAsyncActionFilter`, a call to the ActionExecutionDelegate:

- Executes any subsequent action filters and the action method.
- Returns `ActionExecutedContext`.

To short-circuit, assign Microsoft.AspNetCore.Mvc.Filters.ActionExecutingContext.Result to a result instance and don't call `next` (the `ActionExecutionDelegate`).

The framework provides an abstract ActionFilterAttribute that can be subclassed.

The `OnActionExecuting` action filter can be used to:

- Validate model state.
- Return an error if the state is invalid.

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }
```

The `OnActionExecuted` method runs after the action method:

- And can see and manipulate the results of the action through the Result property.
- Canceled is set to true if the action execution was short-circuited by another filter.
- Exception is set to a non-null value if the action or a subsequent action filter threw an exception. Setting `Exception` to null:
  - Effectively handles an exception.
  - `ActionExecutedContext.Result` is executed as if it were returned normally from the action method.

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }


    public override void OnActionExecuted(ActionExecutedContext context)
    {
        var result = context.Result;
        // Do something with Result.
        if (context.Canceled == true)
        {
            // Action execution was short-circuited by another filter.
        }

        if(context.Exception != null)
        {
            // Exception thrown by action or action filter.
            // Set to null to handle the exception.
            context.Exception = null;
        }
        base.OnActionExecuted(context);
    }
}
```

Exception filters

Exception filters:

- Implement IExceptionFilter or IAsyncExceptionFilter.
- Can be used to implement common error handling policies.

The following sample exception filter uses a custom error view to display details about exceptions that occur when the app is in development:

```
public class CustomExceptionFilter : IExceptionFilter
{
    private readonly IHostingEnvironment _hostingEnvironment;
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionFilter(
        IHostingEnvironment hostingEnvironment,
        IModelMetadataProvider modelMetadataProvider)
    {
        _hostingEnvironment = hostingEnvironment;
        _modelMetadataProvider = modelMetadataProvider;
    }

    public void OnException(ExceptionContext context)
    {
        if (!_hostingEnvironment.IsDevelopment())
        {
            return;
        }
        var result = new ViewResult {ViewName = "CustomError"};
        result.ViewData = new ViewDataDictionary(_modelMetadataProvider,
                                                 context.ModelState);
        result.ViewData.Add("Exception", context.Exception);
        // TODO: Pass additional detailed data via ViewData
        context.Result = result;
    }
}
```

Exception filters:

- Don't have before and after events.
- Implement OnException or OnExceptionAsync.
- Handle unhandled exceptions that occur in Razor Page or controller creation, model binding, action filters, or action methods.
- Do **not** catch exceptions that occur in resource filters, result filters, or MVC result execution.

To handle an exception, set the ExceptionHandled property to `true` or write a response. This stops propagation of the exception. An exception filter can't turn an exception into a "success". Only an action filter can do that.

Exception filters:

- Are good for trapping exceptions that occur within actions.
- Are not as flexible as error handling middleware.

Prefer middleware for exception handling. Use exception filters only where error handling *differs* based on which action method is called. For example, an app might have action methods for both API endpoints and for views/HTML. The API endpoints could return error information as JSON, while the view-based actions could return an error page as HTML.

Result filters

Result filters:

- Implement an interface:
  - IResultFilter or IAsyncResultFilter
  - IAlwaysRunResultFilter or IAsyncAlwaysRunResultFilter
- Their execution surrounds the execution of action results.

IResultFilter and IAsyncResultFilter

The following code shows a result filter that adds an HTTP header:

```
public class AddHeaderResultServiceFilter : IResultFilter
{
    private ILogger _logger;
    public AddHeaderResultServiceFilter(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderResultServiceFilter>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation("Header added: {HeaderName}", headerName);
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has started.
    }
}
```

The kind of result being executed depends on the action. An action returning a view would include all razor processing as part of the ViewResult being executed. An API method might perform some serialization as part of the execution of the result. Learn more about action results.

Result filters are only executed when an action or action filter produces an action result. Result filters are not executed when:

- An authorization filter or resource filter short-circuits the pipeline.
- An exception filter handles an exception by producing an action result.

The Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuting method can short-circuit execution of the action result and subsequent result filters by setting Microsoft.AspNetCore.Mvc.Filters.ResultExecutingContext.Cancel to `true`. Write to the response object when short-circuiting to avoid generating an empty response. Throwing an exception in `IResultFilter.OnResultExecuting` will:

- Prevent execution of the action result and subsequent filters.
- Be treated as a failure instead of a successful result.

When the Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuted method runs, the response has likely already been sent to the client. If the response has already been sent to the client, it cannot be changed further.

`ResultExecutedContext.Canceled` is set to `true` if the action result execution was short-circuited by another filter.

`ResultExecutedContext.Exception` is set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `Exception` to null effectively handles an exception and prevents the exception from being rethrown by ASP.NET Core later in the pipeline. There is no reliable way to write data to a response when handling an exception in a result filter. If the headers have been flushed to the client when an action result throws an exception, there's no reliable mechanism to send a failure code.

For an IAsyncResultFilter, a call to `await next` on the ResultExecutionDelegate executes any subsequent result filters and the action result. To short-circuit, set ResultExecutingContext.Cancel to `true` and don't call the `ResultExecutionDelegate`:

```
public class MyAsyncResponseFilter : IAsyncResultFilter
{
    public async Task OnResultExecutionAsync(ResultExecutingContext context,
                                             ResultExecutionDelegate next)
    {
        if (!(context.Result is EmptyResult))
        {
            await next();
        }
        else
        {
            context.Cancel = true;
        }

    }
}
```

The framework provides an abstract `ResultFilterAttribute` that can be subclassed. The AddHeaderAttribute class shown previously is an example of a result filter attribute.

IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter

The IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter interfaces declare an IResultFilter implementation that runs for all action results. This includes action results produced by:

- Authorization filters and resource filters that short-circuit.
- Exception filters.

For example, the following filter always runs and sets an action result (ObjectResult) with a *422 Unprocessable Entity* status code when content negotiation fails:

```
public class UnprocessableResultFilter : Attribute, IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        if (context.Result is StatusCodeResult statusCodeResult &&
            statusCodeResult.StatusCode == 415)
        {
            context.Result = new ObjectResult("Can't process this!")
            {
                StatusCode = 422,
            };
        }
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

IFilterFactory

IFilterFactory implements IFilterMetadata. Therefore, an `IFilterFactory` instance can be used as an `IFilterMetadata` instance anywhere in the filter pipeline. When the runtime prepares to invoke the filter, it attempts to cast it to an `IFilterFactory`. If that cast succeeds, the CreateInstance method is called to create the `IFilterMetadata` instance that is invoked. This provides a flexible design, since the precise filter pipeline doesn't need to be set explicitly when the app starts.

`IFilterFactory` can be implemented using custom attribute implementations as another approach to creating filters:

```
public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new InternalAddHeaderFilter();
    }

    private class InternalAddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "Internal", new string[] { "My header" });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

The preceding code can be tested by running the download sample:

- Invoke the F12 developer tools.
- Navigate to `https://localhost:5001/Sample/HeaderWithFactory`.

The F12 developer tools display the following response headers added by the sample code:

- **author:** `Joe Smith`
- **globaladdheader:** `Result filter added to MvcOptions.Filters`
- **internal:** `My header`

The preceding code creates the **internal:** `My header` response header.

IFilterFactory implemented on an attribute

Filters that implement `IFilterFactory` are useful for filters that:

- Don't require passing parameters.
- Have constructor dependencies that need to be filled by DI.

TypeFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from the services container (DI).

```
public class SampleActionFilterAttribute : TypeFilterAttribute
{
    public SampleActionFilterAttribute():base(typeof(SampleActionFilterImpl))
    {
    }

    private class SampleActionFilterImpl : IActionFilter
    {
        private readonly ILogger _logger;
        public SampleActionFilterImpl(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            _logger.LogInformation("Business action starting...");
            // perform some business logic work

        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            // perform some business logic work
            _logger.LogInformation("Business action completed.");
        }
    }
}
```

The following code shows three approaches to applying the `[SampleActionFilter]`:

```
[SampleActionFilter]
public IActionResult FilterTest()
{
    return Content($"From FilterTest");
}

[TypeFilter(typeof(SampleActionFilterAttribute))]
public IActionResult TypeFilterTest()
{
    return Content($"From ServiceFilterTest");
}

// ServiceFilter must be registered in ConfigureServices or
// System.InvalidOperationException: No service for type '<filter>' has been registered.
// Is thrown.
[ServiceFilter(typeof(SampleActionFilterAttribute))]
public IActionResult ServiceFilterTest()
{
    return Content($"From ServiceFilterTest");
}
```

In the preceding code, decorating the method with `[SampleActionFilter]` is the preferred approach to applying the `SampleActionFilter`.

Using middleware in the filter pipeline

Resource filters work like middleware in that they surround the execution of everything that comes later in the pipeline. But filters differ from middleware in that they're part of the ASP.NET Core runtime, which means that they have access to ASP.NET Core context and constructs.

To use middleware as a filter, create a type with a `Configure` method that specifies the middleware to inject

into the filter pipeline. The following example uses the localization middleware to establish the current culture for a request:

```
public class LocalizationPipeline
{
    public void Configure(IApplicationBuilder applicationBuilder)
    {
        var supportedCultures = new[]
        {
            new CultureInfo("en-US"),
            new CultureInfo("fr")
        };

        var options = new RequestLocalizationOptions
        {

            DefaultRequestCulture = new RequestCulture(culture: "en-US",
                                                       uiCulture: "en-US"),
            SupportedCultures = supportedCultures,
            SupportedUICultures = supportedCultures
        };
        options.RequestCultureProviders = new[]
            { new RouteDataRequestCultureProvider() { Options = options } };

        applicationBuilder.UseRequestLocalization(options);
    }
}
```

Use the MiddlewareFilterAttribute to run the middleware:

```
[Route("{culture}/[controller]/[action]")]
[MiddlewareFilter(typeof(LocalizationPipeline))]
public IActionResult CultureFromRouteData()
{
    return Content($"CurrentCulture:{CultureInfo.CurrentCulture.Name},"
        + $"CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");
}
```

Middleware filters run at the same stage of the filter pipeline as Resource filters, before model binding and after the rest of the pipeline.

Next actions

- See Filter methods for Razor Pages.
- To experiment with filters, download, test, and modify the GitHub sample.

ASP.NET Core Razor SDK

9/22/2020 • 8 minutes to read • Edit Online

By Rick Anderson

Overview

The .NET Core 2.1 SDK or later includes the `Microsoft.NET.Sdk.Razor` MSBuild SDK (Razor SDK). The Razor SDK:

- Is required to build, package, and publish projects containing Razor files for ASP.NET Core MVC-based or Blazor projects.
- Includes a set of predefined targets, properties, and items that allow customizing the compilation of Razor (*.cshtml* or *.razor*) files.

The Razor SDK includes `Content` items with `Include` attributes set to the `**\*.cshtml` and `**\*.razor` globbing patterns. Matching files are published.

- Standardizes the experience around building, packaging, and publishing projects containing Razor files for ASP.NET Core MVC-based projects.
- Includes a set of predefined targets, properties, and items that allow customizing the compilation of Razor files.

The Razor SDK includes a `Content` item with an `Include` attribute set to the `**\*.cshtml` globbing pattern. Matching files are published.

Prerequisites

.NET Core 2.1 SDK or later

Use the Razor SDK

Most web apps aren't required to explicitly reference the Razor SDK.

To use the Razor SDK to build class libraries containing Razor views or Razor Pages, we recommend starting with the Razor class library (RCL) project template. An RCL that's used to build Blazor (*.razor*) files minimally requires a reference to the Microsoft.AspNetCore.Components package. An RCL that's used to build Razor views or pages (*.cshtml* files) minimally requires targeting `netcoreapp3.0` or later and has a `FrameworkReference` to the Microsoft.AspNetCore.App metapackage in its project file.

To use the Razor SDK to build class libraries containing Razor views or Razor Pages:

- Use `Microsoft.NET.Sdk.Razor` instead of `Microsoft.NET.Sdk`:

```
<Project SDK="Microsoft.NET.Sdk.Razor">
  <!-- omitted for brevity -->
</Project>
```

- Typically, a package reference to `Microsoft.AspNetCore.Mvc` is required to receive additional dependencies that are required to build and compile Razor Pages and Razor views. At a minimum, your project should add package references to:
  - `Microsoft.AspNetCore.Razor.Design`

- `Microsoft.AspNetCore.Mvc.Razor.Extensions`
- `Microsoft.AspNetCore.Mvc.Razor`

The `Microsoft.AspNetCore.Razor.Design` package provides the Razor compilation tasks and targets for the project.

The preceding packages are included in `Microsoft.AspNetCore.Mvc`. The following markup shows a project file that uses the Razor SDK to build Razor files for an ASP.NET Core Razor Pages app:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.1.3" />
  </ItemGroup>

</Project>
```

WARNING

The `Microsoft.AspNetCore.Razor.Design` and `Microsoft.AspNetCore.Mvc.Razor.Extensions` packages are included in the Microsoft.AspNetCore.App metapackage. However, the version-less `Microsoft.AspNetCore.App` package reference provides a metapackage to the app that doesn't include the latest version of `Microsoft.AspNetCore.Razor.Design`. Projects must reference a consistent version of `Microsoft.AspNetCore.Razor.Design` (or `Microsoft.AspNetCore.Mvc`) so that the latest build-time fixes for Razor are included. For more information, see this GitHub issue.

Properties

The following properties control the Razor's SDK behavior as part of a project build:

- `RazorCompileOnBuild`: When `true`, compiles and emits the Razor assembly as part of building the project. Defaults to `true`.
- `RazorCompileOnPublish`: When `true`, compiles and emits the Razor assembly as part of publishing the project. Defaults to `true`.

The properties and items in the following table are used to configure inputs and output to the Razor SDK.

WARNING

Starting with ASP.NET Core 3.0, MVC Views or Razor Pages aren't served by default if the `RazorCompileOnBuild` or `RazorCompileOnPublish` MSBuild properties in the project file are disabled. Applications must add an explicit reference to the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation package if the app relies on runtime compilation to process *.cshtml* files.

ITEMS	DESCRIPTION
`RazorGenerate`	Item elements (*.cshtml* files) that are inputs to code generation.
`RazorComponent`	Item elements (*.razor* files) that are inputs to Razor component code generation.

ITEMS	DESCRIPTION
`RazorCompile`	Item elements (*.cs* files) that are inputs to Razor compilation targets. Use this `ItemGroup` to specify additional files to be compiled into the Razor assembly.
`RazorTargetAssemblyAttribute`	Item elements used to code generate attributes for the Razor assembly. For example: `RazorAssemblyAttribute` `Include="System.Reflection.AssemblyMetadataAttribute"` `_Parameter1="BuildSource"` `_Parameter2="https://docs.microsoft.com/">`
`RazorEmbeddedResource`	Item elements added as embedded resources to the generated Razor assembly.

PROPERTY	DESCRIPTION
`RazorTargetName`	File name (without extension) of the assembly produced by Razor.
`RazorOutputPath`	The Razor output directory.
`RazorCompileToolset`	Used to determine the toolset used to build the Razor assembly. Valid values are `Implicit`, `RazorSDK`, and `PrecompilationTool`.
EnableDefaultContentItems	Default is `true`. When `true`, includes *web.config*, *.json*, and *.cshtml* files as content in the project. When referenced via `Microsoft.NET.Sdk.Web`, files under *wwwroot* and config files are also included.
`EnableDefaultRazorGenerateItems`	When `true`, includes *.cshtml* files from `Content` items in `RazorGenerate` items.
`GenerateRazorTargetAssemblyInfo`	When `true`, generates a *.cs* file containing attributes specified by `RazorAssemblyAttribute` and includes the file in the compile output.
`EnableDefaultRazorTargetAssemblyInfoAttributes`	When `true`, adds a default set of assembly attributes to `RazorAssemblyAttribute`.
`CopyRazorGenerateFilesToPublishDirectory`	When `true`, copies `RazorGenerate` items (*.cshtml*) files to the publish directory. Typically, Razor files aren't required for a published app if they participate in compilation at build-time or publish-time. Defaults to `false`.
`PreserveCompilationReferences`	When `true`, copy reference assembly items to the publish directory. Typically, reference assemblies aren't required for a published app if Razor compilation occurs at build-time or publish-time. Set to `true` if your published app requires runtime compilation. For example, set the value to `true` if the app modifies *.cshtml* files at runtime or uses embedded views. Defaults to `false`.

PROPERTY	DESCRIPTION
`IncludeRazorContentInPack`	When `true`, all Razor content items (*.cshtml* files) are marked for inclusion in the generated NuGet package. Defaults to `false`.
`EmbedRazorGenerateSources`	When `true`, adds RazorGenerate (*.cshtml*) items as embedded files to the generated Razor assembly. Defaults to `false`.
`UseRazorBuildServer`	When `true`, uses a persistent build server process to offload code generation work. Defaults to the value of `UseSharedCompilation`.
`GenerateMvcApplicationPartsAssemblyAttributes`	When `true`, the SDK generates additional attributes used by MVC at runtime to perform application part discovery.
`DefaultWebContentItemExcludes`	A globbing pattern for item elements that are to be excluded from the `Content` item group in projects targeting the Web or Razor SDK
`ExcludeConfigFilesFromBuildOutput`	When `true`, *.config* and *.json* files do not get copied to the build output directory.
`AddRazorSupportForMvc`	When `true`, configures the Razor SDK to add support for the MVC configuration that is required when building applications containing MVC views or Razor Pages. This property is implicitly set for .NET Core 3.0 or later projects targeting the Web SDK
`RazorLangVersion`	The version of the Razor Language to target.

PROPERTY	DESCRIPTION
`RazorTargetName`	File name (without extension) of the assembly produced by Razor.
`RazorOutputPath`	The Razor output directory.
`RazorCompileToolset`	Used to determine the toolset used to build the Razor assembly. Valid values are `Implicit`, `RazorSDK`, and `PrecompilationTool`.
EnableDefaultContentItems	Default is `true`. When `true`, includes *web.config*, *.json*, and *.cshtml* files as content in the project. When referenced via `Microsoft.NET.Sdk.Web`, files under *wwwroot* and config files are also included.
`EnableDefaultRazorGenerateItems`	When `true`, includes *.cshtml* files from `Content` items in `RazorGenerate` items.
`GenerateRazorTargetAssemblyInfo`	When `true`, generates a *.cs* file containing attributes specified by `RazorAssemblyAttribute` and includes the file in the compile output.

PROPERTY	DESCRIPTION
`EnableDefaultRazorTargetAssemblyInfoAttributes`	When `true`, adds a default set of assembly attributes to `RazorAssemblyAttribute`.
`CopyRazorGenerateFilesToPublishDirectory`	When `true`, copies `RazorGenerate` items (*.cshtml*) files to the publish directory. Typically, Razor files aren't required for a published app if they participate in compilation at build-time or publish-time. Defaults to `false`.
`CopyRefAssembliesToPublishDirectory`	When `true`, copy reference assembly items to the publish directory. Typically, reference assemblies aren't required for a published app if Razor compilation occurs at build-time or publish-time. Set to `true` if your published app requires runtime compilation. For example, set the value to `true` if the app modifies *.cshtml* files at runtime or uses embedded views. Defaults to `false`.
`IncludeRazorContentInPack`	When `true`, all Razor content items (*.cshtml* files) are marked for inclusion in the generated NuGet package. Defaults to `false`.
`EmbedRazorGenerateSources`	When `true`, adds RazorGenerate (*.cshtml*) items as embedded files to the generated Razor assembly. Defaults to `false`.
`UseRazorBuildServer`	When `true`, uses a persistent build server process to offload code generation work. Defaults to the value of `UseSharedCompilation`.
`GenerateMvcApplicationPartsAssemblyAttributes`	When `true`, the SDK generates additional attributes used by MVC at runtime to perform application part discovery.
`DefaultWebContentItemExcludes`	A globbing pattern for item elements that are to be excluded from the `Content` item group in projects targeting the Web or Razor SDK
`ExcludeConfigFilesFromBuildOutput`	When `true`, *.config* and *.json* files do not get copied to the build output directory.
`AddRazorSupportForMvc`	When `true`, configures the Razor SDK to add support for the MVC configuration that is required when building applications containing MVC views or Razor Pages. This property is implicitly set for .NET Core 3.0 or later projects targeting the Web SDK
`RazorLangVersion`	The version of the Razor Language to target.

For more information on properties, see MSBuild properties.

Targets

The Razor SDK defines two primary targets:

- `RazorGenerate`: Code generates *.cs* files from `RazorGenerate` item elements. Use the `RazorGenerateDependsOn` property to specify additional targets that can run before or after this target.
- `RazorCompile`: Compiles generated *.cs* files in to a Razor assembly. Use the `RazorCompileDependsOn` to specify

additional targets that can run before or after this target.

- `RazorComponentGenerate`: Code generates *.cs* files for `RazorComponent` item elements. Use the `RazorComponentGenerateDependsOn` property to specify additional targets that can run before or after this target.

Runtime compilation of Razor views

- By default, the Razor SDK doesn't publish reference assemblies that are required to perform runtime compilation. This results in compilation failures when the application model relies on runtime compilation—for example, the app uses embedded views or changes views after the app is published. Set `CopyRefAssembliesToPublishDirectory` to `true` to continue publishing reference assemblies.
- For a web app, ensure your app is targeting the `Microsoft.NET.Sdk.Web` SDK.

Razor language version

When targeting the `Microsoft.NET.Sdk.Web` SDK, the Razor language version is inferred from the app's target framework version. For projects targeting the `Microsoft.NET.Sdk.Razor` SDK or in the rare case that the app requires a different Razor language version than the inferred value, a version can be configured by setting the `<RazorLangVersion>` property in the app's project file:

```
<PropertyGroup>
  <RazorLangVersion>{VERSION}</RazorLangVersion>
</PropertyGroup>
```

Razor's language version is tightly integrated with the version of the runtime that it was built for. Targeting a language version that isn't designed for the runtime is unsupported and likely produces build errors.

Additional resources

- Additions to the csproj format for .NET Core
- Common MSBuild project items

View components in ASP.NET Core

9/22/2020 • 11 minutes to read • Edit Online

By Rick Anderson

View or download sample code (how to download)

View components

View components are similar to partial views, but they're much more powerful. View components don't use model binding, and only depend on the data provided when calling into it. This article was written using controllers and views, but view components also work with Razor Pages.

A view component:

- Renders a chunk rather than a whole response.
- Includes the same separation-of-concerns and testability benefits found between a controller and view.
- Can have parameters and business logic.
- Is typically invoked from a layout page.

View components are intended anywhere you have reusable rendering logic that's too complex for a partial view, such as:

- Dynamic navigation menus
- Tag cloud (where it queries the database)
- Login panel
- Shopping cart
- Recently published articles
- Sidebar content on a typical blog
- A login panel that would be rendered on every page and show either the links to log out or log in, depending on the log in state of the user

A view component consists of two parts: the class (typically derived from ViewComponent) and the result it returns (typically a view). Like controllers, a view component can be a POCO, but most developers will want to take advantage of the methods and properties available by deriving from `ViewComponent`.

When considering if view components meet an app's specifications, consider using Razor Components instead. Razor Components also combine markup with C# code to produce reusable UI units. Razor Components are designed for developer productivity when providing client-side UI logic and composition. For more information, see Create and use ASP.NET Core Razor components.

Creating a view component

This section contains the high-level requirements to create a view component. Later in the article, we'll examine each step in detail and create a view component.

The view component class

A view component class can be created by any of the following:

- Deriving from *ViewComponent*
- Decorating a class with the `[ViewComponent]` attribute, or deriving from a class with the `[ViewComponent]`

attribute

- Creating a class where the name ends with the suffix *ViewComponent*

Like controllers, view components must be public, non-nested, and non-abstract classes. The view component name is the class name with the "ViewComponent" suffix removed. It can also be explicitly specified using the `ViewComponentAttribute.Name` property.

A view component class:

- Fully supports constructor dependency injection
- Doesn't take part in the controller lifecycle, which means you can't use filters in a view component

View component methods

A view component defines its logic in an `InvokeAsync` method that returns a `Task<IViewComponentResult>` or in a synchronous `Invoke` method that returns an `IViewComponentResult`. Parameters come directly from invocation of the view component, not from model binding. A view component never directly handles a request. Typically, a view component initializes a model and passes it to a view by calling the `View` method. In summary, view component methods:

- Define an `InvokeAsync` method that returns a `Task<IViewComponentResult>` or a synchronous `Invoke` method that returns an `IViewComponentResult`.
- Typically initializes a model and passes it to a view by calling the `ViewComponent` `View` method.
- Parameters come from the calling method, not HTTP. There's no model binding.
- Are not reachable directly as an HTTP endpoint. They're invoked from your code (usually in a view). A view component never handles a request.
- Are overloaded on the signature rather than any details from the current HTTP request.

View search path

The runtime searches for the view in the following paths:

- /Views/{Controller Name}/Components/{View Component Name}/{View Name}
- /Views/Shared/Components/{View Component Name}/{View Name}
- /Pages/Shared/Components/{View Component Name}/{View Name}

The search path applies to projects using controllers + views and Razor Pages.

The default view name for a view component is *Default*, which means your view file will typically be named *Default.cshtml*. You can specify a different view name when creating the view component result or when calling the `View` method.

We recommend you name the view file *Default.cshtml* and use the *Views/Shared/Components/{View Component Name}/{View Name}* path. The `PriorityList` view component used in this sample uses *Views/Shared/Components/PriorityList/Default.cshtml* for the view component view.

Customize the view search path

To customize the view search path, modify Razor's ViewLocationFormats collection. For example, to search for views within the path "/Components/{View Component Name}/{View Name}", add a new item to the collection:

```
services.AddMvc()
    .AddRazorOptions(options =>
    {
        options.ViewLocationFormats.Add("/{0}.cshtml");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

In the preceding code, the placeholder "{0}" represents the path "Components/{View Component Name}/{View Name}".

Invoking a view component

To use the view component, call the following inside a view:

```
@await Component.InvokeAsync("Name of view component", {Anonymous Type Containing Parameters})
```

The parameters will be passed to the `InvokeAsync` method. The `PriorityList` view component developed in the article is invoked from the *Views/ToDo/Index.cshtml* view file. In the following, the `InvokeAsync` method is called with two parameters:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

Invoking a view component as a Tag Helper

For ASP.NET Core 1.1 and higher, you can invoke a view component as a Tag Helper:

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

Pascal-cased class and method parameters for Tag Helpers are translated into their kebab case. The Tag Helper to invoke a view component uses the `<vc></vc>` element. The view component is specified as follows:

```
<vc:[view-component-name]
  parameter1="parameter1 value"
  parameter2="parameter2 value">
</vc:[view-component-name]>
```

To use a view component as a Tag Helper, register the assembly containing the view component using the `@addTagHelper` directive. If your view component is in an assembly called `MyWebApp`, add the following directive to the *_ViewImports.cshtml* file:

```
@addTagHelper *, MyWebApp
```

You can register a view component as a Tag Helper to any file that references the view component. See Managing Tag Helper Scope for more information on how to register Tag Helpers.

The `InvokeAsync` method used in this tutorial:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

In Tag Helper markup:

```
<vc:priority-list max-priority="2" is-done="false">
</vc:priority-list>
```

In the sample above, the `PriorityList` view component becomes `priority-list`. The parameters to the view component are passed as attributes in kebab case.

Invoking a view component directly from a controller

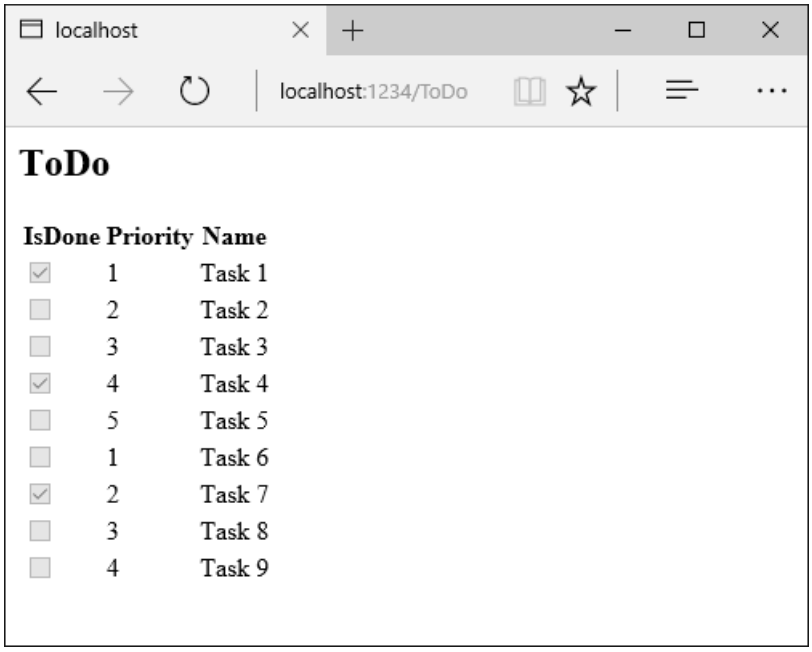
View components are typically invoked from a view, but you can invoke them directly from a controller method. While view components don't define endpoints like controllers, you can easily implement a controller action that returns the content of a `ViewComponentResult`.

In this example, the view component is called directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```

Walkthrough: Creating a simple view component

Download, build and test the starter code. It's a simple project with a `ToDo` controller that displays a list of *ToDo* items.



Add a ViewComponent class

Create a *ViewComponents* folder and add the following `PriorityListViewComponent` class:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityListViewComponent : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityListViewComponent(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }
        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                                 x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

Notes on the code:

- View component classes can be contained in **any** folder in the project.
- Because the class name PriorityList**ViewComponent** ends with the suffix **ViewComponent**, the runtime will use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.
- The `[ViewComponent]` attribute can change the name used to reference a view component. For example, we could've named the class `XYZ` and applied the `ViewComponent` attribute:

  ```
  [ViewComponent(Name = "PriorityList")]
     public class XYZ : ViewComponent
  ```

- The `[ViewComponent]` attribute above tells the view component selector to use the name `PriorityList` when looking for the views associated with the component, and to use the string "PriorityList" when referencing the class component from a view. I'll explain that in more detail later.
- The component uses dependency injection to make the data context available.
- `InvokeAsync` exposes a method which can be called from a view, and it can take an arbitrary number of arguments.
- The `InvokeAsync` method returns the set of `ToDo` items that satisfy the `isDone` and `maxPriority` parameters.

Create the view component Razor view

- Create the *Views/Shared/Components* folder. This folder **must** be named *Components*.

- Create the *Views/Shared/Components/PriorityList* folder. This folder name must match the name of the view component class, or the name of the class minus the suffix (if we followed convention and used the *ViewComponent* suffix in the class name). If you used the `ViewComponent` attribute, the class name would need to match the attribute designation.

- Create a *Views/Shared/Components/PriorityList/Default.cshtml* Razor view:

  ```
  @model IEnumerable<ViewComponentSample.Models.TodoItem>

  <h3>Priority Items</h3>
  <ul>
      @foreach (var todo in Model)
      {
          <li>@todo.Name</li>
      }
  </ul>
  ```

  The Razor view takes a list of `TodoItem` and displays them. If the view component `InvokeAsync` method doesn't pass the name of the view (as in our sample), *Default* is used for the view name by convention. Later in the tutorial, I'll show you how to pass the name of the view. To override the default styling for a specific controller, add a view to the controller-specific view folder (for example *Views/ToDo/Components/PriorityList/Default.cshtml)*.

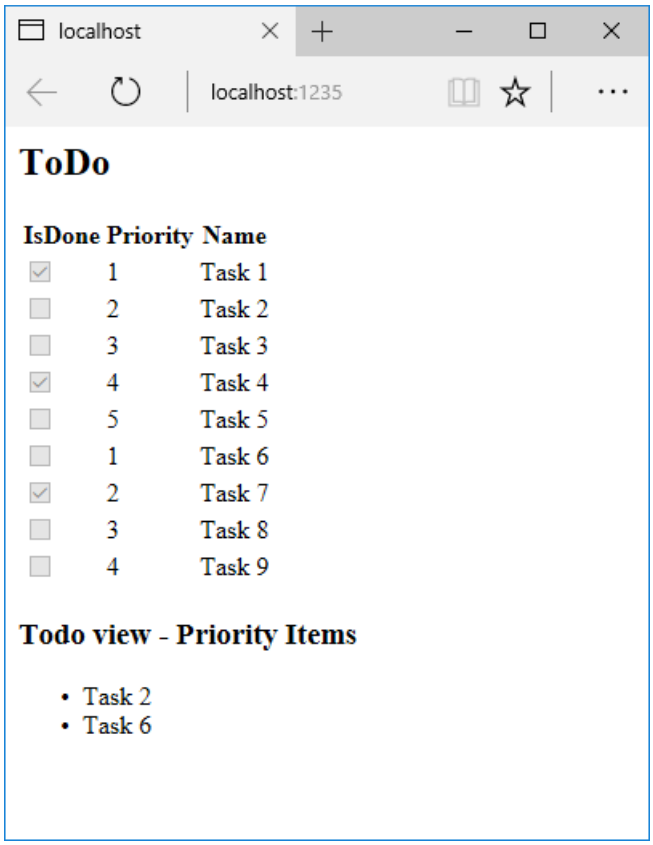
  If the view component is controller-specific, you can add it to the controller-specific folder (*Views/ToDo/Components/PriorityList/Default.cshtml*).

- Add a `div` containing a call to the priority list component to the bottom of the *Views/ToDo/index.cshtml* file:

  ```
  </table>
  <div>
      @await Component.InvokeAsync("PriorityList", new { maxPriority = 2, isDone = false })
  </div>
  ```

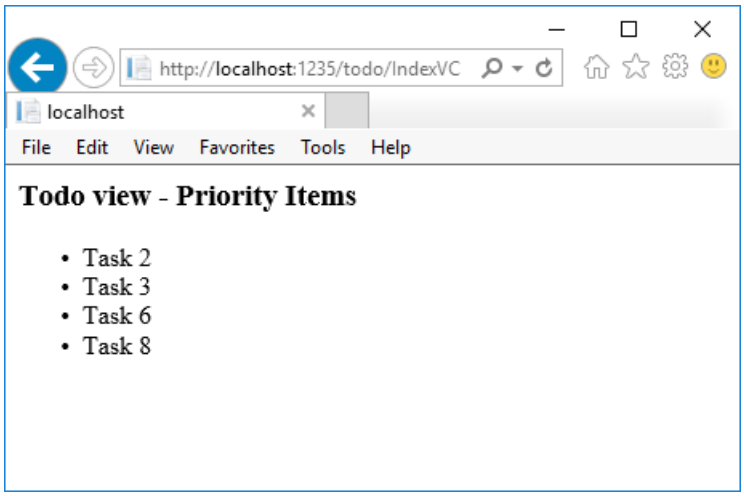
The markup `@await Component.InvokeAsync` shows the syntax for calling view components. The first argument is the name of the component we want to invoke or call. Subsequent parameters are passed to the component. `InvokeAsync` can take an arbitrary number of arguments.

Test the app. The following image shows the ToDo list and the priority items:



You can also call the view component directly from the controller:

```
public IActionResult IndexVC()
{
    return ViewComponent("PriorityList", new { maxPriority = 3, isDone = false });
}
```



Specifying a view name

A complex view component might need to specify a non-default view under some conditions. The following code shows how to specify the "PVC" view from the `InvokeAsync` method. Update the `InvokeAsync` method in the `PriorityListViewComponent` class.

```
public async Task<IViewComponentResult> InvokeAsync(
    int maxPriority, bool isDone)
{
    string MyView = "Default";
    // If asking for all completed tasks, render with the "PVC" view.
    if (maxPriority > 3 && isDone == true)
    {
        MyView = "PVC";
    }
    var items = await GetItemsAsync(maxPriority, isDone);
    return View(MyView, items);
}
```

Copy the *Views/Shared/Components/PriorityList/Default.cshtml* file to a view named *Views/Shared/Components/PriorityList/PVC.cshtml*. Add a heading to indicate the PVC view is being used.

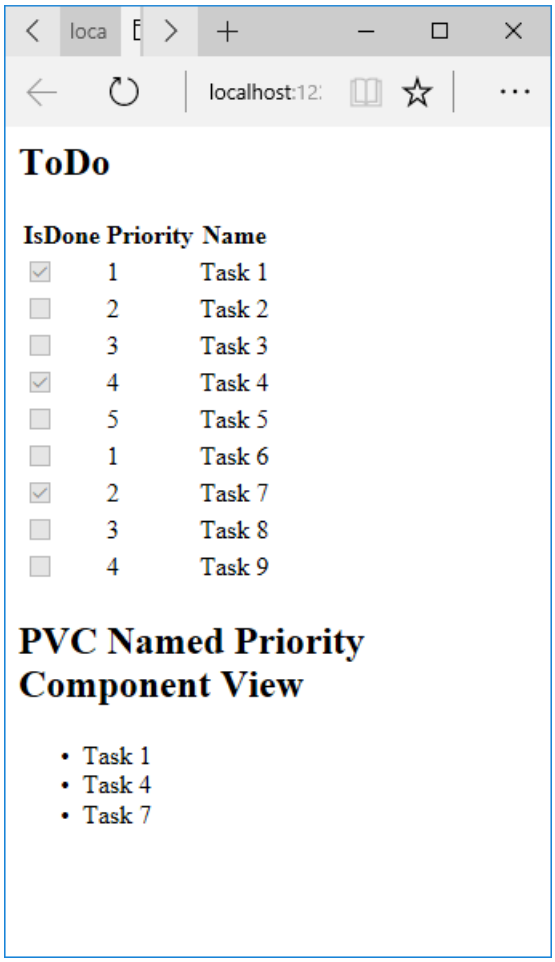
```
@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h2> PVC Named Priority Component View</h2>
<h4>@ViewBag.PriorityMessage</h4>
<ul>
    @foreach (var todo in Model)
    {
        <li>@todo.Name</li>
    }
</ul>
```

Update *Views/ToDo/Index.cshtml*:

```
@await Component.InvokeAsync("PriorityList", new { maxPriority = 4, isDone = true })
```

Run the app and verify PVC view.



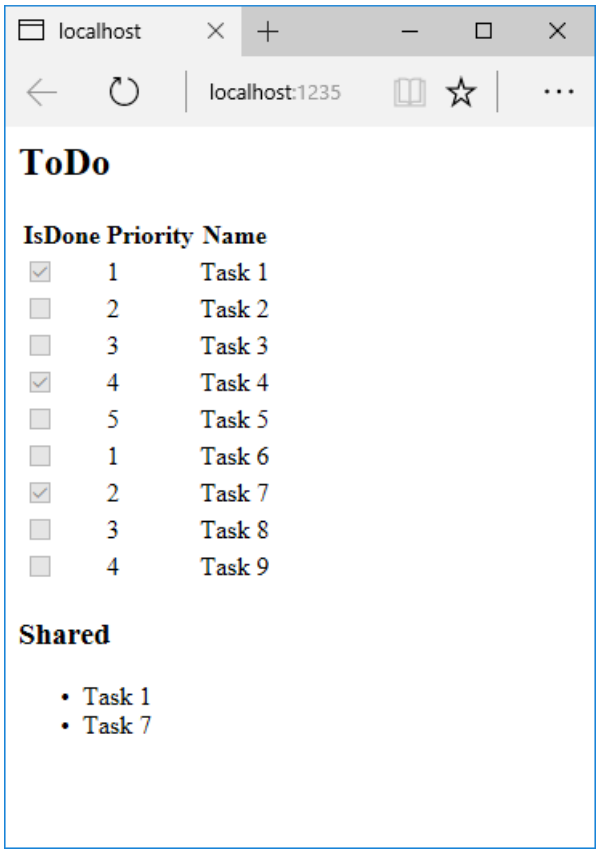
If the PVC view isn't rendered, verify you are calling the view component with a priority of 4 or higher.

Examine the view path

- Change the priority parameter to three or less so the priority view isn't returned.
- Temporarily rename the *Views/ToDo/Components/PriorityList/Default.cshtml* to *1Default.cshtml*.
- Test the app, you'll get the following error:

```
An unhandled exception occurred while processing the request.
InvalidOperationException: The view 'Components/PriorityList/Default' wasn't found. The following
locations were searched:
/Views/ToDo/Components/PriorityList/Default.cshtml
/Views/Shared/Components/PriorityList/Default.cshtml
EnsureSuccessful
```

- Copy *Views/ToDo/Components/PriorityList/1Default.cshtml* to *Views/Shared/Components/PriorityList/Default.cshtml*.
- Add some markup to the *Shared* ToDo view component view to indicate the view is from the *Shared* folder.
- Test the **Shared** component view.



Avoiding hard-coded strings

If you want compile time safety, you can replace the hard-coded view component name with the class name. Create the view component without the "ViewComponent" suffix:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityList : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityList(ToDoContext context)
        {
            db = context;
        }

        public async Task<IViewComponentResult> InvokeAsync(
        int maxPriority, bool isDone)
        {
            var items = await GetItemsAsync(maxPriority, isDone);
            return View(items);
        }
        private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
        {
            return db.ToDo.Where(x => x.IsDone == isDone &&
                                 x.Priority <= maxPriority).ToListAsync();
        }
    }
}
```

Add a `using` statement to your Razor view file, and use the `nameof` operator:

```
@using ViewComponentSample.Models
@using ViewComponentSample.ViewComponents
@model IEnumerable<TodoItem>

    <h2>ToDo nameof</h2>
    <!-- Markup removed for brevity.  -->

    <div>

        @*
            Note:
            To use the below line, you need to #define no_suffix in ViewComponents/PriorityList.cs or it
won't compile.
            By doing so it will cause a problem to index as there will be multiple viewcomponents
            with the same name after the compiler removes the suffix "ViewComponent"
        *@

        @*@await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })*@
    </div>
```

Perform synchronous work

The framework handles invoking a synchronous `Invoke` method if you don't need to perform asynchronous work. The following method creates a synchronous `Invoke` view component:

```
public class PriorityList : ViewComponent
{
    public IViewComponentResult Invoke(int maxPriority, bool isDone)
    {
        var items = new List<string> { $"maxPriority: {maxPriority}", $"isDone: {isDone}" };
        return View(items);
    }
}
```

The view component's Razor file lists the strings passed to the `Invoke` method (*Views/Home/Components/PriorityList/Default.cshtml*):

```
@model List<string>

<h3>Priority Items</h3>
<ul>
    @foreach (var item in Model)
    {
        <li>@item</li>
    }
</ul>
```

The view component is invoked in a Razor file (for example, *Views/Home/Index.cshtml*) using one of the following approaches:

- IViewComponentHelper
- Tag Helper

To use the IViewComponentHelper approach, call `Component.InvokeAsync`:

The view component is invoked in a Razor file (for example, *Views/Home/Index.cshtml*) with IViewComponentHelper.

Call `Component.InvokeAsync`:

```
@await Component.InvokeAsync(nameof(PriorityList), new { maxPriority = 4, isDone = true })
```

To use the Tag Helper, register the assembly containing the View Component using the `@addTagHelper` directive (the view component is in an assembly called `MyWebApp`):

```
@addTagHelper *, MyWebApp
```

Use the view component Tag Helper in the Razor markup file:

```
<vc:priority-list max-priority="999" is-done="false">
</vc:priority-list>
```

The method signature of `PriorityList.Invoke` is synchronous, but Razor finds and calls the method with `Component.InvokeAsync` in the markup file.

All view component parameters are required

Each parameter in a view component is a required attribute. See this GitHub issue. If any parameter is omitted:

- The `InvokeAsync` method signature won't match, therefore the method won't execute.
- The ViewComponent won't render any markup.
- No errors will be thrown.

Additional resources

- Dependency injection into views

Razor file compilation in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

By Rick Anderson

Razor files with a *.cshtml* extension are compiled at both build and publish time using the Razor SDK. Runtime compilation may be optionally enabled by configuring your project.

Razor compilation

Build-time and publish-time compilation of Razor files is enabled by default by the Razor SDK. When enabled, runtime compilation complements build-time compilation, allowing Razor files to be updated if they're edited.

Enable runtime compilation at project creation

The Razor Pages and MVC project templates include an option to enable runtime compilation when the project is created. This option is supported in ASP.NET Core 3.1 and later.

- Visual Studio
- .NET Core CLI

In the **Create a new ASP.NET Core web application** dialog:

1. Select either the **Web Application** or the **Web Application (Model-View-Controller)** project template.
2. Select the **Enable Razor runtime compilation** check box.

Enable runtime compilation in an existing project

To enable runtime compilation for all environments in an existing project:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.
2. Update the project's `Startup.ConfigureServices` method to include a call to AddRazorRuntimeCompilation. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorRuntimeCompilation();

    // code omitted for brevity
}
```

Conditionally enable runtime compilation in an existing project

Runtime compilation can be enabled such that it's only available for local development. Conditionally enabling in this manner ensures that the published output:

- Uses compiled views.
- Doesn't enable file watchers in production.

To enable runtime compilation only in the Development environment:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.
2. Modify the launch profile `environmentVariables` section in *launchSettings.json*:
   - Verify `ASPNETCORE_ENVIRONMENT` is set to `"Development"`.
   - Set `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` to `"Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"`.

In the following example, runtime compilation is enabled in the Development environment for the `IIS Express` and `RazorPagesApp` launch profiles:

```
{
    "iisSettings": {
      "windowsAuthentication": false,
      "anonymousAuthentication": true,
      "iisExpress": {
        "applicationUrl": "http://localhost:57676",
        "sslPort": 44364
      }
    },
    "profiles": {
      "IIS Express": {
        "commandName": "IISExpress",
        "launchBrowser": true,
        "environmentVariables": {
          "ASPNETCORE_ENVIRONMENT": "Development",
          "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES": "Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"
        }
      },
      "RazorPagesApp": {
        "commandName": "Project",
        "launchBrowser": true,
        "applicationUrl": "https://localhost:5001;http://localhost:5000",
        "environmentVariables": {
          "ASPNETCORE_ENVIRONMENT": "Development",
          "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES": "Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"
        }
      }
    }
  }
```

No code changes are needed in the project's `Startup` class. At runtime, ASP.NET Core searches for an assembly-level HostingStartup attribute in `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`. The `HostingStartup` attribute specifies the app startup code to execute. That startup code enables runtime compilation.

Enable runtime compilation for a Razor Class Library

Consider a scenario in which a Razor Pages project references a Razor Class Library (RCL) named *MyClassLib*. The RCL contains a *_Layout.cshtml* file that all of your team's MVC and Razor Pages projects consume. You want to enable runtime compilation for the *_Layout.cshtml* file in that RCL. Make the following changes in the Razor Pages project:

1. Enable runtime compilation with the instructions at Conditionally enable runtime compilation in an existing project.
2. Configure the runtime compilation options in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.Configure<MvcRazorRuntimeCompilationOptions>(options =>
    {
        var libraryPath = Path.GetFullPath(
            Path.Combine(HostEnvironment.ContentRootPath, "..", "MyClassLib"));
        options.FileProviders.Add(new PhysicalFileProvider(libraryPath));
    });
}
```

In the preceding code, an absolute path to the *MyClassLib* RCL is constructed. The PhysicalFileProvider API is used to locate directories and files at that absolute path. Finally, the `PhysicalFileProvider` instance is added to a file providers collection, which allows access to the RCL's *.cshtml* files.

Additional resources

- RazorCompileOnBuild and RazorCompileOnPublish properties.
- Introduction to Razor Pages in ASP.NET Core
- Views in ASP.NET Core MVC
- ASP.NET Core Razor SDK

Razor files with a *.cshtml* extension are compiled at both build and publish time using the Razor SDK. Runtime compilation may be optionally enabled by configuring your application.

Razor compilation

Build-time and publish-time compilation of Razor files is enabled by default by the Razor SDK. When enabled, runtime compilation complements build-time compilation, allowing Razor files to be updated if they're edited.

Runtime compilation

To enable runtime compilation for all environments and configuration modes:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.
2. Update the project's `Startup.ConfigureServices` method to include a call to AddRazorRuntimeCompilation. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorRuntimeCompilation();

    // code omitted for brevity
}
```

Conditionally enable runtime compilation

Runtime compilation can be enabled such that it's only available for local development. Conditionally enabling in this manner ensures that the published output:

- Uses compiled views.
- Is smaller in size.
- Doesn't enable file watchers in production.

To enable runtime compilation based on the environment and configuration mode:

1. Conditionally reference the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation package based on the active `Configuration` value:

   ```
   <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.1.0"
   Condition="'$(Configuration)' == 'Debug'" />
   ```

2. Update the project's `Startup.ConfigureServices` method to include a call to `AddRazorRuntimeCompilation`. Conditionally execute `AddRazorRuntimeCompilation` such that it only runs in Debug mode when the `ASPNETCORE_ENVIRONMENT` variable is set to `Development`:

   ```
   public class Startup
   {
       public Startup(IConfiguration configuration, IWebHostEnvironment env)
       {
           Configuration = configuration;
           Env = env;
       }

       public IWebHostEnvironment Env { get; set; }
       public IConfiguration Configuration { get; }

       public void ConfigureServices(IServiceCollection services)
       {
           IMvcBuilder builder = services.AddRazorPages();

   #if DEBUG
               if (Env.IsDevelopment())
               {
                   builder.AddRazorRuntimeCompilation();
               }
   #endif
       }

       public void Configure(IApplicationBuilder app)
       {
           if (Env.IsDevelopment())
           {
               app.UseDeveloperExceptionPage();
           }
           else
           {
               app.UseExceptionHandler("/Error");
               app.UseHsts();
           }

           app.UseHttpsRedirection();
           app.UseStaticFiles();

           app.UseRouting();

           app.UseAuthorization();

           app.UseEndpoints(endpoints =>
           {
               endpoints.MapRazorPages();
           });
       }
   }
   ```

Additional resources

- RazorCompileOnBuild and RazorCompileOnPublish properties.
- Introduction to Razor Pages in ASP.NET Core
- Views in ASP.NET Core MVC
- ASP.NET Core Razor SDK
- See the runtime compilation sample on GitHub for a sample that shows making runtime compilation work across projects.

A Razor file is compiled at runtime, when the associated Razor Page or MVC view is invoked. Razor files are compiled at both build and publish time using the Razor SDK.

Razor compilation

Build- and publish-time compilation of Razor files is enabled by default by the Razor SDK. Editing Razor files after they're updated is supported at build time. By default, only the compiled *Views.dll* and no *.cshtml* files or references assemblies required to compile Razor files are deployed with your app.

> **IMPORTANT**
>
> The precompilation tool has been deprecated, and will be removed in ASP.NET Core 3.0. We recommend migrating to Razor Sdk.
>
> The Razor SDK is effective only when no precompilation-specific properties are set in the project file. For instance, setting the *.csproj* file's `MvcRazorCompileOnPublish` property to `true` disables the Razor SDK.

Runtime compilation

Build-time compilation is supplemented by runtime compilation of Razor files. ASP.NET Core MVC will recompile Razor files when the contents of a *.cshtml* file change.

Additional resources

- Introduction to Razor Pages in ASP.NET Core
- Views in ASP.NET Core MVC
- ASP.NET Core Razor SDK

Upload files in ASP.NET Core

9/22/2020 • 66 minutes to read • Edit Online

By Steve Smith and Rutger Storm

ASP.NET Core supports uploading one or more files using buffered model binding for smaller files and unbuffered streaming for larger files.

View or download sample code (how to download)

Security considerations

Use caution when providing users with the ability to upload files to a server. Attackers may attempt to:

- Execute denial of service attacks.
- Upload viruses or malware.
- Compromise networks and servers in other ways.

Security steps that reduce the likelihood of a successful attack are:

- Upload files to a dedicated file upload area, preferably to a non-system drive. A dedicated location makes it easier to impose security restrictions on uploaded files. Disable execute permissions on the file upload location.†
- Do **not** persist uploaded files in the same directory tree as the app.†
- Use a safe file name determined by the app. Don't use a file name provided by the user or the untrusted file name of the uploaded file.† HTML encode the untrusted file name when displaying it. For example, logging the file name or displaying in UI (Razor automatically HTML encodes output).
- Allow only approved file extensions for the app's design specification.†
- Verify that client-side checks are performed on the server.† Client-side checks are easy to circumvent.
- Check the size of an uploaded file. Set a maximum size limit to prevent large uploads.†
- When files shouldn't be overwritten by an uploaded file with the same name, check the file name against the database or physical storage before uploading the file.
- **Run a virus/malware scanner on uploaded content before the file is stored.**

†The sample app demonstrates an approach that meets the criteria.

> **WARNING**
>
> Uploading malicious code to a system is frequently the first step to executing code that can:
>
> - Completely gain control of a system.
> - Overload a system with the result that the system crashes.
> - Compromise user or system data.
> - Apply graffiti to a public UI.
>
> For information on reducing the attack surface area when accepting files from users, see the following resources:
>
> - Unrestricted File Upload
> - Azure Security: Ensure appropriate controls are in place when accepting files from users

For more information on implementing security measures, including examples from the sample app, see the Validation section.

Storage scenarios

Common storage options for files include:

- Database
  - For small file uploads, a database is often faster than physical storage (file system or network share) options.
  - A database is often more convenient than physical storage options because retrieval of a database record for user data can concurrently supply the file content (for example, an avatar image).
  - A database is potentially less expensive than using a data storage service.
- Physical storage (file system or network share)
  - For large file uploads:
    - Database limits may restrict the size of the upload.
    - Physical storage is often less economical than storage in a database.
  - Physical storage is potentially less expensive than using a data storage service.
  - The app's process must have read and write permissions to the storage location. **Never grant execute permission.**
- Data storage service (for example, Azure Blob Storage)
  - Services usually offer improved scalability and resiliency over on-premises solutions that are usually subject to single points of failure.
  - Services are potentially lower cost in large storage infrastructure scenarios.

  For more information, see Quickstart: Use .NET to create a blob in object storage.

File upload scenarios

Two general approaches for uploading files are buffering and streaming.

Buffering

The entire file is read into an IFormFile, which is a C# representation of the file used to process or save the file.

The resources (disk, memory) used by file uploads depend on the number and size of concurrent file uploads. If an app attempts to buffer too many uploads, the site crashes when it runs out of memory or disk space. If the size or frequency of file uploads is exhausting app resources, use streaming.

> **NOTE**
> Any single buffered file exceeding 64 KB is moved from memory to a temp file on disk.

Buffering small files is covered in the following sections of this topic:

- Physical storage
- Database

Streaming

The file is received from a multipart request and directly processed or saved by the app. Streaming doesn't improve performance significantly. Streaming reduces the demands for memory or disk space when uploading files.

Streaming large files is covered in the Upload large files with streaming section.

Upload small files with buffered model binding to physical storage

To upload small files, use a multipart form or construct a POST request using JavaScript.

The following example demonstrates the use of a Razor Pages form to upload a single file (*Pages/BufferedSingleFileUploadPhysical.cshtml* in the sample app):

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file">
            <span asp-validation-for="FileUpload.FormFile"></span>
        </dd>
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit" value="Upload" />
</form>
```

The following example is analogous to the prior example except that:

- JavaScript's (Fetch API) is used to submit the form's data.
- There's no validation.

```
<form action="BufferedSingleFileUploadPhysical/?handler=Upload"
      enctype="multipart/form-data" onsubmit="AJAXSubmit(this);return false;"
      method="post">
    <dl>
        <dt>
            <label for="FileUpload_FormFile">File</label>
        </dt>
        <dd>
            <input id="FileUpload_FormFile" type="file"
                name="FileUpload.FormFile" />
        </dd>
    </dl>

    <input class="btn" type="submit" value="Upload" />

    <div style="margin-top:15px">
        <output name="result"></output>
    </div>
</form>

<script>
  async function AJAXSubmit (oFormElement) {
    var resultElement = oFormElement.elements.namedItem("result");
    const formData = new FormData(oFormElement);

    try {
    const response = await fetch(oFormElement.action, {
      method: 'POST',
      body: formData
    });

    if (response.ok) {
      window.location.href = '/';
    }

    resultElement.value = 'Result: ' + response.status + ' ' +
      response.statusText;
    } catch (error) {
      console.error('Error:', error);
    }
  }
</script>
```

To perform the form POST in JavaScript for clients that don't support the Fetch API, use one of the following approaches:

- Use a Fetch Polyfill (for example, window.fetch polyfill (github/fetch)).
- Use `XMLHttpRequest`. For example:

```
<script>
  "use strict";

  function AJAXSubmit (oFormElement) {
    var oReq = new XMLHttpRequest();
    oReq.onload = function(e) {
    oFormElement.elements.namedItem("result").value =
      'Result: ' + this.status + ' ' + this.statusText;
    };
    oReq.open("post", oFormElement.action);
    oReq.send(new FormData(oFormElement));
  }
</script>
```

In order to support file uploads, HTML forms must specify an encoding type (`enctype`) of `multipart/form-data`.

For a `files` input element to support uploading multiple files provide the `multiple` attribute on the `<input>` element:

```
<input asp-for="FileUpload.FormFiles" type="file" multiple>
```

The individual files uploaded to the server can be accessed through Model Binding using IFormFile. The sample app demonstrates multiple buffered file uploads for database and physical storage scenarios.

WARNING

Do **not** use the `FileName` property of IFormFile other than for display and logging. When displaying or logging, HTML encode the file name. An attacker can provide a malicious filename, including full paths or relative paths. Applications should:

- Remove the path from the user-supplied filename.
- Save the HTML-encoded, path-removed filename for UI or logging.
- Generate a new random filename for storage.

The following code removes the path from the file name:

```
string untrustedFileName = Path.GetFileName(pathName);
```

The examples provided thus far don't take into account security considerations. Additional information is provided by the following sections and the sample app:

- Security considerations
- Validation

When uploading files using model binding and IFormFile, the action method can accept:

- A single IFormFile.
- Any of the following collections that represent several files:
  - IFormFileCollection
  - IEnumerable<IFormFile>
  - List<IFormFile>

> **NOTE**
>
> Binding matches form files by name. For example, the HTML `name` value in `<input type="file" name="formFile">` must match the C# parameter/property bound (`FormFile`). For more information, see the Match name attribute value to parameter name of POST method section.

The following example:

- Loops through one or more uploaded files.
- Uses Path.GetTempFileName to return a full path for a file, including the file name.
- Saves the files to the local file system using a file name generated by the app.
- Returns the total number and size of files uploaded.

```
public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            var filePath = Path.GetTempFileName();

            using (var stream = System.IO.File.Create(filePath))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // Process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size });
}
```

Use `Path.GetRandomFileName` to generate a file name without a path. In the following example, the path is obtained from configuration:

```
foreach (var formFile in files)
{
    if (formFile.Length > 0)
    {
        var filePath = Path.Combine(_config["StoredFilesPath"],
            Path.GetRandomFileName());

        using (var stream = System.IO.File.Create(filePath))
        {
            await formFile.CopyToAsync(stream);
        }
    }
}
```

The path passed to the FileStream *must* include the file name. If the file name isn't provided, an UnauthorizedAccessException is thrown at runtime.

Files uploaded using the IFormFile technique are buffered in memory or on disk on the server before processing. Inside the action method, the IFormFile contents are accessible as a Stream. In addition to the local file system, files can be saved to a network share or to a file storage service, such as Azure Blob storage.

For another example that loops over multiple files for upload and uses safe file names, see *Pages/BufferedMultipleFileUploadPhysical.cshtml.cs* in the sample app.

> **WARNING**
>
> Path.GetTempFileName throws an IOException if more than 65,535 files are created without deleting previous temporary files. The limit of 65,535 files is a per-server limit. For more information on this limit on Windows OS, see the remarks in the following topics:
>
> - GetTempFileNameA function
> - GetTempFileName

Upload small files with buffered model binding to a database

To store binary file data in a database using Entity Framework, define a Byte array property on the entity:

```
public class AppFile
{
    public int Id { get; set; }
    public byte[] Content { get; set; }
}
```

Specify a page model property for the class that includes an IFormFile:

```
public class BufferedSingleFileUploadDbModel : PageModel
{
    ...

    [BindProperty]
    public BufferedSingleFileUploadDb FileUpload { get; set; }

    ...
}

public class BufferedSingleFileUploadDb
{
    [Required]
    [Display(Name="File")]
    public IFormFile FormFile { get; set; }
}
```

> **NOTE**
>
> IFormFile can be used directly as an action method parameter or as a bound model property. The prior example uses a bound model property.

The `FileUpload` is used in the Razor Pages form:

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file">
        </dd>
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit" value="Upload">
</form>
```

When the form is POSTed to the server, copy the IFormFile to a stream and save it as a byte array in the database. In the following example, `_dbContext` stores the app's database context:

```
public async Task<IActionResult> OnPostUploadAsync()
{
    using (var memoryStream = new MemoryStream())
    {
        await FileUpload.FormFile.CopyToAsync(memoryStream);

        // Upload the file if less than 2 MB
        if (memoryStream.Length < 2097152)
        {
            var file = new AppFile()
            {
                Content = memoryStream.ToArray()
            };

            _dbContext.File.Add(file);

            await _dbContext.SaveChangesAsync();
        }
        else
        {
            ModelState.AddModelError("File", "The file is too large.");
        }
    }

    return Page();
}
```

The preceding example is similar to a scenario demonstrated in the sample app:

- *Pages/BufferedSingleFileUploadDb.cshtml*
- *Pages/BufferedSingleFileUploadDb.cshtml.cs*

> **WARNING**
>
> Use caution when storing binary data in relational databases, as it can adversely impact performance.
>
> Don't rely on or trust the `FileName` property of IFormFile without validation. The `FileName` property should only be used for display purposes and only after HTML encoding.
>
> The examples provided don't take into account security considerations. Additional information is provided by the following sections and the sample app:
>
> - Security considerations
> - Validation

Upload large files with streaming

The following example demonstrates how to use JavaScript to stream a file to a controller action. The file's antiforgery token is generated using a custom filter attribute and passed to the client HTTP headers instead of in the request body. Because the action method processes the uploaded data directly, form model binding is disabled by another custom filter. Within the action, the form's contents are read using a `MultipartReader`, which reads each individual `MultipartSection`, processing the file or storing the contents as appropriate. After the multipart sections are read, the action performs its own model binding.

The initial page response loads the form and saves an antiforgery token in a cookie (via the `GenerateAntiforgeryTokenCookieAttribute` attribute). The attribute uses ASP.NET Core's built-in antiforgery support to set a cookie with a request token:

```
public class GenerateAntiforgeryTokenCookieAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        var antiforgery = context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // Send the request token as a JavaScript-readable cookie
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);

        context.HttpContext.Response.Cookies.Append(
            "RequestVerificationToken",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }

    public override void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

The `DisableFormValueModelBindingAttribute` is used to disable model binding:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<FormFileValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

In the sample app, `GenerateAntiforgeryTokenCookieAttribute` and `DisableFormValueModelBindingAttribute` are applied as filters to the page application models of `/StreamedSingleFileUploadDb` and `/StreamedSingleFileUploadPhysical` in `Startup.ConfigureServices` using Razor Pages conventions:

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/StreamedSingleFileUploadDb",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
    options.Conventions
        .AddPageApplicationModelConvention("/StreamedSingleFileUploadPhysical",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
});
```

Since model binding doesn't read the form, parameters that are bound from the form don't bind (query, route, and header continue to work). The action method works directly with the `Request` property. A `MultipartReader` is used to read each section. Key/value data is stored in a `KeyValueAccumulator`. After the multipart sections are read, the contents of the `KeyValueAccumulator` are used to bind the form data to a model type.

The complete `StreamingController.UploadDatabase` method for streaming to a database with EF Core:

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadDatabase()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
        // Log error

        return BadRequest(ModelState);
    }

    // Accumulate the form data key-value pairs in the request (formAccumulator).
    var formAccumulator = new KeyValueAccumulator();
    var trustedFileNameForDisplay = string.Empty;
    var untrustedFileNameForStorage = string.Empty;
    var streamedFileContent = new byte[0];

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();

    while (section != null)
    {
        var hasContentDispositionHeader =
            ContentDispositionHeaderValue.TryParse(
                section.ContentDisposition, out var contentDisposition);

        if (hasContentDispositionHeader)
        {
            if (MultipartRequestHelper
```

```
            .HasFileContentDisposition(contentDisposition))
        {
            untrustedFileNameForStorage = contentDisposition.FileName.Value;
            // Don't trust the file name sent by the client. To display
            // the file name, HTML-encode the value.
            trustedFileNameForDisplay = WebUtility.HtmlEncode(
                    contentDisposition.FileName.Value);

            streamedFileContent =
                await FileHelpers.ProcessStreamedFile(section, contentDisposition,
                    ModelState, _permittedExtensions, _fileSizeLimit);

            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
        }
        else if (MultipartRequestHelper
            .HasFormDataContentDisposition(contentDisposition))
        {
            // Don't limit the key name length because the
            // multipart headers length limit is already in effect.
            var key = HeaderUtilities
                .RemoveQuotes(contentDisposition.Name).Value;
            var encoding = GetEncoding(section);

            if (encoding == null)
            {
                ModelState.AddModelError("File",
                    $"The request couldn't be processed (Error 2).");
                // Log error

                return BadRequest(ModelState);
            }

            using (var streamReader = new StreamReader(
                section.Body,
                encoding,
                detectEncodingFromByteOrderMarks: true,
                bufferSize: 1024,
                leaveOpen: true))
            {
                // The value length limit is enforced by
                // MultipartBodyLengthLimit
                var value = await streamReader.ReadToEndAsync();

                if (string.Equals(value, "undefined",
                    StringComparison.OrdinalIgnoreCase))
                {
                    value = string.Empty;
                }

                formAccumulator.Append(key, value);

                if (formAccumulator.ValueCount >
                    _defaultFormOptions.ValueCountLimit)
                {
                    // Form key count limit of
                    // _defaultFormOptions.ValueCountLimit
                    // is exceeded.
                    ModelState.AddModelError("File",
                        $"The request couldn't be processed (Error 3).");
                    // Log error

                    return BadRequest(ModelState);
                }
            }
        }
    }
```

```
        }

        // Drain any remaining section body that hasn't been consumed and
        // read the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    // Bind form data to the model
    var formData = new FormData();
    var formValueProvider = new FormValueProvider(
        BindingSource.Form,
        new FormCollection(formAccumulator.GetResults()),
        CultureInfo.CurrentCulture);
    var bindingSuccessful = await TryUpdateModelAsync(formData, prefix: "",
        valueProvider: formValueProvider);

    if (!bindingSuccessful)
    {
        ModelState.AddModelError("File",
            "The request couldn't be processed (Error 5).");
        // Log error

        return BadRequest(ModelState);
    }

    // **WARNING!**
    // In the following example, the file is saved without
    // scanning the file's contents. In most production
    // scenarios, an anti-virus/anti-malware scanner API
    // is used on the file before making the file available
    // for download or for use by other systems.
    // For more information, see the topic that accompanies
    // this sample app.

    var file = new AppFile()
    {
        Content = streamedFileContent,
        UntrustedName = untrustedFileNameForStorage,
        Note = formData.Note,
        Size = streamedFileContent.Length,
        UploadDT = DateTime.UtcNow
    };

    _context.File.Add(file);
    await _context.SaveChangesAsync();

    return Created(nameof(StreamingController), null);
}
```

`MultipartRequestHelper` (*Utilities/MultipartRequestHelper.cs*):

```
using System;
using System.IO;
using Microsoft.Net.Http.Headers;

namespace SampleApp.Utilities
{
    public static class MultipartRequestHelper
    {
        // Content-Type: multipart/form-data; boundary="----WebKitFormBoundarymx2fSWqWSd0OxQqq"
        // The spec at https://tools.ietf.org/html/rfc2046#section-5.1 states that 70 characters is a
reasonable limit.
        public static string GetBoundary(MediaTypeHeaderValue contentType, int lengthLimit)
        {
            var boundary = HeaderUtilities.RemoveQuotes(contentType.Boundary).Value;

            if (string.IsNullOrWhiteSpace(boundary))
            {
                throw new InvalidDataException("Missing content-type boundary.");
            }

            if (boundary.Length > lengthLimit)
            {
                throw new InvalidDataException(
                    $"Multipart boundary length limit {lengthLimit} exceeded.");
            }

            return boundary;
        }

        public static bool IsMultipartContentType(string contentType)
        {
            return !string.IsNullOrEmpty(contentType)
                   && contentType.IndexOf("multipart/", StringComparison.OrdinalIgnoreCase) >= 0;
        }

        public static bool HasFormDataContentDisposition(ContentDispositionHeaderValue contentDisposition)
        {
            // Content-Disposition: form-data; name="key";
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && string.IsNullOrEmpty(contentDisposition.FileName.Value)
                && string.IsNullOrEmpty(contentDisposition.FileNameStar.Value);
        }

        public static bool HasFileContentDisposition(ContentDispositionHeaderValue contentDisposition)
        {
            // Content-Disposition: form-data; name="myfile1"; filename="Misc 002.jpg"
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && (!string.IsNullOrEmpty(contentDisposition.FileName.Value)
                    || !string.IsNullOrEmpty(contentDisposition.FileNameStar.Value));
        }
    }
}
```

The complete `StreamingController.UploadPhysical` method for streaming to a physical location:

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadPhysical()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
```

```
            // Log error

            return BadRequest(ModelState);
        }

        var boundary = MultipartRequestHelper.GetBoundary(
            MediaTypeHeaderValue.Parse(Request.ContentType),
            _defaultFormOptions.MultipartBoundaryLengthLimit);
        var reader = new MultipartReader(boundary, HttpContext.Request.Body);
        var section = await reader.ReadNextSectionAsync();

        while (section != null)
        {
            var hasContentDispositionHeader =
                ContentDispositionHeaderValue.TryParse(
                    section.ContentDisposition, out var contentDisposition);

            if (hasContentDispositionHeader)
            {
                // This check assumes that there's a file
                // present without form data. If form data
                // is present, this method immediately fails
                // and returns the model error.
                if (!MultipartRequestHelper
                    .HasFileContentDisposition(contentDisposition))
                {
                    ModelState.AddModelError("File",
                        $"The request couldn't be processed (Error 2).");
                    // Log error

                    return BadRequest(ModelState);
                }
                else
                {
                    // Don't trust the file name sent by the client. To display
                    // the file name, HTML-encode the value.
                    var trustedFileNameForDisplay = WebUtility.HtmlEncode(
                            contentDisposition.FileName.Value);
                    var trustedFileNameForFileStorage = Path.GetRandomFileName();

                    // **WARNING!**
                    // In the following example, the file is saved without
                    // scanning the file's contents. In most production
                    // scenarios, an anti-virus/anti-malware scanner API
                    // is used on the file before making the file available
                    // for download or for use by other systems.
                    // For more information, see the topic that accompanies
                    // this sample.

                    var streamedFileContent = await FileHelpers.ProcessStreamedFile(
                        section, contentDisposition, ModelState,
                        _permittedExtensions, _fileSizeLimit);

                    if (!ModelState.IsValid)
                    {
                        return BadRequest(ModelState);
                    }

                    using (var targetStream = System.IO.File.Create(
                        Path.Combine(_targetFilePath, trustedFileNameForFileStorage)))
                    {
                        await targetStream.WriteAsync(streamedFileContent);

                        _logger.LogInformation(
                            "Uploaded file '{TrustedFileNameForDisplay}' saved to " +
                            "'{TargetFilePath}' as {TrustedFileNameForFileStorage}",
                            trustedFileNameForDisplay, _targetFilePath,
                            trustedFileNameForFileStorage);
                    }
```

```
            }
        }

        // Drain any remaining section body that hasn't been consumed and
        // read the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    return Created(nameof(StreamingController), null);
}
```

In the sample app, validation checks are handled by `FileHelpers.ProcessStreamedFile`.

Validation

The sample app's `FileHelpers` class demonstrates a several checks for buffered IFormFile and streamed file uploads. For processing IFormFile buffered file uploads in the sample app, see the `ProcessFormFile` method in the *Utilities/FileHelpers.cs* file. For processing streamed files, see the `ProcessStreamedFile` method in the same file.

> **WARNING**
>
> The validation processing methods demonstrated in the sample app don't scan the content of uploaded files. In most production scenarios, a virus/malware scanner API is used on the file before making the file available to users or other systems.
>
> Although the topic sample provides a working example of validation techniques, don't implement the `FileHelpers` class in a production app unless you:
>
> - Fully understand the implementation.
> - Modify the implementation as appropriate for the app's environment and specifications.
>
> **Never indiscriminately implement security code in an app without addressing these requirements.**

Content validation

Use a third party virus/malware scanning API on uploaded content.

Scanning files is demanding on server resources in high volume scenarios. If request processing performance is diminished due to file scanning, consider offloading the scanning work to a background service, possibly a service running on a server different from the app's server. Typically, uploaded files are held in a quarantined area until the background virus scanner checks them. When a file passes, the file is moved to the normal file storage location. These steps are usually performed in conjunction with a database record that indicates the scanning status of a file. By using such an approach, the app and app server remain focused on responding to requests.

File extension validation

The uploaded file's extension should be checked against a list of permitted extensions. For example:

```
private string[] permittedExtensions = { ".txt", ".pdf" };

var ext = Path.GetExtension(uploadedFileName).ToLowerInvariant();

if (string.IsNullOrEmpty(ext) || !permittedExtensions.Contains(ext))
{
    // The extension is invalid ... discontinue processing the file
}
```

File signature validation

A file's signature is determined by the first few bytes at the start of a file. These bytes can be used to indicate if the

extension matches the content of the file. The sample app checks file signatures for a few common file types. In the following example, the file signature for a JPEG image is checked against the file:

```
private static readonly Dictionary<string, List<byte[]>> _fileSignature =
    new Dictionary<string, List<byte[]>>
{
    { ".jpeg", new List<byte[]>
        {
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE0 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE2 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE3 },
        }
    },
};

using (var reader = new BinaryReader(uploadedFileData))
{
    var signatures = _fileSignature[ext];
    var headerBytes = reader.ReadBytes(signatures.Max(m => m.Length));

    return signatures.Any(signature =>
        headerBytes.Take(signature.Length).SequenceEqual(signature));
}
```

To obtain additional file signatures, see the File Signatures Database and official file specifications.

File name security

Never use a client-supplied file name for saving a file to physical storage. Create a safe file name for the file using Path.GetRandomFileName or Path.GetTempFileName to create a full path (including the file name) for temporary storage.

Razor automatically HTML encodes property values for display. The following code is safe to use:

```
@foreach (var file in Model.DatabaseFiles) {
    <tr>
        <td>
            @file.UntrustedName
        </td>
    </tr>
}
```

Outside of Razor, always HtmlEncode file name content from a user's request.

Many implementations must include a check that the file exists; otherwise, the file is overwritten by a file of the same name. Supply additional logic to meet your app's specifications.

Size validation

Limit the size of uploaded files.

In the sample app, the size of the file is limited to 2 MB (indicated in bytes). The limit is supplied via Configuration from the *appsettings.json* file:

```
{
  "FileSizeLimit": 2097152
}
```

The `FileSizeLimit` is injected into `PageModel` classes:

```
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    private readonly long _fileSizeLimit;

    public BufferedSingleFileUploadPhysicalModel(IConfiguration config)
    {
        _fileSizeLimit = config.GetValue<long>("FileSizeLimit");
    }

    ...
}
```

When a file size exceeds the limit, the file is rejected:

```
if (formFile.Length > _fileSizeLimit)
{
    // The file is too large ... discontinue processing the file
}
```

Match name attribute value to parameter name of POST method

In non-Razor forms that POST form data or use JavaScript's `FormData` directly, the name specified in the form's element or `FormData` must match the name of the parameter in the controller's action.

In the following example:

- When using an `<input>` element, the `name` attribute is set to the value `battlePlans`:

  ```
  <input type="file" name="battlePlans" multiple>
  ```

- When using `FormData` in JavaScript, the name is set to the value `battlePlans`:

  ```
  var formData = new FormData();

  for (var file in files) {
    formData.append("battlePlans", file, file.name);
  }
  ```

Use a matching name for the parameter of the C# method (`battlePlans`):

- For a Razor Pages page handler method named `Upload`:

  ```
  public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> battlePlans)
  ```

- For an MVC POST controller action method:

  ```
  public async Task<IActionResult> Post(List<IFormFile> battlePlans)
  ```

Server and app configuration

Multipart body length limit

MultipartBodyLengthLimit sets the limit for the length of each multipart body. Form sections that exceed this limit throw an InvalidDataException when parsed. The default is 134,217,728 (128 MB). Customize the limit using the MultipartBodyLengthLimit setting in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<FormOptions>(options =>
    {
        // Set the limit to 256 MB
        options.MultipartBodyLengthLimit = 268435456;
    });
}
```

RequestFormLimitsAttribute is used to set the MultipartBodyLengthLimit for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",
            model.Filters.Add(
                new RequestFormLimitsAttribute()
                {
                    // Set the limit to 256 MB
                    MultipartBodyLengthLimit = 268435456
                });
});
```

In a Razor Pages app or an MVC app, apply the filter to the page model or action method:

```
// Set the limit to 256 MB
[RequestFormLimits(MultipartBodyLengthLimit = 268435456)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

Kestrel maximum request body size

For apps hosted by Kestrel, the default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit using the MaxRequestBodySize Kestrel server option:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel((context, options) =>
            {
                // Handle requests up to 50 MB
                options.Limits.MaxRequestBodySize = 52428800;
            })
            .UseStartup<Startup>();
        });
```

RequestSizeLimitAttribute is used to set the MaxRequestBodySize for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",
            model =>
            {
                // Handle requests up to 50 MB
                model.Filters.Add(
                    new RequestSizeLimitAttribute(52428800));
            });
});
```

In a Razor pages app or an MVC app, apply the filter to the page handler class or action method:

```
// Handle requests up to 50 MB
[RequestSizeLimit(52428800)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

The `RequestSizeLimitAttribute` can also be applied using the `@attribute` Razor directive:

```
@attribute [RequestSizeLimitAttribute(52428800)]
```

Other Kestrel limits

Other Kestrel limits may apply for apps hosted by Kestrel:

- Maximum client connections
- Request and response data rates

IIS

The default request limit (`maxAllowedContentLength`) is 30,000,000 bytes, which is approximately 28.6 MB.
Customize the limit in the `web.config` file. In the following example, the limit is set to 50 MB (52,428,800 bytes):

```
<system.webServer>
  <security>
    <requestFiltering>
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

The `maxAllowedContentLength` setting only applies to IIS. For more information, see Request Limits `<requestLimits>`.

Troubleshoot

Below are some common problems encountered when working with uploading files and their possible solutions.

Not Found error when deployed to an IIS server

The following error indicates that the uploaded file exceeds the server's configured content length:

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds the request content length.
```

For more information, see the IIS section.

Connection failure

A connection error and a reset server connection probably indicates that the uploaded file exceeds Kestrel's maximum request body size. For more information, see the Kestrel maximum request body size section. Kestrel client connection limits may also require adjustment.

Null Reference Exception with IFormFile

If the controller is accepting uploaded files using IFormFile but the value is `null`, confirm that the HTML form is specifying an `enctype` value of `multipart/form-data`. If this attribute isn't set on the `<form>` element, the file upload doesn't occur and any bound IFormFile arguments are `null`. Also confirm that the upload naming in form data matches the app's naming.

Stream was too long

The examples in this topic rely upon MemoryStream to hold the uploaded file's content. The size limit of a `MemoryStream` is `int.MaxValue`. If the app's file upload scenario requires holding file content larger than 50 MB, use an alternative approach that doesn't rely upon a single `MemoryStream` for holding an uploaded file's content.

ASP.NET Core supports uploading one or more files using buffered model binding for smaller files and unbuffered streaming for larger files.

View or download sample code (how to download)

Security considerations

Use caution when providing users with the ability to upload files to a server. Attackers may attempt to:

- Execute denial of service attacks.
- Upload viruses or malware.
- Compromise networks and servers in other ways.

Security steps that reduce the likelihood of a successful attack are:

- Upload files to a dedicated file upload area, preferably to a non-system drive. A dedicated location makes it easier to impose security restrictions on uploaded files. Disable execute permissions on the file upload location.†
- Do **not** persist uploaded files in the same directory tree as the app.†
- Use a safe file name determined by the app. Don't use a file name provided by the user or the untrusted file name of the uploaded file.† HTML encode the untrusted file name when displaying it. For example, logging the file name or displaying in UI (Razor automatically HTML encodes output).
- Allow only approved file extensions for the app's design specification.†
- Verify that client-side checks are performed on the server.† Client-side checks are easy to circumvent.
- Check the size of an uploaded file. Set a maximum size limit to prevent large uploads.†
- When files shouldn't be overwritten by an uploaded file with the same name, check the file name against the database or physical storage before uploading the file.
- **Run a virus/malware scanner on uploaded content before the file is stored.**

†The sample app demonstrates an approach that meets the criteria.

WARNING

Uploading malicious code to a system is frequently the first step to executing code that can:

- Completely gain control of a system.
- Overload a system with the result that the system crashes.
- Compromise user or system data.
- Apply graffiti to a public UI.

For information on reducing the attack surface area when accepting files from users, see the following resources:

- Unrestricted File Upload
- Azure Security: Ensure appropriate controls are in place when accepting files from users

For more information on implementing security measures, including examples from the sample app, see the Validation section.

Storage scenarios

Common storage options for files include:

- Database
  - For small file uploads, a database is often faster than physical storage (file system or network share) options.
  - A database is often more convenient than physical storage options because retrieval of a database record for user data can concurrently supply the file content (for example, an avatar image).
  - A database is potentially less expensive than using a data storage service.
- Physical storage (file system or network share)
  - For large file uploads:
    - Database limits may restrict the size of the upload.
    - Physical storage is often less economical than storage in a database.
  - Physical storage is potentially less expensive than using a data storage service.
  - The app's process must have read and write permissions to the storage location. **Never grant execute permission.**
- Data storage service (for example, Azure Blob Storage)
  - Services usually offer improved scalability and resiliency over on-premises solutions that are usually subject to single points of failure.
  - Services are potentially lower cost in large storage infrastructure scenarios.

  For more information, see Quickstart: Use .NET to create a blob in object storage.

File upload scenarios

Two general approaches for uploading files are buffering and streaming.

Buffering

The entire file is read into an IFormFile, which is a C# representation of the file used to process or save the file.

The resources (disk, memory) used by file uploads depend on the number and size of concurrent file uploads. If an app attempts to buffer too many uploads, the site crashes when it runs out of memory or disk space. If the size or frequency of file uploads is exhausting app resources, use streaming.

> **NOTE**
>
> Any single buffered file exceeding 64 KB is moved from memory to a temp file on disk.

Buffering small files is covered in the following sections of this topic:

- Physical storage
- Database

Streaming

The file is received from a multipart request and directly processed or saved by the app. Streaming doesn't improve performance significantly. Streaming reduces the demands for memory or disk space when uploading files.

Streaming large files is covered in the Upload large files with streaming section.

Upload small files with buffered model binding to physical storage

To upload small files, use a multipart form or construct a POST request using JavaScript.

The following example demonstrates the use of a Razor Pages form to upload a single file (*Pages/BufferedSingleFileUploadPhysical.cshtml* in the sample app):

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file">
            <span asp-validation-for="FileUpload.FormFile"></span>
        </dd>
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit" value="Upload" />
</form>
```

The following example is analogous to the prior example except that:

- JavaScript's (Fetch API) is used to submit the form's data.
- There's no validation.

```
<form action="BufferedSingleFileUploadPhysical/?handler=Upload"
      enctype="multipart/form-data" onsubmit="AJAXSubmit(this);return false;"
      method="post">
    <dl>
        <dt>
            <label for="FileUpload_FormFile">File</label>
        </dt>
        <dd>
            <input id="FileUpload_FormFile" type="file"
                name="FileUpload.FormFile" />
        </dd>
    </dl>

    <input class="btn" type="submit" value="Upload" />

    <div style="margin-top:15px">
        <output name="result"></output>
    </div>
</form>

<script>
  async function AJAXSubmit (oFormElement) {
    var resultElement = oFormElement.elements.namedItem("result");
    const formData = new FormData(oFormElement);

    try {
    const response = await fetch(oFormElement.action, {
      method: 'POST',
      body: formData
    });

    if (response.ok) {
      window.location.href = '/';
    }

    resultElement.value = 'Result: ' + response.status + ' ' +
      response.statusText;
    } catch (error) {
      console.error('Error:', error);
    }
  }
</script>
```

To perform the form POST in JavaScript for clients that don't support the Fetch API, use one of the following approaches:

- Use a Fetch Polyfill (for example, window.fetch polyfill (github/fetch)).
- Use `XMLHttpRequest`. For example:

```
  <script>
    "use strict";

    function AJAXSubmit (oFormElement) {
      var oReq = new XMLHttpRequest();
      oReq.onload = function(e) {
      oFormElement.elements.namedItem("result").value =
        'Result: ' + this.status + ' ' + this.statusText;
      };
      oReq.open("post", oFormElement.action);
      oReq.send(new FormData(oFormElement));
    }
  </script>
```

In order to support file uploads, HTML forms must specify an encoding type (`enctype`) of `multipart/form-data` .

For a `files` input element to support uploading multiple files provide the `multiple` attribute on the `<input>` element:

```
<input asp-for="FileUpload.FormFiles" type="file" multiple>
```

The individual files uploaded to the server can be accessed through Model Binding using IFormFile. The sample app demonstrates multiple buffered file uploads for database and physical storage scenarios.

WARNING

Do **not** use the `FileName` property of IFormFile other than for display and logging. When displaying or logging, HTML encode the file name. An attacker can provide a malicious filename, including full paths or relative paths. Applications should:

- Remove the path from the user-supplied filename.
- Save the HTML-encoded, path-removed filename for UI or logging.
- Generate a new random filename for storage.

The following code removes the path from the file name:

```
string untrustedFileName = Path.GetFileName(pathName);
```

The examples provided thus far don't take into account security considerations. Additional information is provided by the following sections and the sample app:

- Security considerations
- Validation

When uploading files using model binding and IFormFile, the action method can accept:

- A single IFormFile.
- Any of the following collections that represent several files:
  - IFormFileCollection
  - IEnumerable<IFormFile>
  - List<IFormFile>

NOTE

Binding matches form files by name. For example, the HTML `name` value in `<input type="file" name="formFile">` must match the C# parameter/property bound (`FormFile`). For more information, see the Match name attribute value to parameter name of POST method section.

The following example:

- Loops through one or more uploaded files.
- Uses Path.GetTempFileName to return a full path for a file, including the file name.
- Saves the files to the local file system using a file name generated by the app.
- Returns the total number and size of files uploaded.

```
public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            var filePath = Path.GetTempFileName();

            using (var stream = System.IO.File.Create(filePath))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // Process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size });
}
```

Use `Path.GetRandomFileName` to generate a file name without a path. In the following example, the path is obtained from configuration:

```
foreach (var formFile in files)
{
    if (formFile.Length > 0)
    {
        var filePath = Path.Combine(_config["StoredFilesPath"],
            Path.GetRandomFileName());

        using (var stream = System.IO.File.Create(filePath))
        {
            await formFile.CopyToAsync(stream);
        }
    }
}
```

The path passed to the FileStream *must* include the file name. If the file name isn't provided, an UnauthorizedAccessException is thrown at runtime.

Files uploaded using the IFormFile technique are buffered in memory or on disk on the server before processing. Inside the action method, the IFormFile contents are accessible as a Stream. In addition to the local file system, files can be saved to a network share or to a file storage service, such as Azure Blob storage.

For another example that loops over multiple files for upload and uses safe file names, see *Pages/BufferedMultipleFileUploadPhysical.cshtml.cs* in the sample app.

> **WARNING**
>
> Path.GetTempFileName throws an IOException if more than 65,535 files are created without deleting previous temporary files. The limit of 65,535 files is a per-server limit. For more information on this limit on Windows OS, see the remarks in the following topics:
>
> - GetTempFileNameA function
> - GetTempFileName

Upload small files with buffered model binding to a database

To store binary file data in a database using Entity Framework, define a Byte array property on the entity:

```
public class AppFile
{
    public int Id { get; set; }
    public byte[] Content { get; set; }
}
```

Specify a page model property for the class that includes an IFormFile:

```
public class BufferedSingleFileUploadDbModel : PageModel
{
    ...

    [BindProperty]
    public BufferedSingleFileUploadDb FileUpload { get; set; }

    ...
}

public class BufferedSingleFileUploadDb
{
    [Required]
    [Display(Name="File")]
    public IFormFile FormFile { get; set; }
}
```

> **NOTE**
>
> IFormFile can be used directly as an action method parameter or as a bound model property. The prior example uses a bound model property.

The `FileUpload` is used in the Razor Pages form:

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file">
        </dd>
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit" value="Upload">
</form>
```

When the form is POSTed to the server, copy the IFormFile to a stream and save it as a byte array in the database. In the following example, `_dbContext` stores the app's database context:

```
public async Task<IActionResult> OnPostUploadAsync()
{
    using (var memoryStream = new MemoryStream())
    {
        await FileUpload.FormFile.CopyToAsync(memoryStream);

        // Upload the file if less than 2 MB
        if (memoryStream.Length < 2097152)
        {
            var file = new AppFile()
            {
                Content = memoryStream.ToArray()
            };

            _dbContext.File.Add(file);

            await _dbContext.SaveChangesAsync();
        }
        else
        {
            ModelState.AddModelError("File", "The file is too large.");
        }
    }

    return Page();
}
```

The preceding example is similar to a scenario demonstrated in the sample app:

- *Pages/BufferedSingleFileUploadDb.cshtml*
- *Pages/BufferedSingleFileUploadDb.cshtml.cs*

WARNING

Use caution when storing binary data in relational databases, as it can adversely impact performance.

Don't rely on or trust the `FileName` property of IFormFile without validation. The `FileName` property should only be used for display purposes and only after HTML encoding.

The examples provided don't take into account security considerations. Additional information is provided by the following sections and the sample app:

- Security considerations
- Validation

Upload large files with streaming

The following example demonstrates how to use JavaScript to stream a file to a controller action. The file's antiforgery token is generated using a custom filter attribute and passed to the client HTTP headers instead of in the request body. Because the action method processes the uploaded data directly, form model binding is disabled by another custom filter. Within the action, the form's contents are read using a `MultipartReader`, which reads each individual `MultipartSection`, processing the file or storing the contents as appropriate. After the multipart sections are read, the action performs its own model binding.

The initial page response loads the form and saves an antiforgery token in a cookie (via the `GenerateAntiforgeryTokenCookieAttribute` attribute). The attribute uses ASP.NET Core's built-in antiforgery support to set a cookie with a request token:

```
public class GenerateAntiforgeryTokenCookieAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        var antiforgery = context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // Send the request token as a JavaScript-readable cookie
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);

        context.HttpContext.Response.Cookies.Append(
            "RequestVerificationToken",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }

    public override void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

The `DisableFormValueModelBindingAttribute` is used to disable model binding:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<FormFileValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

In the sample app, `GenerateAntiforgeryTokenCookieAttribute` and `DisableFormValueModelBindingAttribute` are applied as filters to the page application models of `/StreamedSingleFileUploadDb` and `/StreamedSingleFileUploadPhysical` in `Startup.ConfigureServices` using Razor Pages conventions:

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/StreamedSingleFileUploadDb",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
    options.Conventions
        .AddPageApplicationModelConvention("/StreamedSingleFileUploadPhysical",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
});
```

Since model binding doesn't read the form, parameters that are bound from the form don't bind (query, route, and header continue to work). The action method works directly with the `Request` property. A `MultipartReader` is used to read each section. Key/value data is stored in a `KeyValueAccumulator`. After the multipart sections are read, the contents of the `KeyValueAccumulator` are used to bind the form data to a model type.

The complete `StreamingController.UploadDatabase` method for streaming to a database with EF Core:

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadDatabase()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
        // Log error

        return BadRequest(ModelState);
    }

    // Accumulate the form data key-value pairs in the request (formAccumulator).
    var formAccumulator = new KeyValueAccumulator();
    var trustedFileNameForDisplay = string.Empty;
    var untrustedFileNameForStorage = string.Empty;
    var streamedFileContent = new byte[0];

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();

    while (section != null)
    {
        var hasContentDispositionHeader =
            ContentDispositionHeaderValue.TryParse(
                section.ContentDisposition, out var contentDisposition);

        if (hasContentDispositionHeader)
        {
            if (MultipartRequestHelper
```

```
            .HasFileContentDisposition(contentDisposition))
        {
            untrustedFileNameForStorage = contentDisposition.FileName.Value;
            // Don't trust the file name sent by the client. To display
            // the file name, HTML-encode the value.
            trustedFileNameForDisplay = WebUtility.HtmlEncode(
                    contentDisposition.FileName.Value);

            streamedFileContent =
                await FileHelpers.ProcessStreamedFile(section, contentDisposition,
                    ModelState, _permittedExtensions, _fileSizeLimit);

            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
        }
        else if (MultipartRequestHelper
            .HasFormDataContentDisposition(contentDisposition))
        {
            // Don't limit the key name length because the
            // multipart headers length limit is already in effect.
            var key = HeaderUtilities
                .RemoveQuotes(contentDisposition.Name).Value;
            var encoding = GetEncoding(section);

            if (encoding == null)
            {
                ModelState.AddModelError("File",
                    $"The request couldn't be processed (Error 2).");
                // Log error

                return BadRequest(ModelState);
            }

            using (var streamReader = new StreamReader(
                section.Body,
                encoding,
                detectEncodingFromByteOrderMarks: true,
                bufferSize: 1024,
                leaveOpen: true))
            {
                // The value length limit is enforced by
                // MultipartBodyLengthLimit
                var value = await streamReader.ReadToEndAsync();

                if (string.Equals(value, "undefined",
                    StringComparison.OrdinalIgnoreCase))
                {
                    value = string.Empty;
                }

                formAccumulator.Append(key, value);

                if (formAccumulator.ValueCount >
                    _defaultFormOptions.ValueCountLimit)
                {
                    // Form key count limit of
                    // _defaultFormOptions.ValueCountLimit
                    // is exceeded.
                    ModelState.AddModelError("File",
                        $"The request couldn't be processed (Error 3).");
                    // Log error

                    return BadRequest(ModelState);
                }
            }
        }
    }
```

```
        }

        // Drain any remaining section body that hasn't been consumed and
        // read the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    // Bind form data to the model
    var formData = new FormData();
    var formValueProvider = new FormValueProvider(
        BindingSource.Form,
        new FormCollection(formAccumulator.GetResults()),
        CultureInfo.CurrentCulture);
    var bindingSuccessful = await TryUpdateModelAsync(formData, prefix: "",
        valueProvider: formValueProvider);

    if (!bindingSuccessful)
    {
        ModelState.AddModelError("File",
            "The request couldn't be processed (Error 5).");
        // Log error

        return BadRequest(ModelState);
    }

    // **WARNING!**
    // In the following example, the file is saved without
    // scanning the file's contents. In most production
    // scenarios, an anti-virus/anti-malware scanner API
    // is used on the file before making the file available
    // for download or for use by other systems.
    // For more information, see the topic that accompanies
    // this sample app.

    var file = new AppFile()
    {
        Content = streamedFileContent,
        UntrustedName = untrustedFileNameForStorage,
        Note = formData.Note,
        Size = streamedFileContent.Length,
        UploadDT = DateTime.UtcNow
    };

    _context.File.Add(file);
    await _context.SaveChangesAsync();

    return Created(nameof(StreamingController), null);
}
```

`MultipartRequestHelper` (*Utilities/MultipartRequestHelper.cs*):

```
using System;
using System.IO;
using Microsoft.Net.Http.Headers;

namespace SampleApp.Utilities
{
    public static class MultipartRequestHelper
    {
        // Content-Type: multipart/form-data; boundary="----WebKitFormBoundarymx2fSWqWSd0OxQqq"
        // The spec at https://tools.ietf.org/html/rfc2046#section-5.1 states that 70 characters is a
reasonable limit.
        public static string GetBoundary(MediaTypeHeaderValue contentType, int lengthLimit)
        {
            var boundary = HeaderUtilities.RemoveQuotes(contentType.Boundary).Value;

            if (string.IsNullOrWhiteSpace(boundary))
            {
                throw new InvalidDataException("Missing content-type boundary.");
            }

            if (boundary.Length > lengthLimit)
            {
                throw new InvalidDataException(
                    $"Multipart boundary length limit {lengthLimit} exceeded.");
            }

            return boundary;
        }

        public static bool IsMultipartContentType(string contentType)
        {
            return !string.IsNullOrEmpty(contentType)
                   && contentType.IndexOf("multipart/", StringComparison.OrdinalIgnoreCase) >= 0;
        }

        public static bool HasFormDataContentDisposition(ContentDispositionHeaderValue contentDisposition)
        {
            // Content-Disposition: form-data; name="key";
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && string.IsNullOrEmpty(contentDisposition.FileName.Value)
                && string.IsNullOrEmpty(contentDisposition.FileNameStar.Value);
        }

        public static bool HasFileContentDisposition(ContentDispositionHeaderValue contentDisposition)
        {
            // Content-Disposition: form-data; name="myfile1"; filename="Misc 002.jpg"
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && (!string.IsNullOrEmpty(contentDisposition.FileName.Value)
                    || !string.IsNullOrEmpty(contentDisposition.FileNameStar.Value));
        }
    }
}
```

The complete `StreamingController.UploadPhysical` method for streaming to a physical location:

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadPhysical()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
```

```
            // Log error

            return BadRequest(ModelState);
        }

        var boundary = MultipartRequestHelper.GetBoundary(
            MediaTypeHeaderValue.Parse(Request.ContentType),
            _defaultFormOptions.MultipartBoundaryLengthLimit);
        var reader = new MultipartReader(boundary, HttpContext.Request.Body);
        var section = await reader.ReadNextSectionAsync();

        while (section != null)
        {
            var hasContentDispositionHeader =
                ContentDispositionHeaderValue.TryParse(
                    section.ContentDisposition, out var contentDisposition);

            if (hasContentDispositionHeader)
            {
                // This check assumes that there's a file
                // present without form data. If form data
                // is present, this method immediately fails
                // and returns the model error.
                if (!MultipartRequestHelper
                    .HasFileContentDisposition(contentDisposition))
                {
                    ModelState.AddModelError("File",
                        $"The request couldn't be processed (Error 2).");
                    // Log error

                    return BadRequest(ModelState);
                }
                else
                {
                    // Don't trust the file name sent by the client. To display
                    // the file name, HTML-encode the value.
                    var trustedFileNameForDisplay = WebUtility.HtmlEncode(
                            contentDisposition.FileName.Value);
                    var trustedFileNameForFileStorage = Path.GetRandomFileName();

                    // **WARNING!**
                    // In the following example, the file is saved without
                    // scanning the file's contents. In most production
                    // scenarios, an anti-virus/anti-malware scanner API
                    // is used on the file before making the file available
                    // for download or for use by other systems.
                    // For more information, see the topic that accompanies
                    // this sample.

                    var streamedFileContent = await FileHelpers.ProcessStreamedFile(
                        section, contentDisposition, ModelState,
                        _permittedExtensions, _fileSizeLimit);

                    if (!ModelState.IsValid)
                    {
                        return BadRequest(ModelState);
                    }

                    using (var targetStream = System.IO.File.Create(
                        Path.Combine(_targetFilePath, trustedFileNameForFileStorage)))
                    {
                        await targetStream.WriteAsync(streamedFileContent);

                        _logger.LogInformation(
                            "Uploaded file '{TrustedFileNameForDisplay}' saved to " +
                            "'{TargetFilePath}' as {TrustedFileNameForFileStorage}",
                            trustedFileNameForDisplay, _targetFilePath,
                            trustedFileNameForFileStorage);
                    }
```

```
            }
        }

        // Drain any remaining section body that hasn't been consumed and
        // read the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    return Created(nameof(StreamingController), null);
}
```

In the sample app, validation checks are handled by `FileHelpers.ProcessStreamedFile`.

Validation

The sample app's `FileHelpers` class demonstrates a several checks for buffered IFormFile and streamed file uploads. For processing IFormFile buffered file uploads in the sample app, see the `ProcessFormFile` method in the *Utilities/FileHelpers.cs* file. For processing streamed files, see the `ProcessStreamedFile` method in the same file.

> **WARNING**
>
> The validation processing methods demonstrated in the sample app don't scan the content of uploaded files. In most production scenarios, a virus/malware scanner API is used on the file before making the file available to users or other systems.
>
> Although the topic sample provides a working example of validation techniques, don't implement the `FileHelpers` class in a production app unless you:
>
> - Fully understand the implementation.
> - Modify the implementation as appropriate for the app's environment and specifications.
>
> **Never indiscriminately implement security code in an app without addressing these requirements.**

Content validation

Use a third party virus/malware scanning API on uploaded content.

Scanning files is demanding on server resources in high volume scenarios. If request processing performance is diminished due to file scanning, consider offloading the scanning work to a background service, possibly a service running on a server different from the app's server. Typically, uploaded files are held in a quarantined area until the background virus scanner checks them. When a file passes, the file is moved to the normal file storage location. These steps are usually performed in conjunction with a database record that indicates the scanning status of a file. By using such an approach, the app and app server remain focused on responding to requests.

File extension validation

The uploaded file's extension should be checked against a list of permitted extensions. For example:

```
private string[] permittedExtensions = { ".txt", ".pdf" };

var ext = Path.GetExtension(uploadedFileName).ToLowerInvariant();

if (string.IsNullOrEmpty(ext) || !permittedExtensions.Contains(ext))
{
    // The extension is invalid ... discontinue processing the file
}
```

File signature validation

A file's signature is determined by the first few bytes at the start of a file. These bytes can be used to indicate if the

extension matches the content of the file. The sample app checks file signatures for a few common file types. In the following example, the file signature for a JPEG image is checked against the file:

```
private static readonly Dictionary<string, List<byte[]>> _fileSignature =
    new Dictionary<string, List<byte[]>>
{
    { ".jpeg", new List<byte[]>
        {
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE0 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE2 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE3 },
        }
    },
};

using (var reader = new BinaryReader(uploadedFileData))
{
    var signatures = _fileSignature[ext];
    var headerBytes = reader.ReadBytes(signatures.Max(m => m.Length));

    return signatures.Any(signature =>
        headerBytes.Take(signature.Length).SequenceEqual(signature));
}
```

To obtain additional file signatures, see the File Signatures Database and official file specifications.

File name security

Never use a client-supplied file name for saving a file to physical storage. Create a safe file name for the file using Path.GetRandomFileName or Path.GetTempFileName to create a full path (including the file name) for temporary storage.

Razor automatically HTML encodes property values for display. The following code is safe to use:

```
@foreach (var file in Model.DatabaseFiles) {
    <tr>
        <td>
            @file.UntrustedName
        </td>
    </tr>
}
```

Outside of Razor, always HtmlEncode file name content from a user's request.

Many implementations must include a check that the file exists; otherwise, the file is overwritten by a file of the same name. Supply additional logic to meet your app's specifications.

Size validation

Limit the size of uploaded files.

In the sample app, the size of the file is limited to 2 MB (indicated in bytes). The limit is supplied via Configuration from the *appsettings.json* file:

```
{
  "FileSizeLimit": 2097152
}
```

The `FileSizeLimit` is injected into `PageModel` classes:

```
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    private readonly long _fileSizeLimit;

    public BufferedSingleFileUploadPhysicalModel(IConfiguration config)
    {
        _fileSizeLimit = config.GetValue<long>("FileSizeLimit");
    }

    ...
}
```

When a file size exceeds the limit, the file is rejected:

```
if (formFile.Length > _fileSizeLimit)
{
    // The file is too large ... discontinue processing the file
}
```

Match name attribute value to parameter name of POST method

In non-Razor forms that POST form data or use JavaScript's `FormData` directly, the name specified in the form's element or `FormData` must match the name of the parameter in the controller's action.

In the following example:

- When using an `<input>` element, the `name` attribute is set to the value `battlePlans`:

  ```
  <input type="file" name="battlePlans" multiple>
  ```

- When using `FormData` in JavaScript, the name is set to the value `battlePlans`:

  ```
  var formData = new FormData();

  for (var file in files) {
    formData.append("battlePlans", file, file.name);
  }
  ```

Use a matching name for the parameter of the C# method (`battlePlans`):

- For a Razor Pages page handler method named `Upload`:

  ```
  public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> battlePlans)
  ```

- For an MVC POST controller action method:

  ```
  public async Task<IActionResult> Post(List<IFormFile> battlePlans)
  ```

Server and app configuration

Multipart body length limit

MultipartBodyLengthLimit sets the limit for the length of each multipart body. Form sections that exceed this limit throw an InvalidDataException when parsed. The default is 134,217,728 (128 MB). Customize the limit using the MultipartBodyLengthLimit setting in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<FormOptions>(options =>
    {
        // Set the limit to 256 MB
        options.MultipartBodyLengthLimit = 268435456;
    });
}
```

RequestFormLimitsAttribute is used to set the MultipartBodyLengthLimit for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",
            model.Filters.Add(
                new RequestFormLimitsAttribute()
                {
                    // Set the limit to 256 MB
                    MultipartBodyLengthLimit = 268435456
                });
});
```

In a Razor Pages app or an MVC app, apply the filter to the page model or action method:

```
// Set the limit to 256 MB
[RequestFormLimits(MultipartBodyLengthLimit = 268435456)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

Kestrel maximum request body size

For apps hosted by Kestrel, the default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit using the MaxRequestBodySize Kestrel server option:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel((context, options) =>
            {
                // Handle requests up to 50 MB
                options.Limits.MaxRequestBodySize = 52428800;
            })
            .UseStartup<Startup>();
        });
```

RequestSizeLimitAttribute is used to set the MaxRequestBodySize for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",
            model =>
            {
                // Handle requests up to 50 MB
                model.Filters.Add(
                    new RequestSizeLimitAttribute(52428800));
            });
});
```

In a Razor pages app or an MVC app, apply the filter to the page handler class or action method:

```
// Handle requests up to 50 MB
[RequestSizeLimit(52428800)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

The `RequestSizeLimitAttribute` can also be applied using the `@attribute` Razor directive:

```
@attribute [RequestSizeLimitAttribute(52428800)]
```

Other Kestrel limits

Other Kestrel limits may apply for apps hosted by Kestrel:

- Maximum client connections
- Request and response data rates

IIS

The default request limit (`maxAllowedContentLength`) is 30,000,000 bytes, which is approximately 28.6 MB.
Customize the limit in the `web.config` file. In the following example, the limit is set to 50 MB (52,428,800 bytes):

```
<system.webServer>
  <security>
    <requestFiltering>
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

The `maxAllowedContentLength` setting only applies to IIS. For more information, see Request Limits `<requestLimits>`.

Increase the maximum request body size for the HTTP request by setting IISServerOptions.MaxRequestBodySize in `Startup.ConfigureServices`. In the following example, the limit is set to 50 MB (52,428,800 bytes):

```
services.Configure<IISServerOptions>(options =>
{
    options.MaxRequestBodySize = 52428800;
});
```

For more information, see Host ASP.NET Core on Windows with IIS.

Troubleshoot

Below are some common problems encountered when working with uploading files and their possible solutions.

Not Found error when deployed to an IIS server

The following error indicates that the uploaded file exceeds the server's configured content length:

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds the request content length.
```

For more information, see the IIS section.

Connection failure

A connection error and a reset server connection probably indicates that the uploaded file exceeds Kestrel's maximum request body size. For more information, see the Kestrel maximum request body size section. Kestrel client connection limits may also require adjustment.

Null Reference Exception with IFormFile

If the controller is accepting uploaded files using IFormFile but the value is `null`, confirm that the HTML form is specifying an `enctype` value of `multipart/form-data`. If this attribute isn't set on the `<form>` element, the file upload doesn't occur and any bound IFormFile arguments are `null`. Also confirm that the upload naming in form data matches the app's naming.

Stream was too long

The examples in this topic rely upon MemoryStream to hold the uploaded file's content. The size limit of a `MemoryStream` is `int.MaxValue`. If the app's file upload scenario requires holding file content larger than 50 MB, use an alternative approach that doesn't rely upon a single `MemoryStream` for holding an uploaded file's content.

ASP.NET Core supports uploading one or more files using buffered model binding for smaller files and unbuffered streaming for larger files.

View or download sample code (how to download)

Security considerations

Use caution when providing users with the ability to upload files to a server. Attackers may attempt to:

- Execute denial of service attacks.
- Upload viruses or malware.
- Compromise networks and servers in other ways.

Security steps that reduce the likelihood of a successful attack are:

- Upload files to a dedicated file upload area, preferably to a non-system drive. A dedicated location makes it easier to impose security restrictions on uploaded files. Disable execute permissions on the file upload location.†
- Do **not** persist uploaded files in the same directory tree as the app.†
- Use a safe file name determined by the app. Don't use a file name provided by the user or the untrusted file name of the uploaded file.† HTML encode the untrusted file name when displaying it. For example, logging the file name or displaying in UI (Razor automatically HTML encodes output).
- Allow only approved file extensions for the app's design specification.†
- Verify that client-side checks are performed on the server.† Client-side checks are easy to circumvent.
- Check the size of an uploaded file. Set a maximum size limit to prevent large uploads.†
- When files shouldn't be overwritten by an uploaded file with the same name, check the file name against the database or physical storage before uploading the file.

- **Run a virus/malware scanner on uploaded content before the file is stored.**

†The sample app demonstrates an approach that meets the criteria.

> **WARNING**
> Uploading malicious code to a system is frequently the first step to executing code that can:
>
> - Completely gain control of a system.
> - Overload a system with the result that the system crashes.
> - Compromise user or system data.
> - Apply graffiti to a public UI.
>
> For information on reducing the attack surface area when accepting files from users, see the following resources:
>
> - Unrestricted File Upload
> - Azure Security: Ensure appropriate controls are in place when accepting files from users

For more information on implementing security measures, including examples from the sample app, see the Validation section.

Storage scenarios

Common storage options for files include:

- Database
  - For small file uploads, a database is often faster than physical storage (file system or network share) options.
  - A database is often more convenient than physical storage options because retrieval of a database record for user data can concurrently supply the file content (for example, an avatar image).
  - A database is potentially less expensive than using a data storage service.
- Physical storage (file system or network share)
  - For large file uploads:
    - Database limits may restrict the size of the upload.
    - Physical storage is often less economical than storage in a database.
  - Physical storage is potentially less expensive than using a data storage service.
  - The app's process must have read and write permissions to the storage location. **Never grant execute permission.**
- Data storage service (for example, Azure Blob Storage)
  - Services usually offer improved scalability and resiliency over on-premises solutions that are usually subject to single points of failure.
  - Services are potentially lower cost in large storage infrastructure scenarios.

  For more information, see Quickstart: Use .NET to create a blob in object storage. The topic demonstrates UploadFromFileAsync, but UploadFromStreamAsync can be used to save a FileStream to blob storage when working with a Stream.

File upload scenarios

Two general approaches for uploading files are buffering and streaming.

Buffering

The entire file is read into an IFormFile, which is a C# representation of the file used to process or save the file.

The resources (disk, memory) used by file uploads depend on the number and size of concurrent file uploads. If an app attempts to buffer too many uploads, the site crashes when it runs out of memory or disk space. If the size or frequency of file uploads is exhausting app resources, use streaming.

> **NOTE**
> Any single buffered file exceeding 64 KB is moved from memory to a temp file on disk.

Buffering small files is covered in the following sections of this topic:

- Physical storage
- Database

Streaming

The file is received from a multipart request and directly processed or saved by the app. Streaming doesn't improve performance significantly. Streaming reduces the demands for memory or disk space when uploading files.

Streaming large files is covered in the Upload large files with streaming section.

Upload small files with buffered model binding to physical storage

To upload small files, use a multipart form or construct a POST request using JavaScript.

The following example demonstrates the use of a Razor Pages form to upload a single file (*Pages/BufferedSingleFileUploadPhysical.cshtml* in the sample app):

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file">
            <span asp-validation-for="FileUpload.FormFile"></span>
        </dd>
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit" value="Upload" />
</form>
```

The following example is analogous to the prior example except that:

- JavaScript's (Fetch API) is used to submit the form's data.
- There's no validation.

```
<form action="BufferedSingleFileUploadPhysical/?handler=Upload"
      enctype="multipart/form-data" onsubmit="AJAXSubmit(this);return false;"
      method="post">
    <dl>
        <dt>
            <label for="FileUpload_FormFile">File</label>
        </dt>
        <dd>
            <input id="FileUpload_FormFile" type="file"
                name="FileUpload.FormFile" />
        </dd>
    </dl>

    <input class="btn" type="submit" value="Upload" />

    <div style="margin-top:15px">
        <output name="result"></output>
    </div>
</form>

<script>
  async function AJAXSubmit (oFormElement) {
    var resultElement = oFormElement.elements.namedItem("result");
    const formData = new FormData(oFormElement);

    try {
    const response = await fetch(oFormElement.action, {
      method: 'POST',
      body: formData
    });

    if (response.ok) {
      window.location.href = '/';
    }

    resultElement.value = 'Result: ' + response.status + ' ' +
      response.statusText;
    } catch (error) {
      console.error('Error:', error);
    }
  }
</script>
```

To perform the form POST in JavaScript for clients that don't support the Fetch API, use one of the following approaches:

- Use a Fetch Polyfill (for example, window.fetch polyfill (github/fetch)).
- Use `XMLHttpRequest`. For example:

```
  <script>
    "use strict";

    function AJAXSubmit (oFormElement) {
      var oReq = new XMLHttpRequest();
      oReq.onload = function(e) {
      oFormElement.elements.namedItem("result").value =
        'Result: ' + this.status + ' ' + this.statusText;
      };
      oReq.open("post", oFormElement.action);
      oReq.send(new FormData(oFormElement));
    }
  </script>
```

In order to support file uploads, HTML forms must specify an encoding type (`enctype`) of `multipart/form-data` .

For a `files` input element to support uploading multiple files provide the `multiple` attribute on the `<input>` element:

```
<input asp-for="FileUpload.FormFiles" type="file" multiple>
```

The individual files uploaded to the server can be accessed through Model Binding using IFormFile. The sample app demonstrates multiple buffered file uploads for database and physical storage scenarios.

WARNING

Do **not** use the `FileName` property of IFormFile other than for display and logging. When displaying or logging, HTML encode the file name. An attacker can provide a malicious filename, including full paths or relative paths. Applications should:

- Remove the path from the user-supplied filename.
- Save the HTML-encoded, path-removed filename for UI or logging.
- Generate a new random filename for storage.

The following code removes the path from the file name:

```
string untrustedFileName = Path.GetFileName(pathName);
```

The examples provided thus far don't take into account security considerations. Additional information is provided by the following sections and the sample app:

- Security considerations
- Validation

When uploading files using model binding and IFormFile, the action method can accept:

- A single IFormFile.
- Any of the following collections that represent several files:
  - IFormFileCollection
  - IEnumerable<IFormFile>
  - List<IFormFile>

NOTE

Binding matches form files by name. For example, the HTML `name` value in `<input type="file" name="formFile">` must match the C# parameter/property bound (`FormFile`). For more information, see the Match name attribute value to parameter name of POST method section.

The following example:

- Loops through one or more uploaded files.
- Uses Path.GetTempFileName to return a full path for a file, including the file name.
- Saves the files to the local file system using a file name generated by the app.
- Returns the total number and size of files uploaded.

```
public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            var filePath = Path.GetTempFileName();

            using (var stream = System.IO.File.Create(filePath))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // Process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size });
}
```

Use `Path.GetRandomFileName` to generate a file name without a path. In the following example, the path is obtained from configuration:

```
foreach (var formFile in files)
{
    if (formFile.Length > 0)
    {
        var filePath = Path.Combine(_config["StoredFilesPath"],
            Path.GetRandomFileName());

        using (var stream = System.IO.File.Create(filePath))
        {
            await formFile.CopyToAsync(stream);
        }
    }
}
```

The path passed to the FileStream *must* include the file name. If the file name isn't provided, an UnauthorizedAccessException is thrown at runtime.

Files uploaded using the IFormFile technique are buffered in memory or on disk on the server before processing. Inside the action method, the IFormFile contents are accessible as a Stream. In addition to the local file system, files can be saved to a network share or to a file storage service, such as Azure Blob storage.

For another example that loops over multiple files for upload and uses safe file names, see *Pages/BufferedMultipleFileUploadPhysical.cshtml.cs* in the sample app.

> **WARNING**
>
> Path.GetTempFileName throws an IOException if more than 65,535 files are created without deleting previous temporary files. The limit of 65,535 files is a per-server limit. For more information on this limit on Windows OS, see the remarks in the following topics:
>
> - GetTempFileNameA function
> - GetTempFileName

Upload small files with buffered model binding to a database

To store binary file data in a database using Entity Framework, define a Byte array property on the entity:

```
public class AppFile
{
    public int Id { get; set; }
    public byte[] Content { get; set; }
}
```

Specify a page model property for the class that includes an IFormFile:

```
public class BufferedSingleFileUploadDbModel : PageModel
{
    ...

    [BindProperty]
    public BufferedSingleFileUploadDb FileUpload { get; set; }

    ...
}

public class BufferedSingleFileUploadDb
{
    [Required]
    [Display(Name="File")]
    public IFormFile FormFile { get; set; }
}
```

> **NOTE**
>
> IFormFile can be used directly as an action method parameter or as a bound model property. The prior example uses a bound model property.

The `FileUpload` is used in the Razor Pages form:

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file">
        </dd>
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit" value="Upload">
</form>
```

When the form is POSTed to the server, copy the IFormFile to a stream and save it as a byte array in the database. In the following example, `_dbContext` stores the app's database context:

```
public async Task<IActionResult> OnPostUploadAsync()
{
    using (var memoryStream = new MemoryStream())
    {
        await FileUpload.FormFile.CopyToAsync(memoryStream);

        // Upload the file if less than 2 MB
        if (memoryStream.Length < 2097152)
        {
            var file = new AppFile()
            {
                Content = memoryStream.ToArray()
            };

            _dbContext.File.Add(file);

            await _dbContext.SaveChangesAsync();
        }
        else
        {
            ModelState.AddModelError("File", "The file is too large.");
        }
    }

    return Page();
}
```

The preceding example is similar to a scenario demonstrated in the sample app:

- *Pages/BufferedSingleFileUploadDb.cshtml*
- *Pages/BufferedSingleFileUploadDb.cshtml.cs*

> **WARNING**
>
> Use caution when storing binary data in relational databases, as it can adversely impact performance.
>
> Don't rely on or trust the `FileName` property of IFormFile without validation. The `FileName` property should only be used for display purposes and only after HTML encoding.
>
> The examples provided don't take into account security considerations. Additional information is provided by the following sections and the sample app:
>
> - Security considerations
> - Validation

Upload large files with streaming

The following example demonstrates how to use JavaScript to stream a file to a controller action. The file's antiforgery token is generated using a custom filter attribute and passed to the client HTTP headers instead of in the request body. Because the action method processes the uploaded data directly, form model binding is disabled by another custom filter. Within the action, the form's contents are read using a `MultipartReader`, which reads each individual `MultipartSection`, processing the file or storing the contents as appropriate. After the multipart sections are read, the action performs its own model binding.

The initial page response loads the form and saves an antiforgery token in a cookie (via the `GenerateAntiforgeryTokenCookieAttribute` attribute). The attribute uses ASP.NET Core's built-in antiforgery support to set a cookie with a request token:

```
public class GenerateAntiforgeryTokenCookieAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        var antiforgery = context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // Send the request token as a JavaScript-readable cookie
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);

        context.HttpContext.Response.Cookies.Append(
            "RequestVerificationToken",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }

    public override void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

The `DisableFormValueModelBindingAttribute` is used to disable model binding:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

In the sample app, `GenerateAntiforgeryTokenCookieAttribute` and `DisableFormValueModelBindingAttribute` are applied as filters to the page application models of `/StreamedSingleFileUploadDb` and `/StreamedSingleFileUploadPhysical` in `Startup.ConfigureServices` using Razor Pages conventions:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
        {
            options.Conventions
                .AddPageApplicationModelConvention("/StreamedSingleFileUploadDb",
                    model =>
                    {
                        model.Filters.Add(
                            new GenerateAntiforgeryTokenCookieAttribute());
                        model.Filters.Add(
                            new DisableFormValueModelBindingAttribute());
                    });
            options.Conventions
                .AddPageApplicationModelConvention("/StreamedSingleFileUploadPhysical",
                    model =>
                    {
                        model.Filters.Add(
                            new GenerateAntiforgeryTokenCookieAttribute());
                        model.Filters.Add(
                            new DisableFormValueModelBindingAttribute());
                    });
        })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Since model binding doesn't read the form, parameters that are bound from the form don't bind (query, route, and header continue to work). The action method works directly with the `Request` property. A `MultipartReader` is used to read each section. Key/value data is stored in a `KeyValueAccumulator`. After the multipart sections are read, the contents of the `KeyValueAccumulator` are used to bind the form data to a model type.

The complete `StreamingController.UploadDatabase` method for streaming to a database with EF Core:

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadDatabase()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
        // Log error

        return BadRequest(ModelState);
    }

    // Accumulate the form data key-value pairs in the request (formAccumulator).
    var formAccumulator = new KeyValueAccumulator();
    var trustedFileNameForDisplay = string.Empty;
    var untrustedFileNameForStorage = string.Empty;
    var streamedFileContent = new byte[0];

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();

    while (section != null)
    {
        var hasContentDispositionHeader =
            ContentDispositionHeaderValue.TryParse(
                section.ContentDisposition, out var contentDisposition);

        if (hasContentDispositionHeader)
```

```
        {
            if (MultipartRequestHelper
                .HasFileContentDisposition(contentDisposition))
            {
                untrustedFileNameForStorage = contentDisposition.FileName.Value;
                // Don't trust the file name sent by the client. To display
                // the file name, HTML-encode the value.
                trustedFileNameForDisplay = WebUtility.HtmlEncode(
                        contentDisposition.FileName.Value);

                streamedFileContent =
                    await FileHelpers.ProcessStreamedFile(section, contentDisposition,
                        ModelState, _permittedExtensions, _fileSizeLimit);

                if (!ModelState.IsValid)
                {
                    return BadRequest(ModelState);
                }
            }
            else if (MultipartRequestHelper
                .HasFormDataContentDisposition(contentDisposition))
            {
                // Don't limit the key name length because the
                // multipart headers length limit is already in effect.
                var key = HeaderUtilities
                    .RemoveQuotes(contentDisposition.Name).Value;
                var encoding = GetEncoding(section);

                if (encoding == null)
                {
                    ModelState.AddModelError("File",
                        $"The request couldn't be processed (Error 2).");
                    // Log error

                    return BadRequest(ModelState);
                }

                using (var streamReader = new StreamReader(
                    section.Body,
                    encoding,
                    detectEncodingFromByteOrderMarks: true,
                    bufferSize: 1024,
                    leaveOpen: true))
                {
                    // The value length limit is enforced by
                    // MultipartBodyLengthLimit
                    var value = await streamReader.ReadToEndAsync();

                    if (string.Equals(value, "undefined",
                        StringComparison.OrdinalIgnoreCase))
                    {
                        value = string.Empty;
                    }

                    formAccumulator.Append(key, value);

                    if (formAccumulator.ValueCount >
                        _defaultFormOptions.ValueCountLimit)
                    {
                        // Form key count limit of
                        // _defaultFormOptions.ValueCountLimit
                        // is exceeded.
                        ModelState.AddModelError("File",
                            $"The request couldn't be processed (Error 3).");
                        // Log error

                        return BadRequest(ModelState);
                    }
                }
```

```
                }
            }

            // Drain any remaining section body that hasn't been consumed and
            // read the headers for the next section.
            section = await reader.ReadNextSectionAsync();
        }

        // Bind form data to the model
        var formData = new FormData();
        var formValueProvider = new FormValueProvider(
            BindingSource.Form,
            new FormCollection(formAccumulator.GetResults()),
            CultureInfo.CurrentCulture);
        var bindingSuccessful = await TryUpdateModelAsync(formData, prefix: "",
            valueProvider: formValueProvider);

        if (!bindingSuccessful)
        {
            ModelState.AddModelError("File",
                "The request couldn't be processed (Error 5).");
            // Log error

            return BadRequest(ModelState);
        }

        // **WARNING!**
        // In the following example, the file is saved without
        // scanning the file's contents. In most production
        // scenarios, an anti-virus/anti-malware scanner API
        // is used on the file before making the file available
        // for download or for use by other systems.
        // For more information, see the topic that accompanies
        // this sample app.

        var file = new AppFile()
        {
            Content = streamedFileContent,
            UntrustedName = untrustedFileNameForStorage,
            Note = formData.Note,
            Size = streamedFileContent.Length,
            UploadDT = DateTime.UtcNow
        };

        _context.File.Add(file);
        await _context.SaveChangesAsync();

        return Created(nameof(StreamingController), null);
    }
```

`MultipartRequestHelper` (*Utilities/MultipartRequestHelper.cs*):

```
using System;
using System.IO;
using Microsoft.Net.Http.Headers;

namespace SampleApp.Utilities
{
    public static class MultipartRequestHelper
    {
        // Content-Type: multipart/form-data; boundary="----WebKitFormBoundarymx2fSWqWSd0OxQqq"
        // The spec at https://tools.ietf.org/html/rfc2046#section-5.1 states that 70 characters is a
reasonable limit.
        public static string GetBoundary(MediaTypeHeaderValue contentType, int lengthLimit)
        {
            var boundary = HeaderUtilities.RemoveQuotes(contentType.Boundary).Value;

            if (string.IsNullOrWhiteSpace(boundary))
            {
                throw new InvalidDataException("Missing content-type boundary.");
            }

            if (boundary.Length > lengthLimit)
            {
                throw new InvalidDataException(
                    $"Multipart boundary length limit {lengthLimit} exceeded.");
            }

            return boundary;
        }

        public static bool IsMultipartContentType(string contentType)
        {
            return !string.IsNullOrEmpty(contentType)
                   && contentType.IndexOf("multipart/", StringComparison.OrdinalIgnoreCase) >= 0;
        }

        public static bool HasFormDataContentDisposition(ContentDispositionHeaderValue contentDisposition)
        {
            // Content-Disposition: form-data; name="key";
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && string.IsNullOrEmpty(contentDisposition.FileName.Value)
                && string.IsNullOrEmpty(contentDisposition.FileNameStar.Value);
        }

        public static bool HasFileContentDisposition(ContentDispositionHeaderValue contentDisposition)
        {
            // Content-Disposition: form-data; name="myfile1"; filename="Misc 002.jpg"
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && (!string.IsNullOrEmpty(contentDisposition.FileName.Value)
                    || !string.IsNullOrEmpty(contentDisposition.FileNameStar.Value));
        }
    }
}
```

The complete `StreamingController.UploadPhysical` method for streaming to a physical location:

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadPhysical()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
```

```
            // Log error

            return BadRequest(ModelState);
        }

        var boundary = MultipartRequestHelper.GetBoundary(
            MediaTypeHeaderValue.Parse(Request.ContentType),
            _defaultFormOptions.MultipartBoundaryLengthLimit);
        var reader = new MultipartReader(boundary, HttpContext.Request.Body);
        var section = await reader.ReadNextSectionAsync();

        while (section != null)
        {
            var hasContentDispositionHeader =
                ContentDispositionHeaderValue.TryParse(
                    section.ContentDisposition, out var contentDisposition);

            if (hasContentDispositionHeader)
            {
                // This check assumes that there's a file
                // present without form data. If form data
                // is present, this method immediately fails
                // and returns the model error.
                if (!MultipartRequestHelper
                    .HasFileContentDisposition(contentDisposition))
                {
                    ModelState.AddModelError("File",
                        $"The request couldn't be processed (Error 2).");
                    // Log error

                    return BadRequest(ModelState);
                }
                else
                {
                    // Don't trust the file name sent by the client. To display
                    // the file name, HTML-encode the value.
                    var trustedFileNameForDisplay = WebUtility.HtmlEncode(
                            contentDisposition.FileName.Value);
                    var trustedFileNameForFileStorage = Path.GetRandomFileName();

                    // **WARNING!**
                    // In the following example, the file is saved without
                    // scanning the file's contents. In most production
                    // scenarios, an anti-virus/anti-malware scanner API
                    // is used on the file before making the file available
                    // for download or for use by other systems.
                    // For more information, see the topic that accompanies
                    // this sample.

                    var streamedFileContent = await FileHelpers.ProcessStreamedFile(
                        section, contentDisposition, ModelState,
                        _permittedExtensions, _fileSizeLimit);

                    if (!ModelState.IsValid)
                    {
                        return BadRequest(ModelState);
                    }

                    using (var targetStream = System.IO.File.Create(
                        Path.Combine(_targetFilePath, trustedFileNameForFileStorage)))
                    {
                        await targetStream.WriteAsync(streamedFileContent);

                        _logger.LogInformation(
                            "Uploaded file '{TrustedFileNameForDisplay}' saved to " +
                            "'{TargetFilePath}' as {TrustedFileNameForFileStorage}",
                            trustedFileNameForDisplay, _targetFilePath,
                            trustedFileNameForFileStorage);
                    }
```

```
            }
        }

        // Drain any remaining section body that hasn't been consumed and
        // read the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    return Created(nameof(StreamingController), null);
}
```

In the sample app, validation checks are handled by `FileHelpers.ProcessStreamedFile`.

Validation

The sample app's `FileHelpers` class demonstrates a several checks for buffered IFormFile and streamed file uploads. For processing IFormFile buffered file uploads in the sample app, see the `ProcessFormFile` method in the *Utilities/FileHelpers.cs* file. For processing streamed files, see the `ProcessStreamedFile` method in the same file.

> **WARNING**
>
> The validation processing methods demonstrated in the sample app don't scan the content of uploaded files. In most production scenarios, a virus/malware scanner API is used on the file before making the file available to users or other systems.
>
> Although the topic sample provides a working example of validation techniques, don't implement the `FileHelpers` class in a production app unless you:
>
> - Fully understand the implementation.
> - Modify the implementation as appropriate for the app's environment and specifications.
>
> **Never indiscriminately implement security code in an app without addressing these requirements.**

Content validation

Use a third party virus/malware scanning API on uploaded content.

Scanning files is demanding on server resources in high volume scenarios. If request processing performance is diminished due to file scanning, consider offloading the scanning work to a background service, possibly a service running on a server different from the app's server. Typically, uploaded files are held in a quarantined area until the background virus scanner checks them. When a file passes, the file is moved to the normal file storage location. These steps are usually performed in conjunction with a database record that indicates the scanning status of a file. By using such an approach, the app and app server remain focused on responding to requests.

File extension validation

The uploaded file's extension should be checked against a list of permitted extensions. For example:

```
private string[] permittedExtensions = { ".txt", ".pdf" };

var ext = Path.GetExtension(uploadedFileName).ToLowerInvariant();

if (string.IsNullOrEmpty(ext) || !permittedExtensions.Contains(ext))
{
    // The extension is invalid ... discontinue processing the file
}
```

File signature validation

A file's signature is determined by the first few bytes at the start of a file. These bytes can be used to indicate if the

extension matches the content of the file. The sample app checks file signatures for a few common file types. In the following example, the file signature for a JPEG image is checked against the file:

```
private static readonly Dictionary<string, List<byte[]>> _fileSignature =
    new Dictionary<string, List<byte[]>>
{
    { ".jpeg", new List<byte[]>
        {
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE0 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE2 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE3 },
        }
    },
};

using (var reader = new BinaryReader(uploadedFileData))
{
    var signatures = _fileSignature[ext];
    var headerBytes = reader.ReadBytes(signatures.Max(m => m.Length));

    return signatures.Any(signature =>
        headerBytes.Take(signature.Length).SequenceEqual(signature));
}
```

To obtain additional file signatures, see the File Signatures Database and official file specifications.

File name security

Never use a client-supplied file name for saving a file to physical storage. Create a safe file name for the file using Path.GetRandomFileName or Path.GetTempFileName to create a full path (including the file name) for temporary storage.

Razor automatically HTML encodes property values for display. The following code is safe to use:

```
@foreach (var file in Model.DatabaseFiles) {
    <tr>
        <td>
            @file.UntrustedName
        </td>
    </tr>
}
```

Outside of Razor, always HtmlEncode file name content from a user's request.

Many implementations must include a check that the file exists; otherwise, the file is overwritten by a file of the same name. Supply additional logic to meet your app's specifications.

Size validation

Limit the size of uploaded files.

In the sample app, the size of the file is limited to 2 MB (indicated in bytes). The limit is supplied via Configuration from the *appsettings.json* file:

```
{
  "FileSizeLimit": 2097152
}
```

The `FileSizeLimit` is injected into `PageModel` classes:

```
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    private readonly long _fileSizeLimit;

    public BufferedSingleFileUploadPhysicalModel(IConfiguration config)
    {
        _fileSizeLimit = config.GetValue<long>("FileSizeLimit");
    }

    ...
}
```

When a file size exceeds the limit, the file is rejected:

```
if (formFile.Length > _fileSizeLimit)
{
    // The file is too large ... discontinue processing the file
}
```

Match name attribute value to parameter name of POST method

In non-Razor forms that POST form data or use JavaScript's `FormData` directly, the name specified in the form's element or `FormData` must match the name of the parameter in the controller's action.

In the following example:

- When using an `<input>` element, the `name` attribute is set to the value `battlePlans`:

  ```
  <input type="file" name="battlePlans" multiple>
  ```

- When using `FormData` in JavaScript, the name is set to the value `battlePlans`:

  ```
  var formData = new FormData();

  for (var file in files) {
    formData.append("battlePlans", file, file.name);
  }
  ```

Use a matching name for the parameter of the C# method (`battlePlans`):

- For a Razor Pages page handler method named `Upload`:

  ```
  public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> battlePlans)
  ```

- For an MVC POST controller action method:

  ```
  public async Task<IActionResult> Post(List<IFormFile> battlePlans)
  ```

Server and app configuration

Multipart body length limit

MultipartBodyLengthLimit sets the limit for the length of each multipart body. Form sections that exceed this limit throw an InvalidDataException when parsed. The default is 134,217,728 (128 MB). Customize the limit using the MultipartBodyLengthLimit setting in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<FormOptions>(options =>
    {
        // Set the limit to 256 MB
        options.MultipartBodyLengthLimit = 268435456;
    });
}
```

RequestFormLimitsAttribute is used to set the MultipartBodyLengthLimit for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions
            .AddPageApplicationModelConvention("/FileUploadPage",
                model.Filters.Add(
                    new RequestFormLimitsAttribute()
                    {
                        // Set the limit to 256 MB
                        MultipartBodyLengthLimit = 268435456
                    });
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

In a Razor Pages app or an MVC app, apply the filter to the page model or action method:

```
// Set the limit to 256 MB
[RequestFormLimits(MultipartBodyLengthLimit = 268435456)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

Kestrel maximum request body size

For apps hosted by Kestrel, the default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit using the MaxRequestBodySize Kestrel server option:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Handle requests up to 50 MB
            options.Limits.MaxRequestBodySize = 52428800;
        });
```

RequestSizeLimitAttribute is used to set the MaxRequestBodySize for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions
            .AddPageApplicationModelConvention("/FileUploadPage",
                model =>
                {
                    // Handle requests up to 50 MB
                    model.Filters.Add(
                        new RequestSizeLimitAttribute(52428800));
                });
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

In a Razor pages app or an MVC app, apply the filter to the page handler class or action method:

```
// Handle requests up to 50 MB
[RequestSizeLimit(52428800)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

Other Kestrel limits

Other Kestrel limits may apply for apps hosted by Kestrel:

- Maximum client connections
- Request and response data rates

IIS

The default request limit (`maxAllowedContentLength`) is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit in the `web.config` file. In the following example, the limit is set to 50 MB (52,428,800 bytes):

```
<system.webServer>
  <security>
    <requestFiltering>
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

The `maxAllowedContentLength` setting only applies to IIS. For more information, see Request Limits `<requestLimits>`.

Increase the maximum request body size for the HTTP request by setting IISServerOptions.MaxRequestBodySize in `Startup.ConfigureServices`. In the following example, the limit is set to 50 MB (52,428,800 bytes):

```
services.Configure<IISServerOptions>(options =>
{
    options.MaxRequestBodySize = 52428800;
});
```

For more information, see Host ASP.NET Core on Windows with IIS.

Troubleshoot

Below are some common problems encountered when working with uploading files and their possible solutions.

Not Found error when deployed to an IIS server

The following error indicates that the uploaded file exceeds the server's configured content length:

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds the request content length.
```

For more information, see the IIS section.

Connection failure

A connection error and a reset server connection probably indicates that the uploaded file exceeds Kestrel's maximum request body size. For more information, see the Kestrel maximum request body size section. Kestrel client connection limits may also require adjustment.

Null Reference Exception with IFormFile

If the controller is accepting uploaded files using IFormFile but the value is `null`, confirm that the HTML form is specifying an `enctype` value of `multipart/form-data`. If this attribute isn't set on the `<form>` element, the file upload doesn't occur and any bound IFormFile arguments are `null`. Also confirm that the upload naming in form data matches the app's naming.

Stream was too long

The examples in this topic rely upon MemoryStream to hold the uploaded file's content. The size limit of a `MemoryStream` is `int.MaxValue`. If the app's file upload scenario requires holding file content larger than 50 MB, use an alternative approach that doesn't rely upon a single `MemoryStream` for holding an uploaded file's content.

Additional resources

- HTTP connection request draining
- Unrestricted File Upload
- Azure Security: Security Frame: Input Validation | Mitigations
- Azure Cloud Design Patterns: Valet Key pattern

ASP.NET Core Web SDK

9/22/2020 • 2 minutes to read • Edit Online

Overview

`Microsoft.NET.Sdk.Web` is an MSBuild project SDK for building ASP.NET Core apps. It's possible to build an ASP.NET Core app without this SDK, however, the Web SDK is:

- Tailored towards providing a first-class experience.
- The recommended target for most users.

Use the Web.SDK in a project:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <!-- omitted for brevity -->
</Project>
```

Features enabled by using the Web SDK:

- Projects targeting .NET Core 3.0 or later implicitly reference:
  - The ASP.NET Core shared framework.
  - Analyzers designed for building ASP.NET Core apps.
- The Web SDK imports MSBuild targets that enable the use of publish profiles and publishing using WebDeploy.

Properties

PROPERTY	DESCRIPTION
`DisableImplicitFrameworkReferences`	Disables implicit reference to the `Microsoft.AspNetCore.App` shared framework.
`DisableImplicitAspNetCoreAnalyzers`	Disables implicit reference to ASP.NET Core analyzers.
`DisableImplicitComponentsAnalyzers`	Disables implicit reference to Razor Components analyzers when building Blazor (server) applications.

dotnet aspnet-codegenerator

9/22/2020 • 4 minutes to read • Edit Online

By Rick Anderson

`dotnet aspnet-codegenerator` - Runs the ASP.NET Core scaffolding engine. `dotnet aspnet-codegenerator` is only required to scaffold from the command line, it's not needed to use scaffolding with Visual Studio.

This article applies to .NET Core 2.1 SDK and later.

Installing aspnet-codegenerator

`dotnet-aspnet-codegenerator` is a global tool that must be installed. The following command installs the latest stable version of the `dotnet-aspnet-codegenerator` tool:

```
dotnet tool install -g dotnet-aspnet-codegenerator
```

The following command updates `dotnet-aspnet-codegenerator` to the latest stable version available from the installed .NET Core SDKs:

```
dotnet tool update -g dotnet-aspnet-codegenerator
```

Synopsis

```
dotnet aspnet-codegenerator [arguments] [-p|--project] [-n|--nuget-package-dir] [-c|--configuration] [-
tfm|--target-framework] [-b|--build-base-path] [--no-build]
dotnet aspnet-codegenerator [-h|--help]
```

Description

The `dotnet aspnet-codegenerator` global command runs the ASP.NET Core code generator and scaffolding engine.

Arguments

`generator`

The code generator to run. The following generators are available:

GENERATOR	OPERATION
area	Scaffolds an Area
controller	Scaffolds a controller
identity	Scaffolds Identity
razorpage	Scaffolds Razor Pages

GENERATOR	OPERATION
view	Scaffolds a view

Options

`-n|--nuget-package-dir`

Specifies the NuGet package directory.

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`.

`-tfm|--target-framework`

Target Framework to use. For example, `net46`.

`-b|--build-base-path`

The build base path.

`-h|--help`

Prints out a short help for the command.

`--no-build`

Doesn't build the project before running. It also implicitly sets the `--no-restore` flag.

`-p|--project <PATH>`

Specifies the path of the project file to run (folder name or full path). If not specified, it defaults to the current directory.

Generator options

The following sections detail the options available for the supported generators:

- Area
- Controller
- Identity
- Razorpage
- View

Area options

This tool is intended for ASP.NET Core web projects with controllers and views. It's not intended for Razor Pages apps.

Usage: `dotnet aspnet-codegenerator area AreaNameToGenerate`

The preceding command generates the following folders:

- *Areas*
  - *AreaNameToGenerate*
    - *Controllers*
    - *Data*
    - *Models*

- *Views*

Controller options

The following table lists options for `aspnet-codegenerator` `controller` and `razorpage`:

OPTION	DESCRIPTION
--model or -m	Model class to use.
--dataContext or -dc	The `DbContext` class to use.
--bootstrapVersion or -b	Specifies the bootstrap version. Valid values are `3` or `4`. Default is `4`. If needed and not present, a *wwwroot* directory is created that includes the bootstrap files of the specified version.
--referenceScriptLibraries or -scripts	Reference script libraries in the generated views. Adds `_ValidationScriptsPartial` to Edit and Create pages.
--layout or -l	Custom Layout page to use.
--useDefaultLayout or -udl	Use the default layout for the views.
--force or -f	Overwrite existing files.
--relativeFolderPath or -outDir	The relative output folder path from project where the file are generated. If not specified, files are generated in the project folder.

The following table lists options unique to `aspnet-codegenerator controller`:

OPTION	DESCRIPTION
--controllerName or -name	Name of the controller.
--useAsyncActions or -async	Generate async controller actions.
--noViews or -nv	Generate **no** views.
--restWithNoViews or -api	Generate a Controller with REST style API. `noViews` is assumed and any view related options are ignored.
--readWriteActions or -actions	Generate controller with read/write actions without a model.

Use the `-h` switch for help on the `aspnet-codegenerator controller` command:

```
dotnet aspnet-codegenerator controller -h
```

See Scaffold the movie model for an example of `dotnet aspnet-codegenerator controller`.

Razorpage

Razor Pages can be individually scaffolded by specifying the name of the new page and the template to use. The supported templates are:

- `Empty`
- `Create`
- `Edit`
- `Delete`
- `Details`
- `List`

For example, the following command uses the Edit template to generate *MyEdit.cshtml* and *MyEdit.cshtml.cs*:

```
dotnet aspnet-codegenerator razorpage MyEdit Edit -m Movie -dc RazorPagesMovieContext -outDir
Pages/Movies
```

Typically, the template and generated file name is not specified, and the following templates are created:

- `Create`
- `Edit`
- `Delete`
- `Details`
- `List`

The following table lists options for `aspnet-codegenerator` `razorpage` and `controller`:

OPTION	DESCRIPTION
--model or -m	Model class to use.
--dataContext or -dc	The `DbContext` class to use.
--bootstrapVersion or -b	Specifies the bootstrap version. Valid values are `3` or `4`. Default is `4`. If needed and not present, a *wwwroot* directory is created that includes the bootstrap files of the specified version.
--referenceScriptLibraries or -scripts	Reference script libraries in the generated views. Adds `_ValidationScriptsPartial` to Edit and Create pages.
--layout or -l	Custom Layout page to use.
--useDefaultLayout or -udl	Use the default layout for the views.
--force or -f	Overwrite existing files.
--relativeFolderPath or -outDir	The relative output folder path from project where the file are generated. If not specified, files are generated in the project folder.

The following table lists options unique to `aspnet-codegenerator razorpage`:

OPTION	DESCRIPTION
--namespaceName or -namespace	The name of the namespace to use for the generated PageModel

OPTION	DESCRIPTION
--partialView or -partial	Generate a partial view. Layout options -l and -udl are ignored if this is specified.
--noPageModel or -npm	Switch to not generate a PageModel class for Empty template

Use the `-h` switch for help on the `aspnet-codegenerator razorpage` command:

```
dotnet aspnet-codegenerator razorpage -h
```

See Scaffold the movie model for an example of `dotnet aspnet-codegenerator razorpage`.

Identity

See Scaffold Identity

Create web APIs with ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Scott Addie and Tom Dykstra

ASP.NET Core supports creating RESTful services, also known as web APIs, using C#. To handle requests, a web API uses controllers. *Controllers* in a web API are classes that derive from `ControllerBase`. This article shows how to use controllers for handling web API requests.

View or download sample code. (How to download).

ControllerBase class

A web API consists of one or more controller classes that derive from ControllerBase. The web API project template provides a starter controller:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
```

Don't create a web API controller by deriving from the Controller class. `Controller` derives from `ControllerBase` and adds support for views, so it's for handling web pages, not web API requests. There's an exception to this rule: if you plan to use the same controller for both views and web APIs, derive it from `Controller`.

The `ControllerBase` class provides many properties and methods that are useful for handling HTTP requests. For example, `ControllerBase.CreatedAtAction` returns a 201 status code:

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
             _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

Here are some more examples of methods that `ControllerBase` provides.

METHOD	NOTES
BadRequest	Returns 400 status code.
NotFound	Returns 404 status code.

METHOD	NOTES
PhysicalFile	Returns a file.
TryUpdateModelAsync	Invokes model binding.
TryValidateModel	Invokes model validation.

For a list of all available methods and properties, see ControllerBase.

Attributes

The Microsoft.AspNetCore.Mvc namespace provides attributes that can be used to configure the behavior of web API controllers and action methods. The following example uses attributes to specify the supported HTTP action verb and any known HTTP status codes that could be returned:

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
             _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

Here are some more examples of attributes that are available.

ATTRIBUTE	NOTES
`[Route]`	Specifies URL pattern for a controller or action.
`[Bind]`	Specifies prefix and properties to include for model binding.
`[HttpGet]`	Identifies an action that supports the HTTP GET action verb.
`[Consumes]`	Specifies data types that an action accepts.
`[Produces]`	Specifies data types that an action returns.

For a list that includes the available attributes, see the Microsoft.AspNetCore.Mvc namespace.

ApiController attribute

The `[ApiController]` attribute can be applied to a controller class to enable the following opinionated, API-specific behaviors:

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference

- Problem details for error status codes

The *Problem details for error status codes* feature requires a compatibility version of 2.2 or later. The other features require a compatibility version of 2.1 or later.

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference

These features require a compatibility version of 2.1 or later.

Attribute on specific controllers

The `[ApiController]` attribute can be applied to specific controllers, as in the following example from the project template:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
```

Attribute on multiple controllers

One approach to using the attribute on more than one controller is to create a custom base controller class annotated with the `[ApiController]` attribute. The following example shows a custom base class and a controller that derives from it:

```
[ApiController]
public class MyControllerBase : ControllerBase
{
}
```

```
[Produces(MediaTypeNames.Application.Json)]
[Route("[controller]")]
public class PetsController : MyControllerBase
```

```
[Produces(MediaTypeNames.Application.Json)]
[Route("api/[controller]")]
public class PetsController : MyControllerBase
```

Attribute on an assembly

If compatibility version is set to 2.2 or later, the `[ApiController]` attribute can be applied to an assembly. Annotation in this manner applies web API behavior to all controllers in the assembly. There's no way to opt out for individual controllers. Apply the assembly-level attribute to the namespace declaration surrounding the `Startup` class:

```
[assembly: ApiController]
namespace WebApiSample
{
    public class Startup
    {
        ...
    }
}
```

Attribute routing requirement

The `[ApiController]` attribute makes attribute routing a requirement. For example:

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Actions are inaccessible via conventional routes defined by `UseEndpoints`, UseMvc, or UseMvcWithDefaultRoute in `Startup.Configure`.

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
```

Actions are inaccessible via conventional routes defined by UseMvc or UseMvcWithDefaultRoute in `Startup.Configure`.

Automatic HTTP 400 responses

The `[ApiController]` attribute makes model validation errors automatically trigger an HTTP 400 response. Consequently, the following code is unnecessary in an action method:

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

ASP.NET Core MVC uses the ModelStateInvalidFilter action filter to do the preceding check.

Default BadRequest response

With a compatibility version of 2.1, the default response type for an HTTP 400 response is SerializableError. The following request body is an example of the serialized type:

```
{
  "": [
    "A non-empty request body is required."
  ]
}
```

With a compatibility version of 2.2 or later, the default response type for an HTTP 400 response is ValidationProblemDetails. The following request body is an example of the serialized type:

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "|7fb5e16a-4c8f23bbfc974667.",
  "errors": {
    "": [
      "A non-empty request body is required."
    ]
  }
}
```

The `ValidationProblemDetails` type:

- Provides a machine-readable format for specifying errors in web API responses.
- Complies with the RFC 7807 specification.

To make automatic and custom responses consistent, call the ValidationProblem method instead of BadRequest. `ValidationProblem` returns a ValidationProblemDetails object as well as the automatic response.

Log automatic 400 responses

See How to log automatic 400 responses on model validation errors (dotnet/AspNetCore.Docs#12157).

Disable automatic 400 response

To disable the automatic 400 behavior, set the SuppressModelStateInvalidFilter property to `true`. Add the following highlighted code in `Startup.ConfigureServices`:

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressConsumesConstraintForFormFileParameters = true;
    options.SuppressInferBindingSourcesForParameters = true;
    options.SuppressModelStateInvalidFilter = true;
});
```

Binding source parameter inference

A binding source attribute defines the location at which an action parameter's value is found. The following binding source attributes exist:

ATTRIBUTE	BINDING SOURCE
`[FromBody]`	Request body
`[FromForm]`	Form data in the request body
`[FromHeader]`	Request header
`[FromQuery]`	Request query string parameter
`[FromRoute]`	Route data from the current request
`[FromServices]`	The request service injected as an action parameter

WARNING

Don't use `[FromRoute]` when values might contain `%2f` (that is `/`). `%2f` won't be unescaped to `/`. Use `[FromQuery]` if the value might contain `%2f`.

Without the `[ApiController]` attribute or binding source attributes like `[FromQuery]`, the ASP.NET Core runtime attempts to use the complex object model binder. The complex object model binder pulls data from value providers in a defined order.

In the following example, the `[FromQuery]` attribute indicates that the `discontinuedOnly` parameter value is provided in the request URL's query string:

```
[HttpGet]
public ActionResult<List<Product>> Get(
    [FromQuery] bool discontinuedOnly = false)
{
    List<Product> products = null;

    if (discontinuedOnly)
    {
        products = _productsInMemoryStore.Where(p => p.IsDiscontinued).ToList();
    }
    else
    {
        products = _productsInMemoryStore;
    }

    return products;
}
```

The `[ApiController]` attribute applies inference rules for the default data sources of action parameters. These rules save you from having to identify binding sources manually by applying attributes to the action parameters. The binding source inference rules behave as follows:

- `[FromBody]` is inferred for complex type parameters. An exception to the `[FromBody]` inference rule is any complex, built-in type with a special meaning, such as IFormCollection and CancellationToken. The binding source inference code ignores those special types.

- `[FromForm]` is inferred for action parameters of type IFormFile and IFormFileCollection. It's not inferred for any simple or user-defined types.
- `[FromRoute]` is inferred for any action parameter name matching a parameter in the route template. When more than one route matches an action parameter, any route value is considered `[FromRoute]`.
- `[FromQuery]` is inferred for any other action parameters.

FromBody inference notes

`[FromBody]` isn't inferred for simple types such as `string` or `int`. Therefore, the `[FromBody]` attribute should be used for simple types when that functionality is needed.

When an action has more than one parameter bound from the request body, an exception is thrown. For example, all of the following action method signatures cause an exception:

- `[FromBody]` inferred on both because they're complex types.

```
[HttpPost]
public IActionResult Action1(Product product, Order order)
```

- `[FromBody]` attribute on one, inferred on the other because it's a complex type.

```
[HttpPost]
public IActionResult Action2(Product product, [FromBody] Order order)
```

- `[FromBody]` attribute on both.

```
[HttpPost]
public IActionResult Action3([FromBody] Product product, [FromBody] Order order)
```

> **NOTE**
> In ASP.NET Core 2.1, collection type parameters such as lists and arrays are incorrectly inferred as `[FromQuery]`. The `[FromBody]` attribute should be used for these parameters if they are to be bound from the request body. This behavior is corrected in ASP.NET Core 2.2 or later, where collection type parameters are inferred to be bound from the body by default.

Disable inference rules

To disable binding source inference, set SuppressInferBindingSourcesForParameters to `true`. Add the following code in `Startup.ConfigureServices`:

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressConsumesConstraintForFormFileParameters = true;
    options.SuppressInferBindingSourcesForParameters = true;
    options.SuppressModelStateInvalidFilter = true;
});
```

Multipart/form-data request inference

The `[ApiController]` attribute applies an inference rule when an action parameter is annotated with the `[FromForm]` attribute. The `multipart/form-data` request content type is inferred.

To disable the default behavior, set the SuppressConsumesConstraintForFormFileParameters property to `true` in `Startup.ConfigureServices`:

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.Configure<ApiBehaviorOptions>(options =>
{
    options.SuppressConsumesConstraintForFormFileParameters = true;
    options.SuppressInferBindingSourcesForParameters = true;
    options.SuppressModelStateInvalidFilter = true;
});
```

Problem details for error status codes

When the compatibility version is 2.2 or later, MVC transforms an error result (a result with status code 400 or higher) to a result with ProblemDetails. The `ProblemDetails` type is based on the RFC 7807 specification for providing machine-readable error details in an HTTP response.

Consider the following code in a controller action:

```
if (pet == null)
{
    return NotFound();
}
```

The `NotFound` method produces an HTTP 404 status code with a `ProblemDetails` body. For example:

```
{
  type: "https://tools.ietf.org/html/rfc7231#section-6.5.4",
  title: "Not Found",
  status: 404,
  traceId: "0HLHLV31KRN83:00000001"
}
```

Disable ProblemDetails response

The automatic creation of a `ProblemDetails` for error status codes is disabled when the SuppressMapClientErrors property is set to `true`. Add the following code in `Startup.ConfigureServices`:

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

Define supported request content types with the [Consumes] attribute

By default, an action supports all available request content types. For example, if an app is configured to support both JSON and XML input formatters, an action supports multiple content types, including `application/json` and `application/xml`.

The [Consumes] attribute allows an action to limit the supported request content types. Apply the `[Consumes]` attribute to an action or controller, specifying one or more content types:

```
[HttpPost]
[Consumes("application/xml")]
public IActionResult CreateProduct(Product product)
```

In the preceding code, the `CreateProduct` action specifies the content type `application/xml`. Requests routed to this action must specify a `Content-Type` header of `application/xml`. Requests that don't specify a `Content-Type` header of `application/xml` result in a 415 Unsupported Media Type response.

The `[Consumes]` attribute also allows an action to influence its selection based on an incoming request's content type by applying a type constraint. Consider the following example:

```
[ApiController]
[Route("api/[controller]")]
public class ConsumesController : ControllerBase
{
    [HttpPost]
    [Consumes("application/json")]
    public IActionResult PostJson(IEnumerable<int> values) =>
        Ok(new { Consumes = "application/json", Values = values });

    [HttpPost]
    [Consumes("application/x-www-form-urlencoded")]
    public IActionResult PostForm([FromForm] IEnumerable<int> values) =>
        Ok(new { Consumes = "application/x-www-form-urlencoded", Values = values });
}
```

In the preceding code, `ConsumesController` is configured to handle requests sent to the `https://localhost:5001/api/Consumes` URL. Both of the controller's actions, `PostJson` and `PostForm`, handle POST requests with the same URL. Without the `[Consumes]` attribute applying a type constraint, an ambiguous match exception is thrown.

The `[Consumes]` attribute is applied to both actions. The `PostJson` action handles requests sent with a `Content-Type` header of `application/json`. The `PostForm` action handles requests sent with a `Content-Type` header of `application/x-www-form-urlencoded`.

Additional resources

- Controller action return types in ASP.NET Core web API
- Handle errors in ASP.NET Core web APIs
- Custom formatters in ASP.NET Core Web API
- Format response data in ASP.NET Core Web API
- ASP.NET Core Web API help pages with Swagger / OpenAPI
- Routing to controller actions in ASP.NET Core

Tutorial: Create a web API with ASP.NET Core

9/22/2020 • 47 minutes to read • Edit Online

By Rick Anderson, Kirk Larkin, and Mike Wasson

This tutorial teaches the basics of building a web API with ASP.NET Core.

In this tutorial, you learn how to:

- Create a web API project.
- Add a model class and a database context.
- Scaffold a controller with CRUD methods.
- Configure routing, URL paths, and return values.
- Call the web API with Postman.

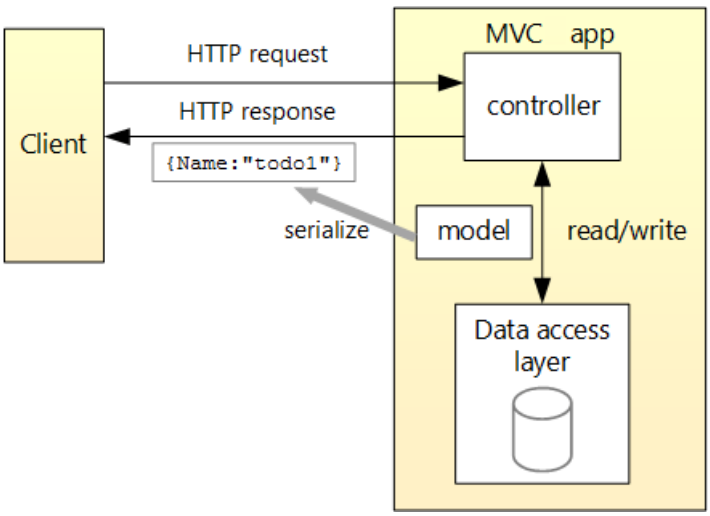
At the end, you have a web API that can manage "to-do" items stored in a database.

Overview

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
`GET /api/TodoItems`	Get all to-do items	None	Array of to-do items
`GET /api/TodoItems/{id}`	Get an item by ID	None	To-do item
`POST /api/TodoItems`	Add a new item	To-do item	To-do item
`PUT /api/TodoItems/{id}`	Update an existing item	To-do item	None
`DELETE /api/TodoItems/{id}`	Delete an item	None	None

The following diagram shows the design of the app.



Prerequisites

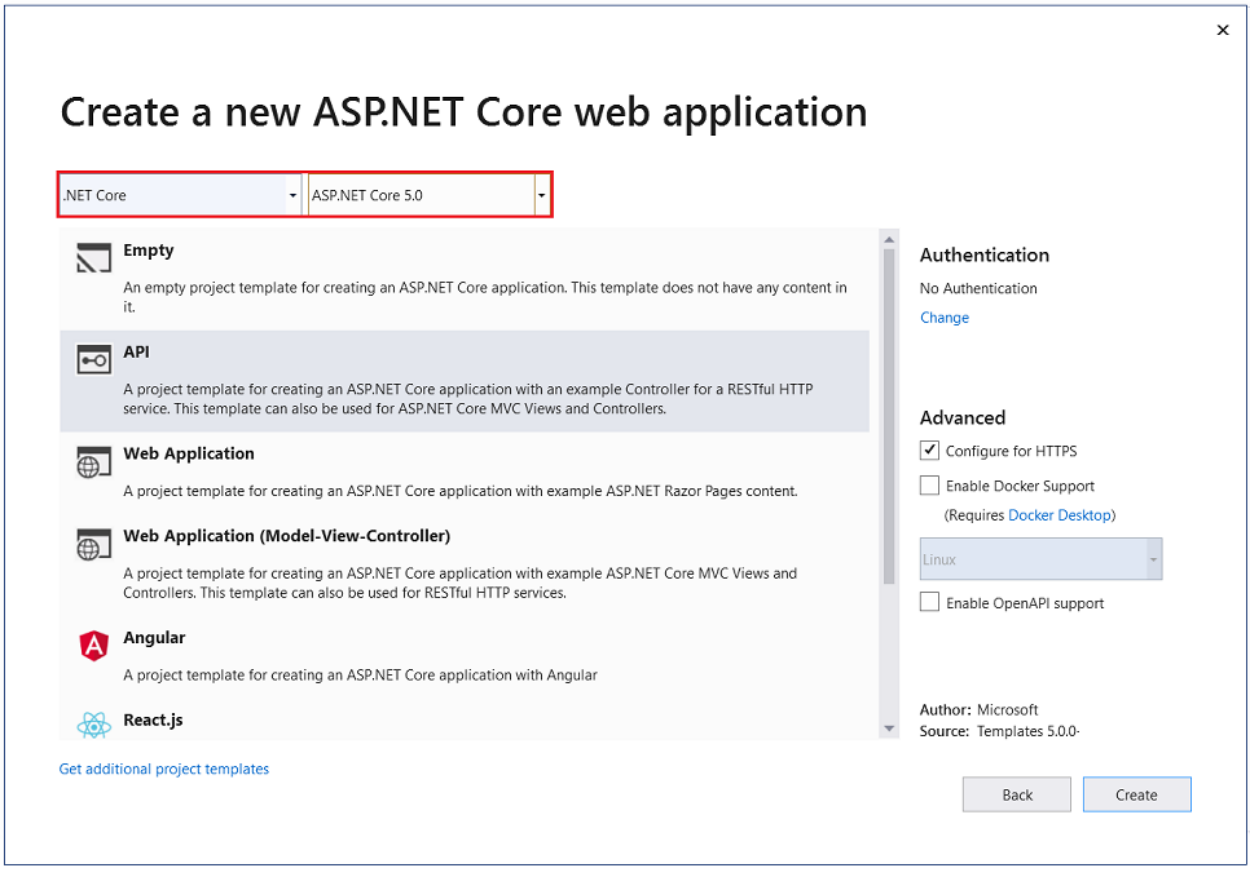
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.8 or later with the **ASP.NET and web development** workload
- .NET 5.0 SDK or later

Create a web project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the **File** menu, select **New** > **Project**.
- Select the **ASP.NET Core Web Application** template and click **Next**.
- Name the project *TodoApi* and click **Create**.
- In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 5.0** are selected. Select the **API** template and click **Create**.



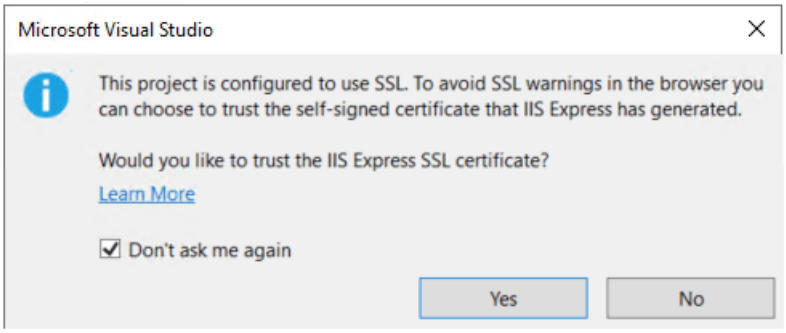
Test the project

The project template creates a `WeatherForecast` API with support for Swagger.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

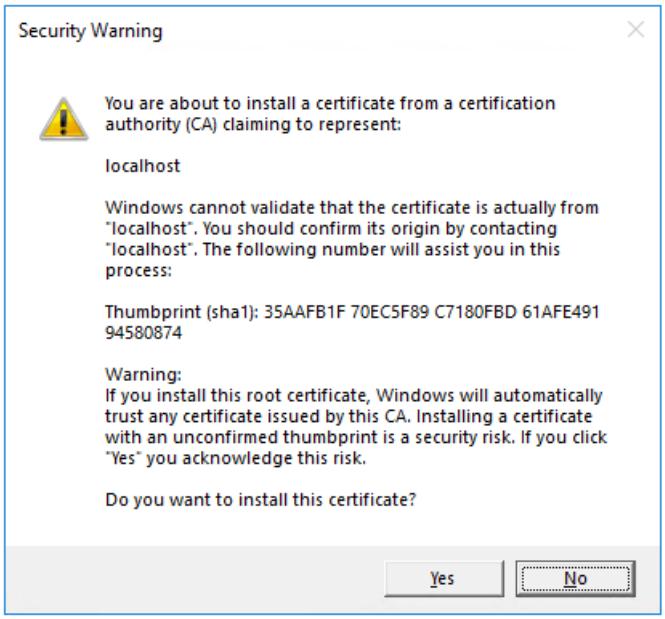
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

Visual Studio launches:

- The IIS Express web server.
- The default browser and navigates to `https://localhost:<port>/https://localhost:5001/swagger/index.html`, where `<port>` is a randomly chosen port number.

The Swagger page `/swagger/index.html` is displayed. Select **GET** > **Try it out** > **Execute**. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop down list box with media types and the example value and schema.

Swagger is used to generate useful documentation and help pages for web APIs. This tutorial focuses on creating a web API. For more information on Swagger, see ASP.NET Core Web API help pages with Swagger / OpenAPI.

Copy and past the **Request URL** in the browser: `https://localhost:<port>/WeatherForecast`

JSON similar to the following is returned:

```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
        "summary": "Chilly"
    }
]
```

Update the launchUrl

In *Properties\launchSettings.json*, update `launchUrl` from `"swagger"` to `"api/TodoItems"`:

```
"launchUrl": "api/TodoItems",
```

Because Swagger has been removed, the preceding markup changes the URL that is launched to the GET method of the controller added in the following sections.

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is a single `TodoItem` class.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.
- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the *Models* folder is used by convention.

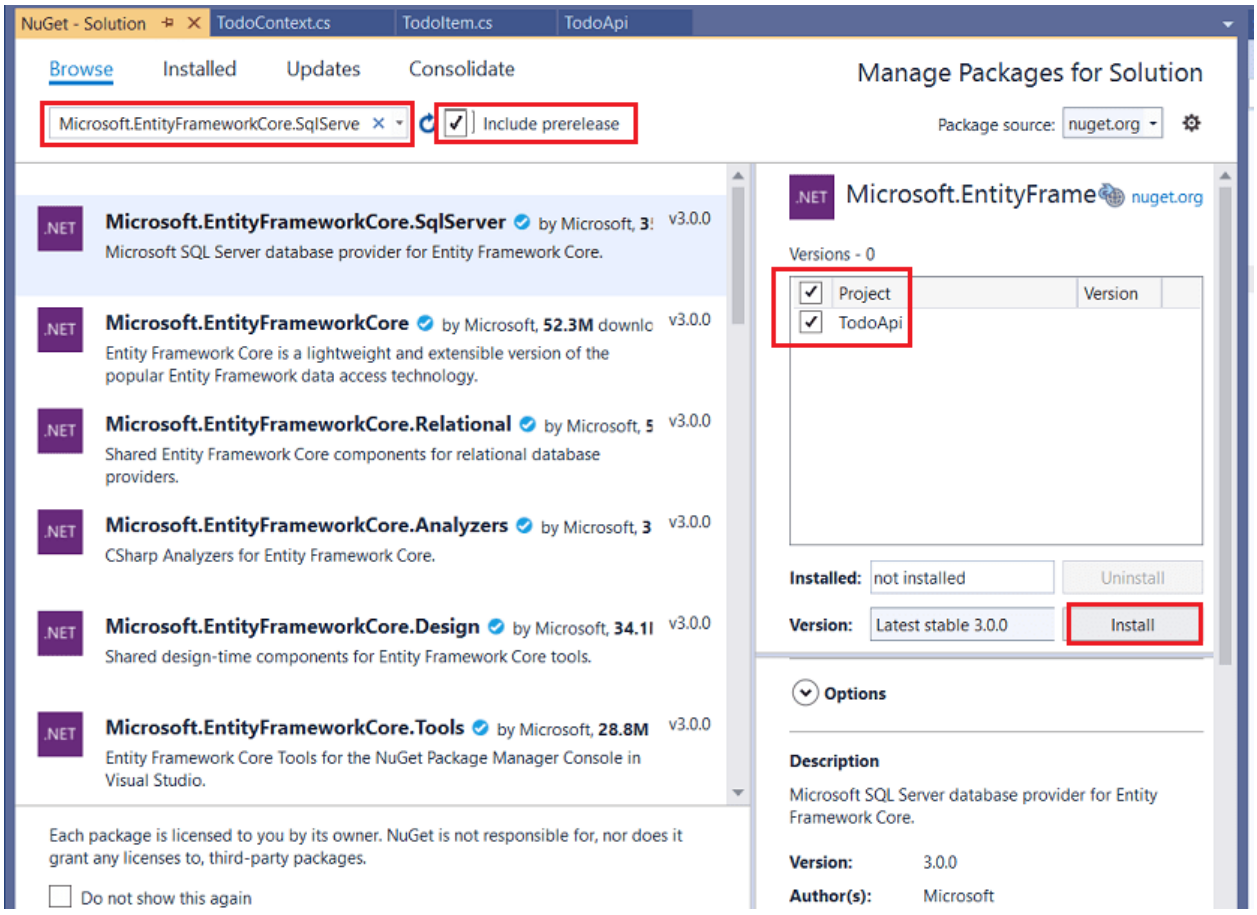
Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add NuGet packages

- From the **Tools** menu, select **NuGet Package Manager > Manage NuGet Packages for Solution**.
- Select the **Browse** tab, and then enter **Microsoft. **EntityFrameworkCore.SqlServer** in the search box.
- Select the **Include prerelease** checkbox so the 5.0 RC version is available.
- Select **Microsoft.EntityFrameworkCore.SqlServer** in the left pane.
- Select the **Project** check box in the right pane and then select **Install**.
- Use the preceding instructions to add the **Microsoft.EntityFrameworkCore.InMemory** NuGet package.



Add the TodoContext database context

- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.
- Enter the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update *Startup.cs* with the following code:

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
                                               opt.UseInMemoryDatabase("TodoList"));
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

The preceding code:

- Removes the Swagger calls.
- Removes unused `using` declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Controllers* folder.
- Select **Add** > **New Scaffolded Item**.

- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

The generated code:

- Marks the class with the `[ApiController]` attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name is excluded from the route. That is, the action's associated method name isn't used in the matching route.

Update the PostTodoItem create method

Replace the return statement in the `PostTodoItem` to use the nameof operator:

```
// POST: api/TodoItems
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
}
```

The preceding code is an HTTP POST method, as indicated by the `[HttpPost]` attribute. The method gets the value of the to-do item from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

Install Postman

This tutorial uses Postman to test the web API.

- Install Postman

- Start the web app.
- Start Postman.
- Disable **SSL certificate verification**
  - From **File** > **Settings** (**General** tab), disable **SSL certificate verification**.

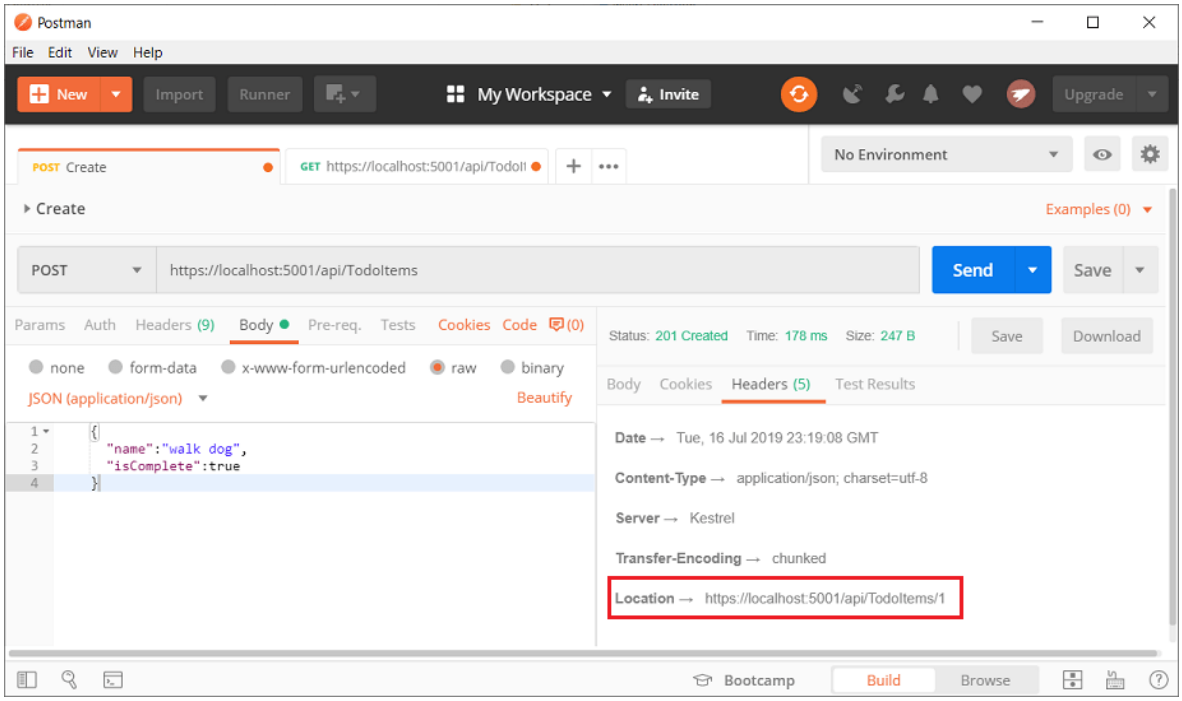
> **WARNING**
>
> Re-enable SSL certificate verification after testing the controller.

Test PostTodoItem with Postman

- Create a new request.
- Set the HTTP method to `POST`.
- Set the URI to `https://localhost:<port>/api/TodoItems`. For example, `https://localhost:5001/api/TodoItems`.
- Select the **Body** tab.
- Select the **raw** radio button.
- Set the type to **JSON (application/json)**.
- In the request body enter JSON for a to-do item:

```
{
  "name":"walk dog",
  "isComplete":true
}
```

- Select **Send**.

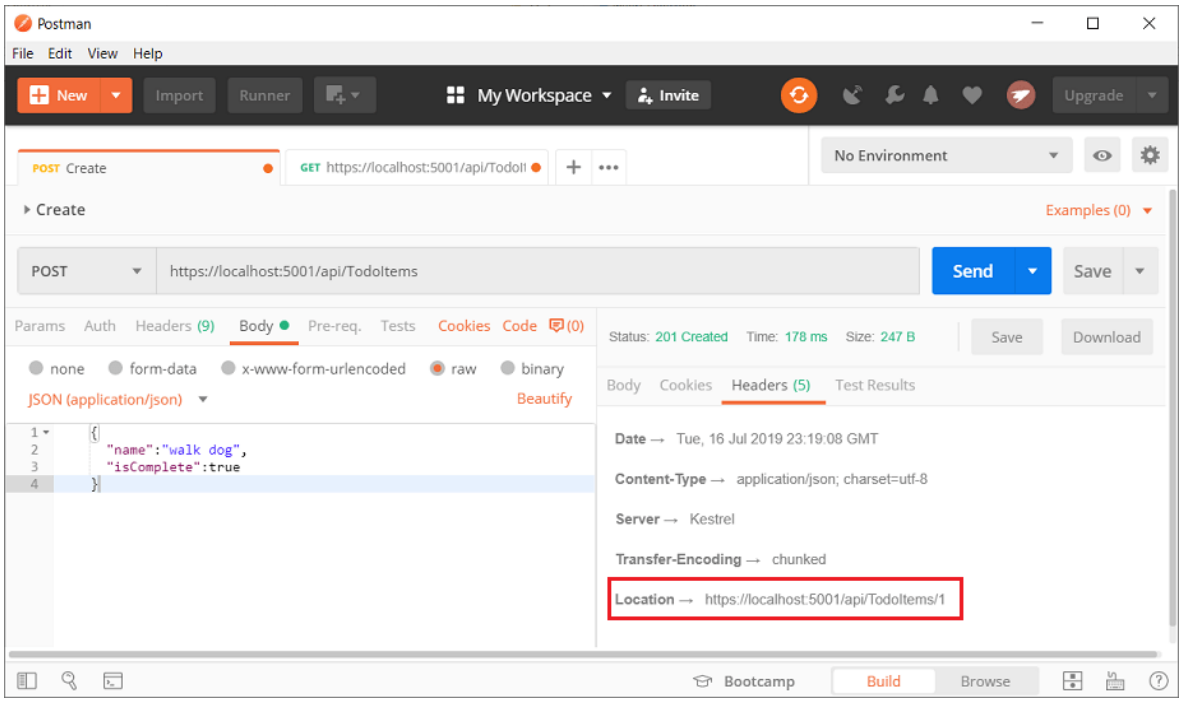


Test the location header URI

The location header URI can be tested in the browser. Copy and paste the location header URI into the browser.

To test in Postman:

- Select the **Headers** tab in the **Response** pane.
- Copy the **Location** header value:



- Set the HTTP method to `GET`.
- Set the URI to `https://localhost:<port>/api/TodoItems/1`. For example, `https://localhost:5001/api/TodoItems/1`.
- Select **Send**.

Examine the GET methods

Two GET endpoints are implemented:

- `GET /api/TodoItems`
- `GET /api/TodoItems/{id}`

Test the app by calling the two endpoints from a browser or Postman. For example:

- `https://localhost:5001/api/TodoItems`
- `https://localhost:5001/api/TodoItems/1`

A response similar to the following is produced by the call to `GetTodoItems`:

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Test Get with Postman

- Create a new request.
- Set the HTTP method to **GET**.
- Set the request URI to `https://localhost:<port>/api/TodoItems`. For example,

`https://localhost:5001/api/TodoItems`.

- Set **Two pane view** in Postman.
- Select **Send**.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not return any data. If no data is returned, POST data to the app.

Routing and URL paths

The `[HttpGet]` attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
    {
        _context = context;
    }
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.
- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

```
// GET: api/TodoItems/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

The PutTodoItem method

Examine the `PutTodoItem` method:

```
// PUT: api/TodoItems/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses HTTP PUT. The response is 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

If you get an error calling `PutTodoItem`, call `GET` to ensure there's an item in the database.

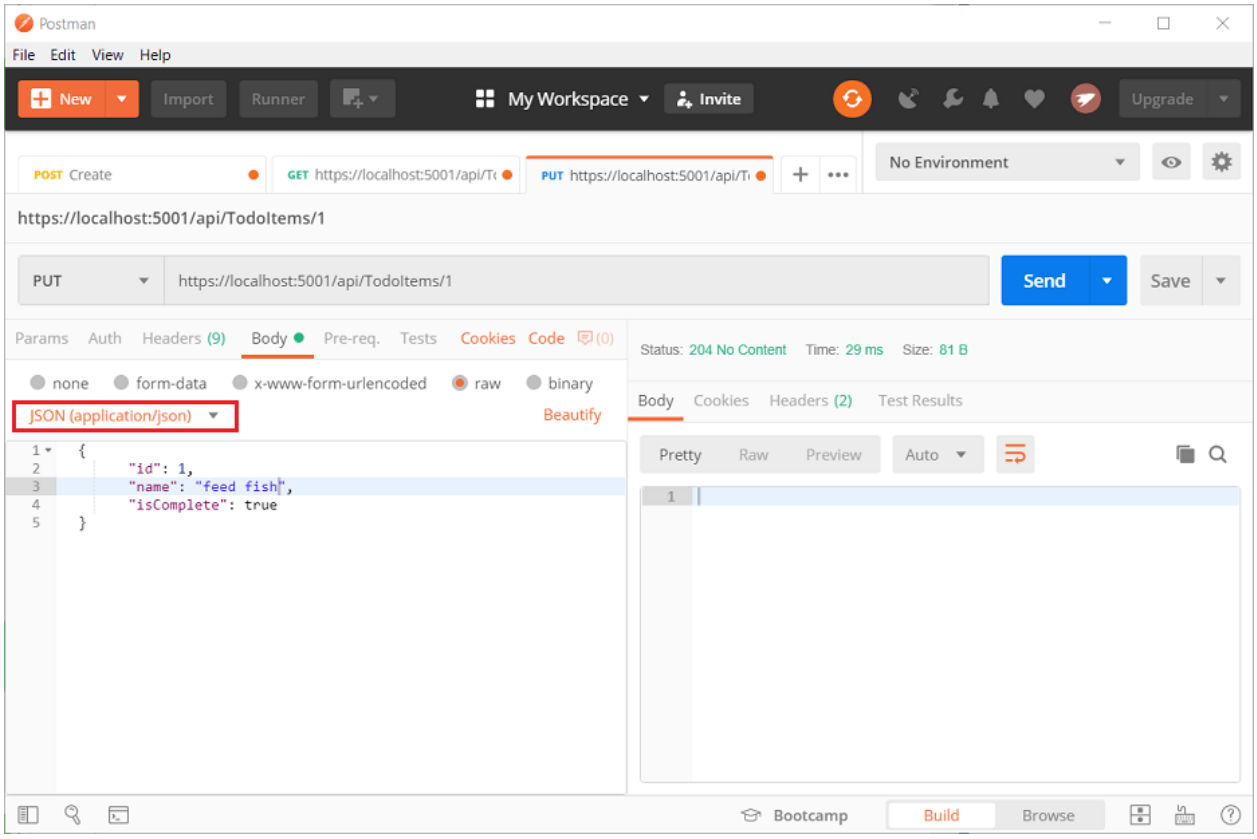
Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to `"feed fish"`:

```
{
  "Id":1,
  "name":"feed fish",
  "isComplete":true
}
```

The following image shows the Postman update:



The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

```
// DELETE: api/TodoItems/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Test the DeleteTodoItem method

Use Postman to delete a to-do item:

- Set the method to `DELETE`.
- Set the URI of the object to delete (for example `https://localhost:5001/api/TodoItems/1`).
- Select **Send**.

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used

in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
        public string Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

```
public class TodoItemDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

```
// GET: api/TodoItems
[HttpGet]
public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
{
    return await _context.TodoItems
        .Select(x => ItemToDTO(x))
        .ToListAsync();
}

[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return ItemToDTO(todoItem);
}

[HttpPut("{id}")]
public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO todoItemDTO)
{
```

```
    if (id != todoItemDTO.Id)
    {
        return BadRequest();
    }

    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    todoItem.Name = todoItemDTO.Name;
    todoItem.IsComplete = todoItemDTO.IsComplete;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
    {
        return NotFound();
    }

    return NoContent();
}

[HttpPost]
public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO todoItemDTO)
{
    var todoItem = new TodoItem
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    return CreatedAtAction(
        nameof(GetTodoItem),
        new { id = todoItem.Id },
        ItemToDTO(todoItem));
}

[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}

private bool TodoItemExists(long id) =>
     _context.TodoItems.Any(e => e.Id == id);

private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
    new TodoItemDTO
    {
        Id = todoItem.Id,
        Name = todoItem.Name,
        IsComplete = todoItem.IsComplete
```

```
    };
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

In this tutorial, you learn how to:

- Create a web API project.
- Add a model class and a database context.
- Scaffold a controller with CRUD methods.
- Configure routing, URL paths, and return values.
- Call the web API with Postman.

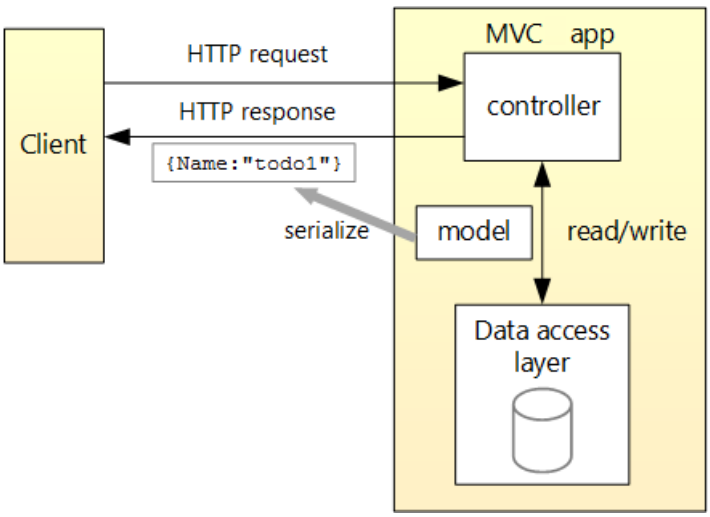
At the end, you have a web API that can manage "to-do" items stored in a database.

Overview

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
`GET /api/TodoItems`	Get all to-do items	None	Array of to-do items
`GET /api/TodoItems/{id}`	Get an item by ID	None	To-do item
`POST /api/TodoItems`	Add a new item	To-do item	To-do item
`PUT /api/TodoItems/{id}`	Update an existing item	To-do item	None
`DELETE /api/TodoItems/{id}`	Delete an item	None	None

The following diagram shows the design of the app.



Prerequisites

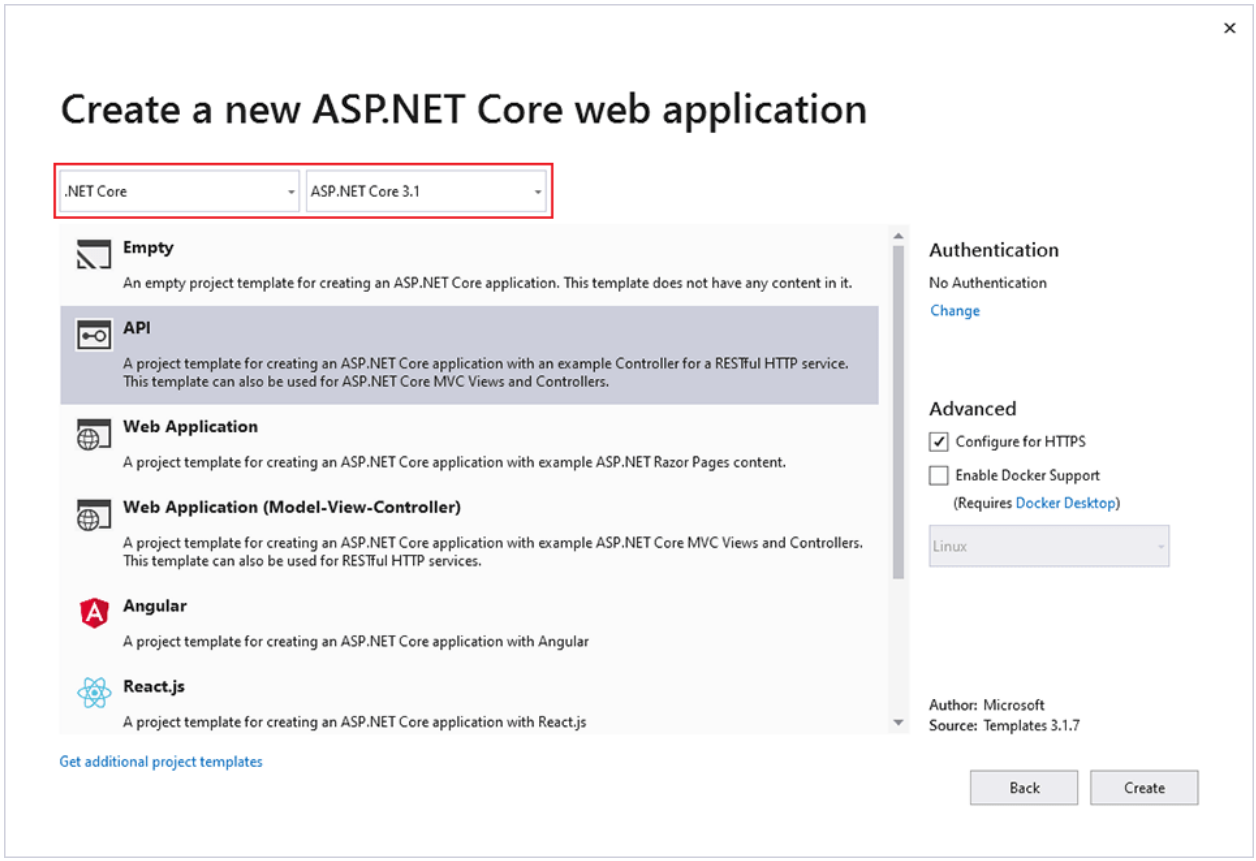
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a web project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the **File** menu, select **New** > **Project**.
- Select the **ASP.NET Core Web Application** template and click **Next**.
- Name the project *TodoApi* and click **Create**.
- In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 3.1** are selected. Select the **API** template and click **Create**.



Test the API

The project template creates a `WeatherForecast` API. Call the `Get` method from a browser to test the app.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Press Ctrl+F5 to run the app. Visual Studio launches a browser and navigates to `https://localhost:<port>/WeatherForecast`, where `<port>` is a randomly chosen port number.

If you get a dialog box that asks if you should trust the IIS Express certificate, select **Yes**. In the **Security Warning** dialog that appears next, select **Yes**.

JSON similar to the following is returned:

```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
        "summary": "Chilly"
    }
]
```

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is a single `TodoItem` class.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.
- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following code:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the *Models* folder is used by convention.

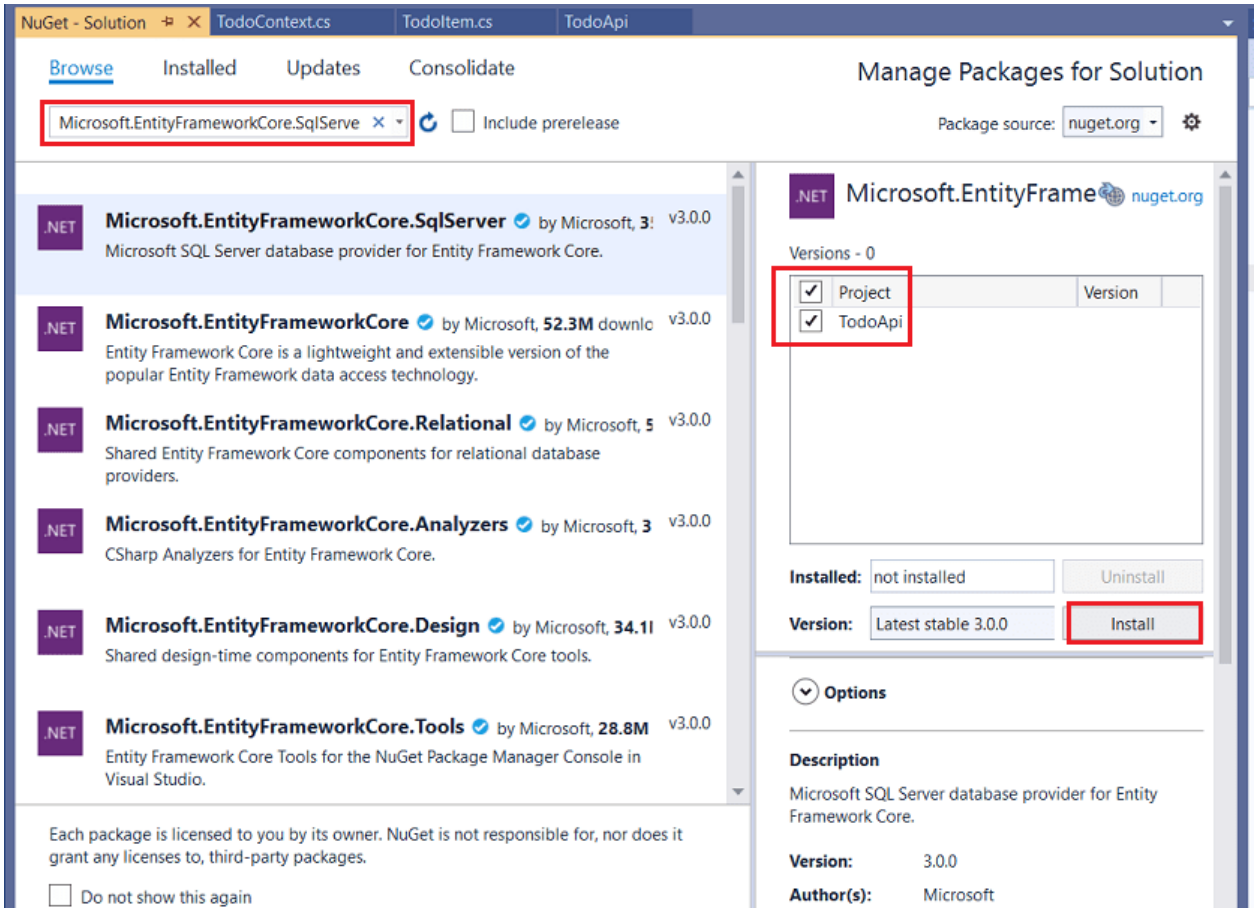
Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

Add NuGet packages

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab, and then enter **Microsoft.EntityFrameworkCore.SqlServer** in the search box.
- Select **Microsoft.EntityFrameworkCore.SqlServer** in the left pane.
- Select the **Project** check box in the right pane and then select **Install**.
- Use the preceding instructions to add the **Microsoft.EntityFrameworkCore.InMemory** NuGet package.



Add the TodoContext database context

- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.
- Enter the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update *Startup.cs* with the following highlighted code:

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
               opt.UseInMemoryDatabase("TodoList"));
            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

The preceding code:

- Removes unused `using` declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Controllers* folder.
- Select **Add** > **New Scaffolded Item**.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.

- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

The generated code:

- Marks the class with the `[ApiController]` attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name is excluded from the route. That is, the action's associated method name isn't used in the matching route.

Examine the PostTodoItem create method

Replace the return statement in the `PostTodoItem` to use the nameof operator:

```
// POST: api/TodoItems
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
}
```

The preceding code is an HTTP POST method, as indicated by the `[HttpPost]` attribute. The method gets the value of the to-do item from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

Install Postman

This tutorial uses Postman to test the web API.

- Install Postman
- Start the web app.
- Start Postman.

- Disable **SSL certificate verification**
  - From **File** > **Settings** (**General** tab), disable **SSL certificate verification**.

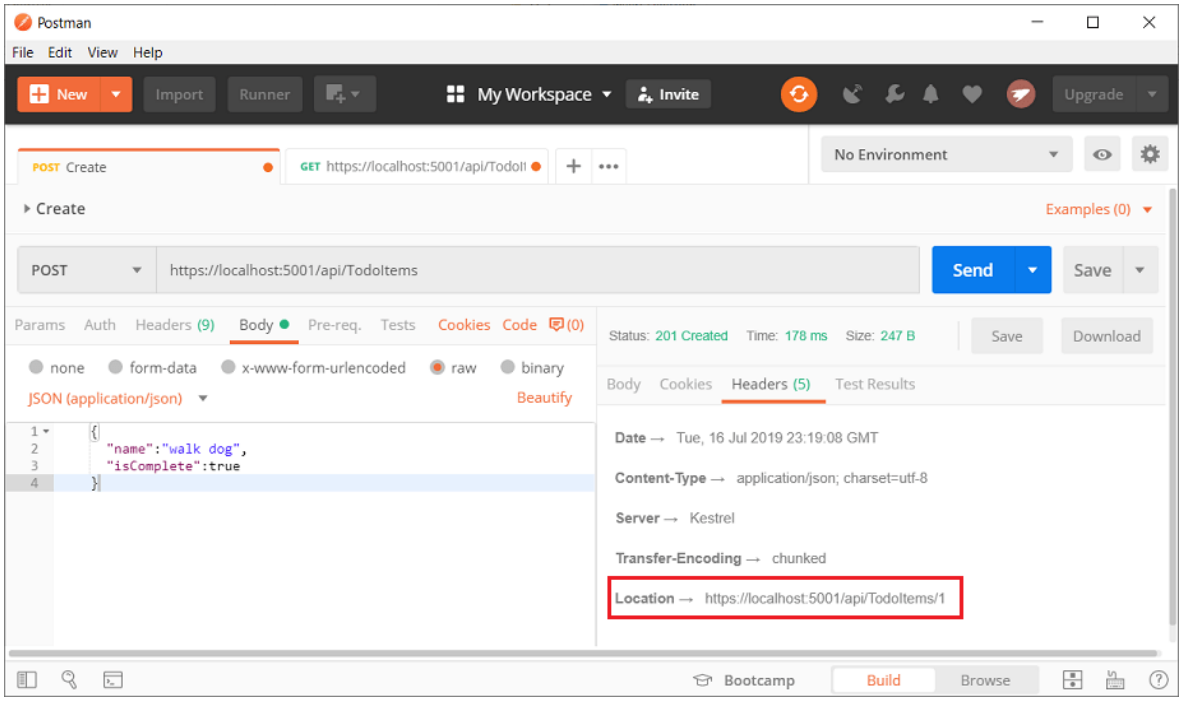
> **WARNING**
> Re-enable SSL certificate verification after testing the controller.

Test PostTodoItem with Postman

- Create a new request.
- Set the HTTP method to `POST`.
- Set the URI to `https://localhost:<port>/api/TodoItems`. For example, `https://localhost:5001/api/TodoItems`.
- Select the **Body** tab.
- Select the **raw** radio button.
- Set the type to **JSON (application/json)**.
- In the request body enter JSON for a to-do item:

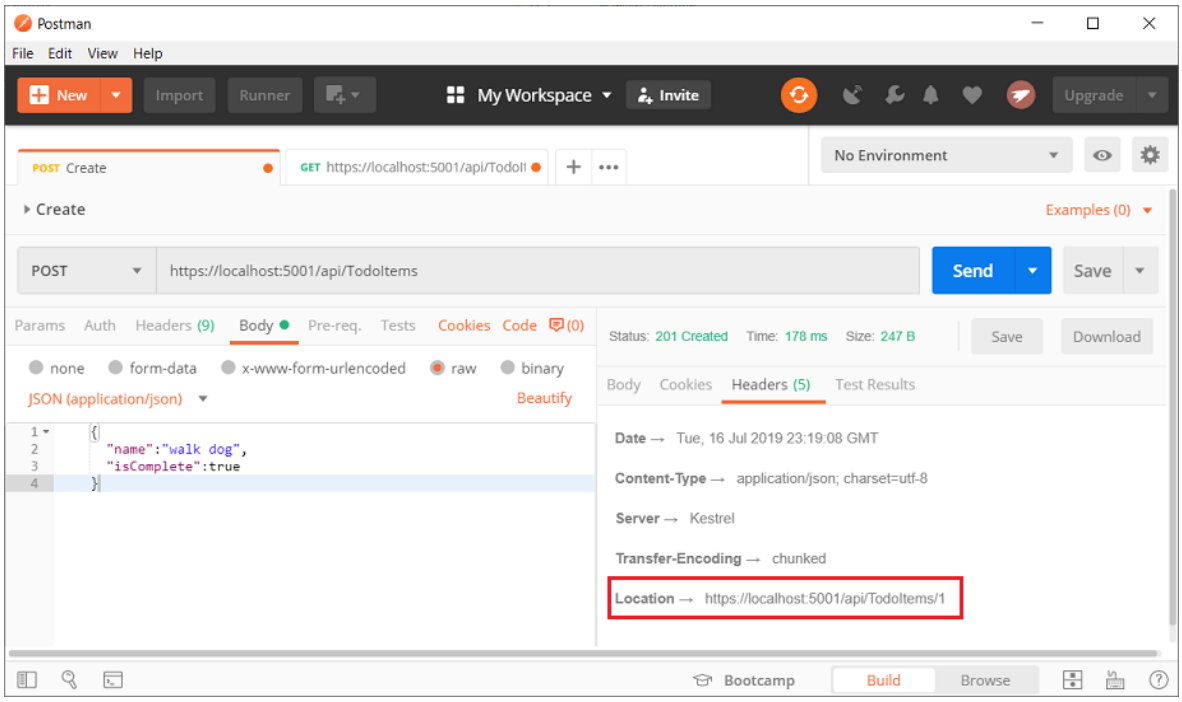
```
{
  "name":"walk dog",
  "isComplete":true
}
```

- Select **Send**.



Test the location header URI with Postman

- Select the **Headers** tab in the **Response** pane.
- Copy the **Location** header value:



- Set the HTTP method to `GET`.
- Set the URI to `https://localhost:<port>/api/TodoItems/1`. For example, `https://localhost:5001/api/TodoItems/1`.
- Select **Send**.

Examine the GET methods

These methods implement two GET endpoints:

- `GET /api/TodoItems`
- `GET /api/TodoItems/{id}`

Test the app by calling the two endpoints from a browser or Postman. For example:

- `https://localhost:5001/api/TodoItems`
- `https://localhost:5001/api/TodoItems/1`

A response similar to the following is produced by the call to `GetTodoItems`:

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Test Get with Postman

- Create a new request.
- Set the HTTP method to **GET**.
- Set the request URI to `https://localhost:<port>/api/TodoItems`. For example, `https://localhost:5001/api/TodoItems`.
- Set **Two pane view** in Postman.
- Select **Send**.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request will not return any data. If no data is returned, POST data to the app.

Routing and URL paths

The `[HttpGet]` attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
    {
        _context = context;
    }
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.
- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

```
// GET: api/TodoItems/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 NotFound error code.

- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

The PutTodoItem method

Examine the `PutTodoItem` method:

```
// PUT: api/TodoItems/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses HTTP PUT. The response is 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

If you get an error calling `PutTodoItem`, call `GET` to ensure there's an item in the database.

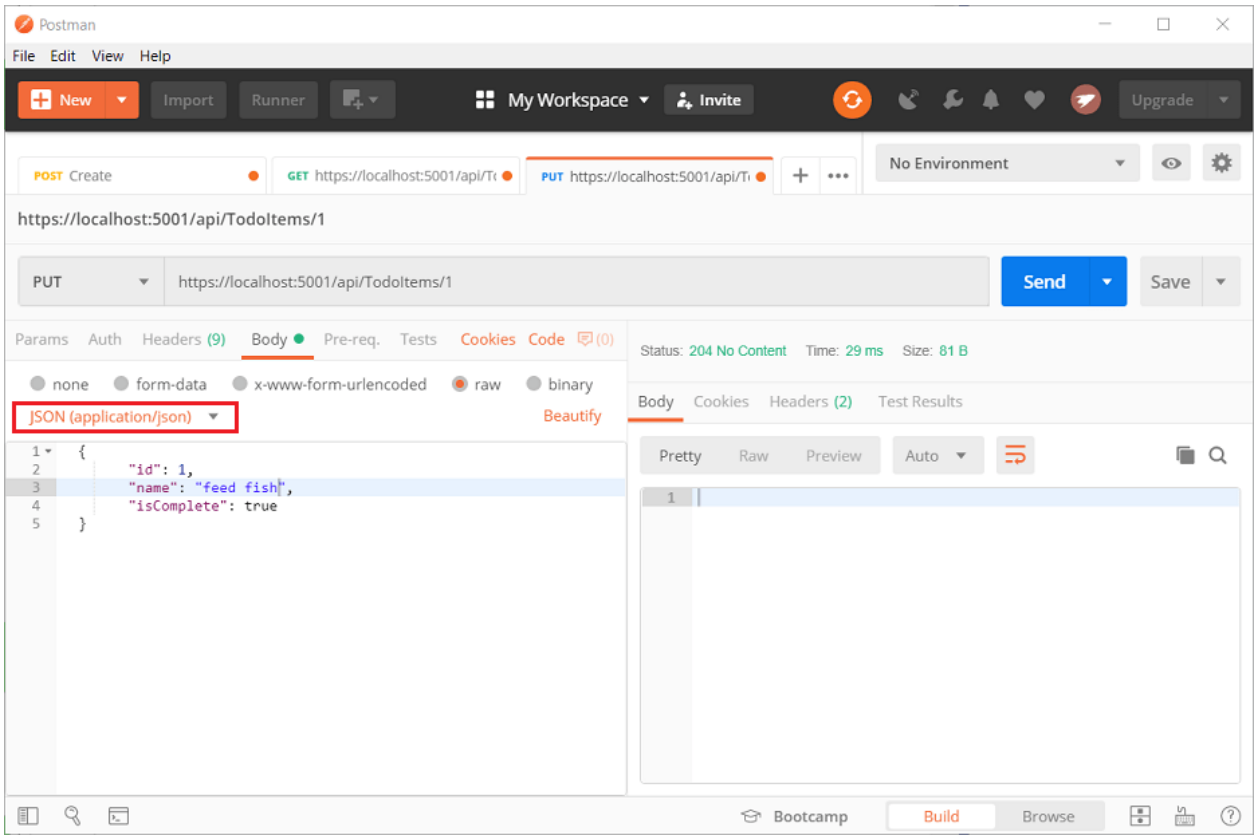
Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to "feed fish":

```
{
  "Id":1,
  "name":"feed fish",
  "isComplete":true
}
```

The following image shows the Postman update:



The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

```
// DELETE: api/TodoItems/5
[HttpDelete("{id}")]
public async Task<ActionResult<TodoItem>> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return todoItem;
}
```

Test the DeleteTodoItem method

Use Postman to delete a to-do item:

- Set the method to `DELETE`.
- Set the URI of the object to delete (for example `https://localhost:5001/api/TodoItems/1`).
- Select **Send**.

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
    public string Secret { get; set; }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

```
public class TodoItemDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

```
    [HttpGet]
    public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
        return await _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO todoItemDTO)
    {
        if (id != todoItemDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
```

```
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoItemDTO.Name;
        todoItem.IsComplete = todoItemDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }

    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO todoItemDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoItemDTO.IsComplete,
            Name = todoItemDTO.Name
        };

        _context.TodoItems.Add(todoItem);
        await _context.SaveChangesAsync();

        return CreatedAtAction(
            nameof(GetTodoItem),
            new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        _context.TodoItems.Remove(todoItem);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool TodoItemExists(long id) =>
         _context.TodoItems.Any(e => e.Id == id);

    private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
        new TodoItemDTO
        {
            Id = todoItem.Id,
            Name = todoItem.Name,
            IsComplete = todoItem.IsComplete
        };
}
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

In this tutorial, you learn how to:

- Create a web API project.
- Add a model class and a database context.
- Add a controller.
- Add CRUD methods.
- Configure routing and URL paths.
- Specify return values.
- Call the web API with Postman.
- Call the web API with JavaScript.

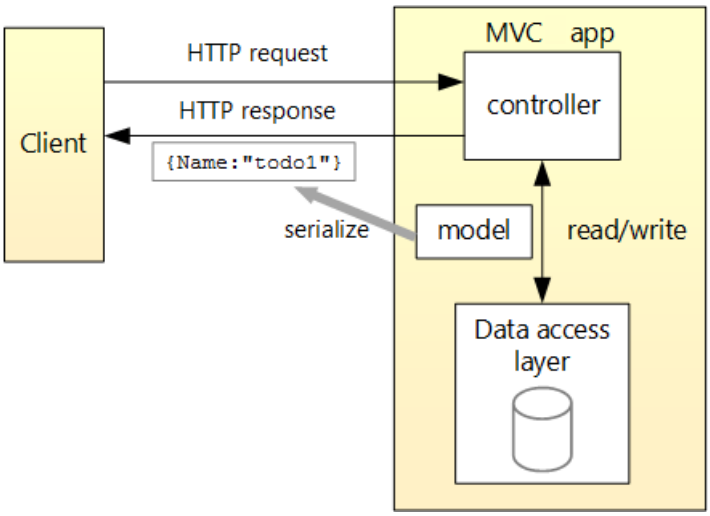
At the end, you have a web API that can manage "to-do" items stored in a relational database.

Overview 2.1

This tutorial creates the following API:

API	DESCRIPTION	REQUEST BODY	RESPONSE BODY
GET /api/TodoItems	Get all to-do items	None	Array of to-do items
GET /api/TodoItems/{id}	Get an item by ID	None	To-do item
POST /api/TodoItems	Add a new item	To-do item	To-do item
PUT /api/TodoItems/{id}	Update an existing item	To-do item	None
DELETE /api/TodoItems/{id}	Delete an item	None	None

The following diagram shows the design of the app.



Prerequisites 2.1

- Visual Studio
- Visual Studio Code

- Visual Studio for Mac

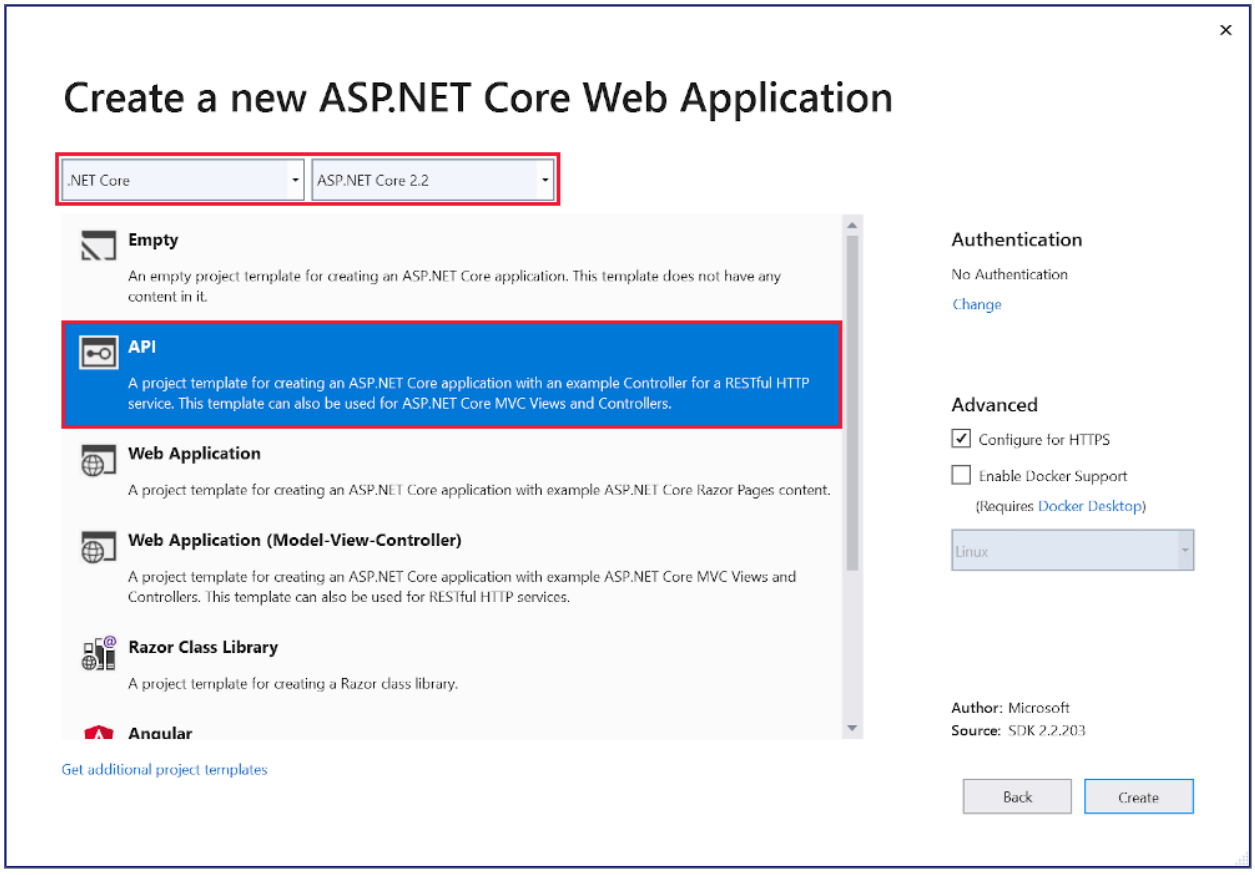
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

Create a web project 2.1

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the **File** menu, select **New** > **Project**.
- Select the **ASP.NET Core Web Application** template and click **Next**.
- Name the project *TodoApi* and click **Create**.
- In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 2.2** are selected. Select the **API** template and click **Create**. **Don't** select **Enable Docker Support**.



Test the API 2.1

The project template creates a `values` API. Call the `Get` method from a browser to test the app.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Press Ctrl+F5 to run the app. Visual Studio launches a browser and navigates to

`https://localhost:<port>/api/values`, where `<port>` is a randomly chosen port number.

If you get a dialog box that asks if you should trust the IIS Express certificate, select **Yes**. In the **Security Warning** dialog that appears next, select **Yes**.

The following JSON is returned:

```
["value1","value2"]
```

Add a model class 2.1

A *model* is a set of classes that represent the data that the app manages. The model for this app is a single `TodoItem` class.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.
- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following code:

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the *Models* folder is used by convention.

Add a database context 2.1

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Models* folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.
- Replace the template code with the following code:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Register the database context 2.1

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update *Startup.cs* with the following highlighted code:

```
// Unused usings removed
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using TodoApi.Models;

namespace TodoApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the
        //container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<TodoContext>(opt =>
                opt.UseInMemoryDatabase("TodoList"));
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        }

        // This method gets called by the runtime. Use this method to configure the HTTP
        //request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                // The default HSTS value is 30 days. You may want to change this for
                // production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseMvc();
        }
    }
}
```

The preceding code:

- Removes unused `using` declarations.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

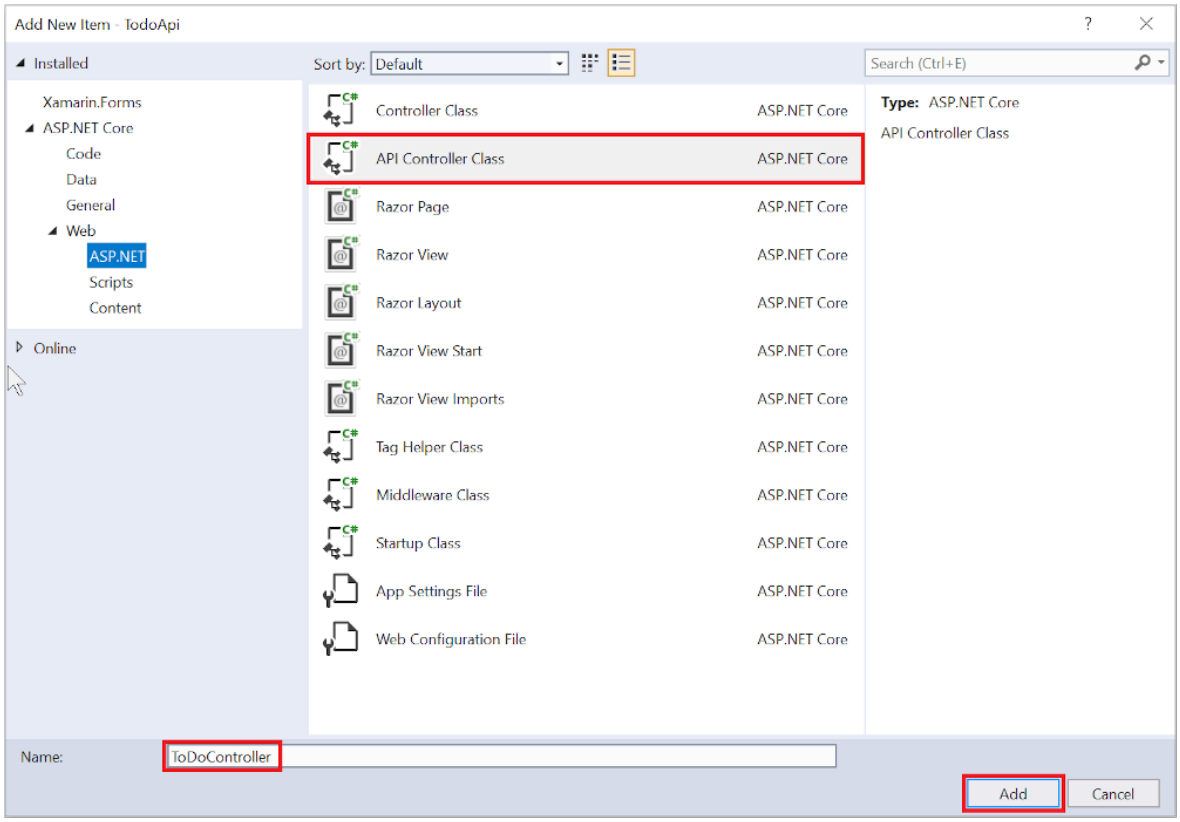
Add a controller 2.1

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Right-click the *Controllers* folder.

- Select **Add** > **New Item**.

- In the **Add New Item** dialog, select the **API Controller Class** template.
- Name the class *TodoController*, and select **Add**.



- Replace the template code with the following code:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TodoApi.Models;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                // Create a new TodoItem if collection is empty,
                // which means you can't delete all TodoItems.
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

The preceding code:

- Defines an API controller class without methods.
- Marks the class with the `[ApiController]` attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.
- Adds an item named `Item1` to the database if the database is empty. This code is in the constructor, so it runs every time there's a new HTTP request. If you delete all items, the constructor creates `Item1` again the next time an API method is called. So it may look like the deletion didn't work when it actually did work.

Add Get methods 2.1

To provide an API that retrieves to-do items, add the following methods to the `TodoController` class:

```
// GET: api/Todo
[HttpGet]
public async Task<ActionResult<IEnumerable<TodoItem>>> GetTodoItems()
{
    return await _context.TodoItems.ToListAsync();
}

// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

These methods implement two GET endpoints:

- `GET /api/todo`
- `GET /api/todo/{id}`

Stop the app if it's still running. Then run it again to include the latest changes.

Test the app by calling the two endpoints from a browser. For example:

- `https://localhost:<port>/api/todo`
- `https://localhost:<port>/api/todo/1`

The following HTTP response is produced by the call to `GetTodoItems` :

```
[
  {
    "id": 1,
    "name": "Item1",
    "isComplete": false
  }
]
```

Routing and URL paths 2.1

The `[HttpGet]` attribute denotes a method that responds to an HTTP GET request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoController : ControllerBase
    {
        private readonly TodoContext _context;
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **Todo**Controller, so the controller name is "todo". ASP.NET Core routing is case insensitive.
- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values 2.1

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

Test the GetTodoItems method 2.1

This tutorial uses Postman to test the web API.

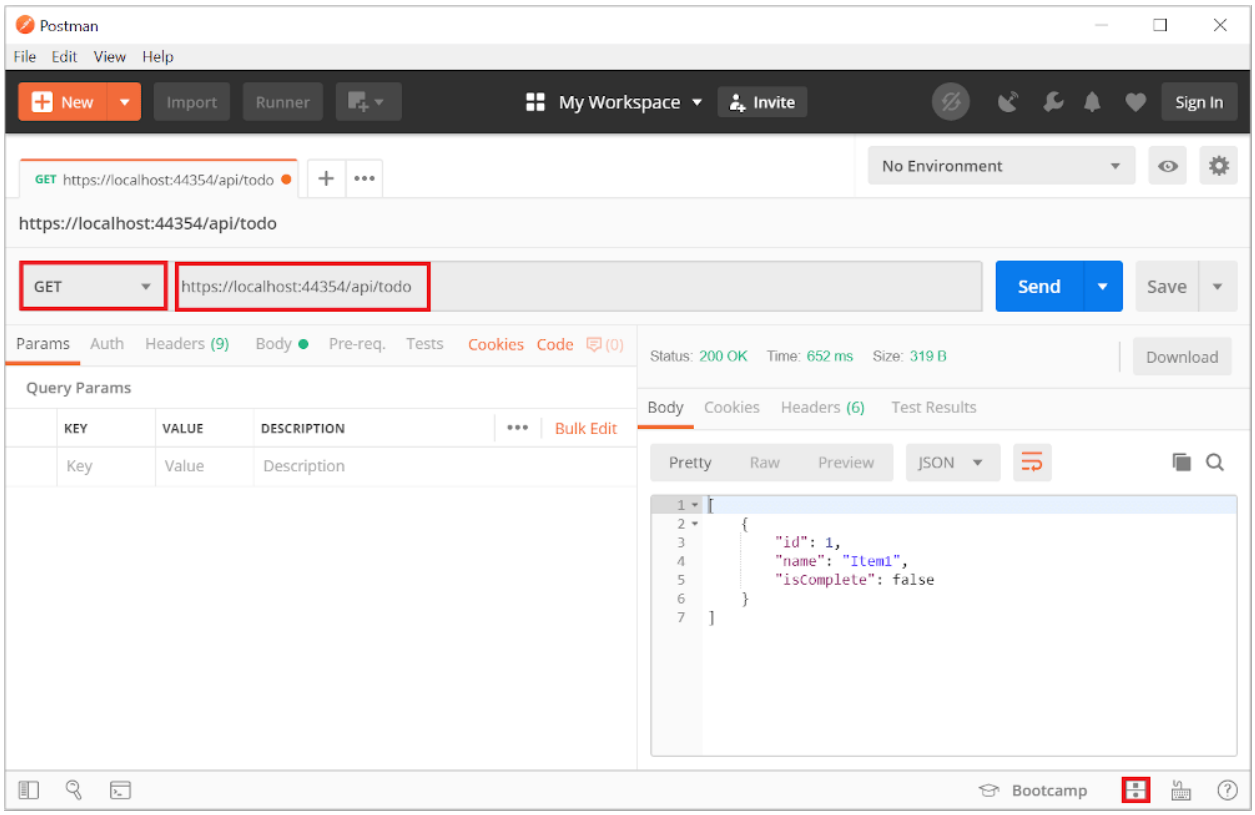
- Install Postman.
- Start the web app.
- Start Postman.
- Disable **SSL certificate verification**.

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- From **File** > **Settings** (**General** tab), disable **SSL certificate verification**.

> **WARNING**
> Re-enable SSL certificate verification after testing the controller.

- Create a new request.
  - Set the HTTP method to **GET**.
  - Set the request URI to `https://localhost:<port>/api/todo`. For example, `https://localhost:5001/api/todo`.
- Set **Two pane view** in Postman.
- Select **Send**.



Add a Create method 2.1

Add the following `PostTodoItem` method inside of *Controllers/TodoController.cs*:

```
// POST: api/Todo
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem item)
{
    _context.TodoItems.Add(item);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(GetTodoItem), new { id = item.Id }, item);
}
```

The preceding code is an HTTP POST method, as indicated by the `[HttpPost]` attribute. The method gets the value of the to-do item from the body of the HTTP request.

The `CreatedAtAction` method:

- Returns an HTTP 201 status code, if successful. HTTP 201 is the standard response for an HTTP POST method that creates a new resource on the server.
- Adds a `Location` header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

```
// GET: api/Todo/5
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

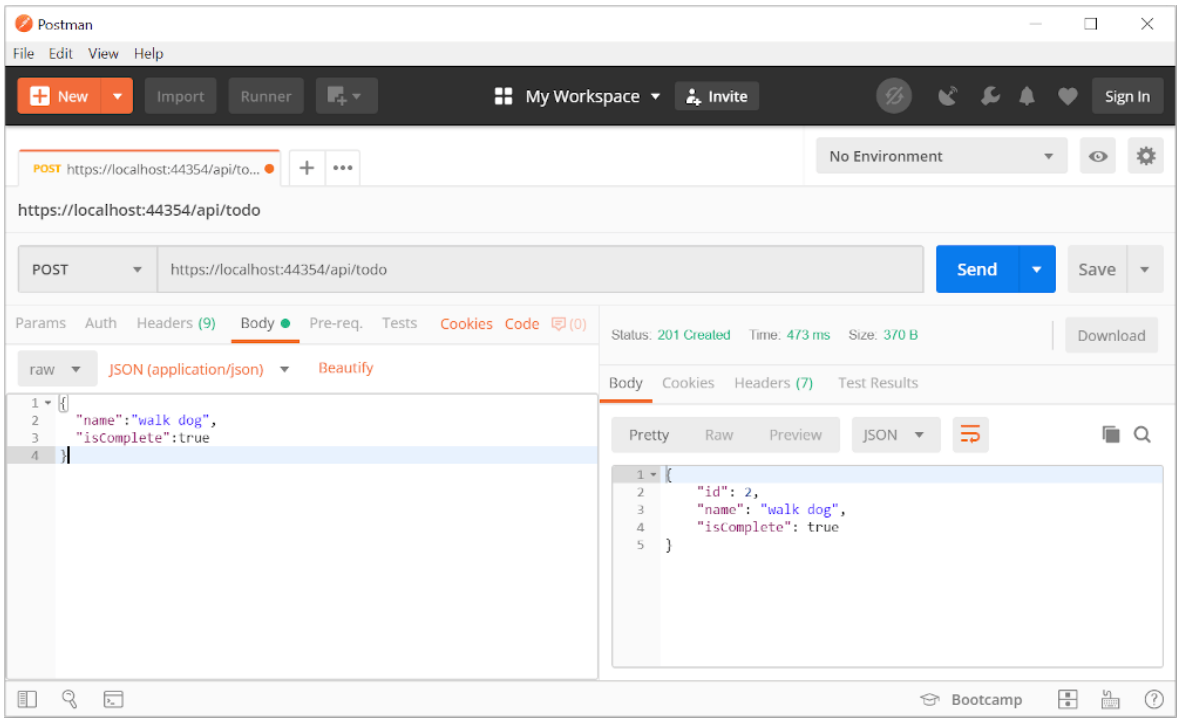
    return todoItem;
}
```

Test the PostTodoItem method 2.1

- Build the project.
- In Postman, set the HTTP method to `POST`.
- Set the URI to `https://localhost:<port>/api/TodoItem`. For example, `https://localhost:5001/api/TodoItem`.
- Select the **Body** tab.
- Select the **raw** radio button.
- Set the type to **JSON (application/json)**.
- In the request body enter JSON for a to-do item:

```
{
  "name":"walk dog",
  "isComplete":true
}
```

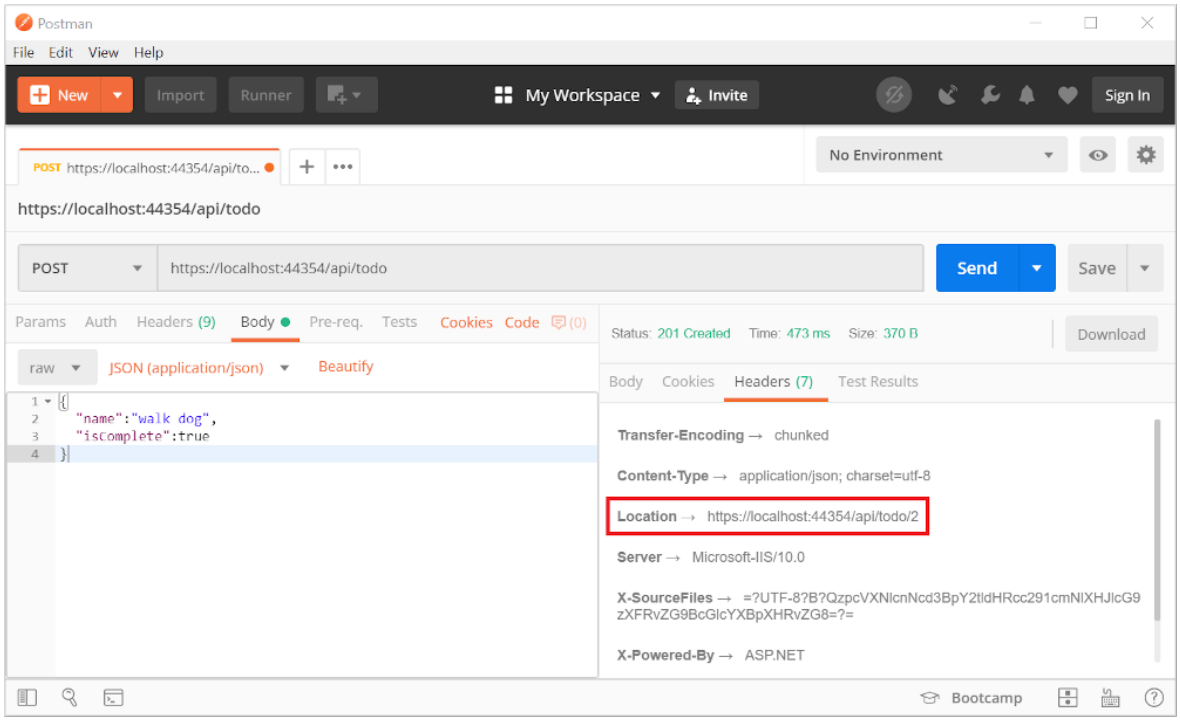
- Select **Send**.



If you get a 405 Method Not Allowed error, it's probably the result of not compiling the project after adding the `PostTodoItem` method.

Test the location header URI 2.1

- Select the **Headers** tab in the **Response** pane.
- Copy the **Location** header value:



- Set the method to GET. * Set the URI to `https://localhost:<port>/api/TodoItems/2`. For example, `https://localhost:5001/api/TodoItems/2`.
- Select **Send**.

Add a PutTodoItem method 2.1

Add the following `PutTodoItem` method:

```
// PUT: api/Todo/5
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem item)
{
    if (id != item.Id)
    {
        return BadRequest();
    }

    _context.Entry(item).State = EntityState.Modified;
    await _context.SaveChangesAsync();

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses HTTP PUT. The response is 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

If you get an error calling `PutTodoItem`, call `GET` to ensure there's an item in the database.

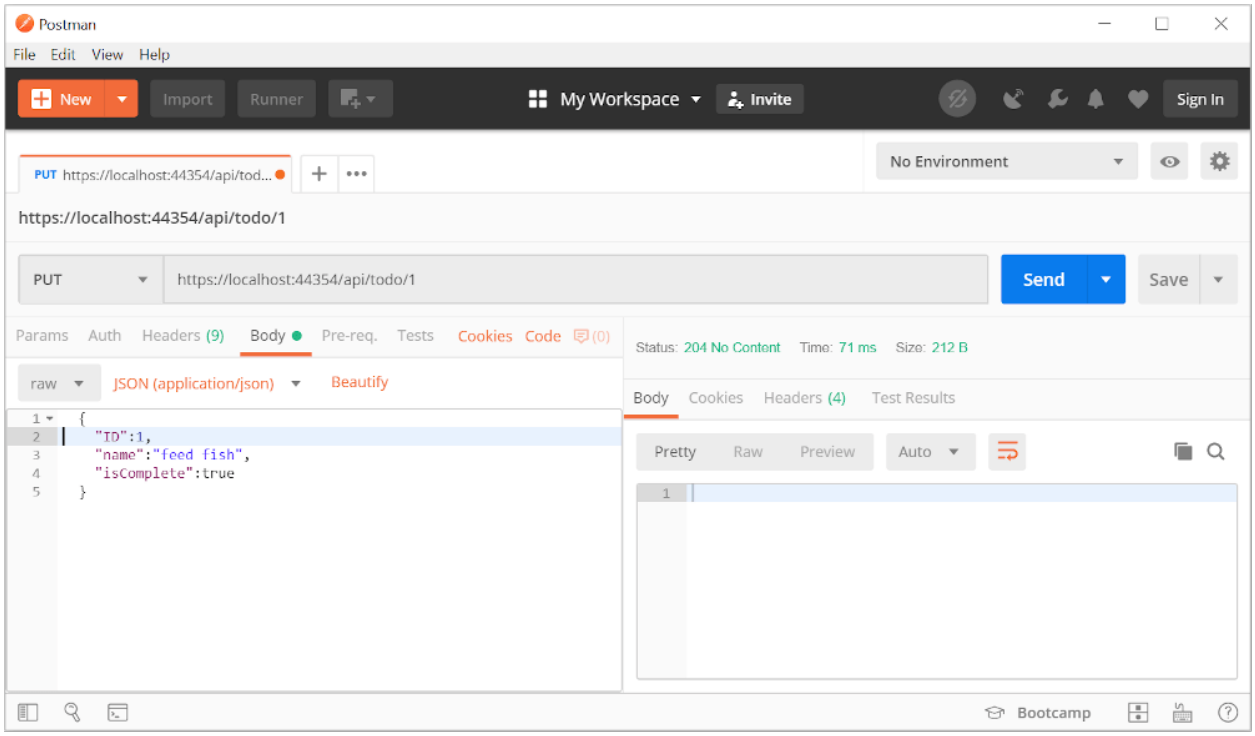
Test the PutTodoItem method 2.1

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has Id = 1 and set its name to "feed fish":

```
{
  "Id":1,
  "name":"feed fish",
  "isComplete":true
}
```

The following image shows the Postman update:



Add a DeleteTodoItem method 2.1

Add the following `DeleteTodoItem` method:

```
// DELETE: api/Todo/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

The `DeleteTodoItem` response is 204 (No Content).

Test the DeleteTodoItem method 2.1

Use Postman to delete a to-do item:

- Set the method to `DELETE`.
- Set the URI of the object to delete (for example, `https://localhost:5001/api/todo/1`).
- Select **Send**.

The sample app allows you to delete all the items. However, when the last item is deleted, a new one is created by the model class constructor the next time the API is called.

Call the web API with JavaScript 2.1

In this section, an HTML page is added that uses JavaScript to call the web API. jQuery initiates the request. JavaScript updates the page with the details from the web API's response.

Configure the app to serve static files and enable default file mapping by updating *Startup.cs* with the following highlighted code:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change this for
        // production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();
    app.UseHttpsRedirection();
    app.UseMvc();
}
```

Create a *wwwroot* folder in the project directory.

Add an HTML file named *index.html* to the *wwwroot* directory. Replace its contents with the following markup:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <style>
        input[type='submit'], button, [aria-label] {
            cursor: pointer;
        }

        #spoiler {
            display: none;
        }

        table {
            font-family: Arial, sans-serif;
            border: 1px solid;
            border-collapse: collapse;
        }

        th {
            background-color: #0066CC;
            color: white;
        }

        td {
            border: 1px solid;
            padding: 5px;
        }
    </style>
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST" onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="spoiler">
        <h3>Edit</h3>
        <form class="my-form">
            <input type="hidden" id="edit-id">
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"
            integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8="
```

```
            integrity="sha256-[illegible]"
            crossorigin="anonymous"></script>
    <script src="site.js"></script>
</body>
</html>
```

Add a JavaScript file named *site.js* to the *wwwroot* directory. Replace its contents with the following code:

```
const uri = "api/todo";
let todos = null;
function getCount(data) {
  const el = $("#counter");
  let name = "to-do";
  if (data) {
    if (data > 1) {
      name = "to-dos";
    }
    el.text(data + " " + name);
  } else {
    el.text("No " + name);
  }
}

$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });
```

```
      todos = data;
    }
  });
}

function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}

function deleteItem(id) {
  $.ajax({
    url: uri + "/" + id,
    type: "DELETE",
    success: function(result) {
      getData();
    }
  });
}

function editItem(id) {
  $.each(todos, function(key, item) {
    if (item.id === id) {
      $("#edit-name").val(item.name);
      $("#edit-id").val(item.id);
      $("#edit-isComplete")[0].checked = item.isComplete;
    }
  });
  $("#spoiler").css({ display: "block" });
}

$(".my-form").on("submit", function() {
  const item = {
    name: $("#edit-name").val(),
    isComplete: $("#edit-isComplete").is(":checked"),
    id: $("#edit-id").val()
  };

  $.ajax({
    url: uri + "/" + $("#edit-id").val(),
    type: "PUT",
    accepts: "application/json",
    contentType: "application/json",
    data: JSON.stringify(item),
    success: function(result) {
      getData();
    }
  });

  closeInput();
  return false;
```

```
  return false;
});

function closeInput() {
  $("#spoiler").css({ display: "none" });
}
```

A change to the ASP.NET Core project's launch settings may be required to test the HTML page locally:

- Open *Properties\launchSettings.json*.
- Remove the `launchUrl` property to force the app to open at *index.html*—the project's default file.

This sample calls all of the CRUD methods of the web API. Following are explanations of the calls to the API.

Get a list of to-do items 2.1

jQuery sends an HTTP GET request to the web API, which returns JSON representing an array of to-do items. The `success` callback function is invoked if the request succeeds. In the callback, the DOM is updated with the to-do information.

```
$(document).ready(function() {
  getData();
});

function getData() {
  $.ajax({
    type: "GET",
    url: uri,
    cache: false,
    success: function(data) {
      const tBody = $("#todos");

      $(tBody).empty();

      getCount(data.length);

      $.each(data, function(key, item) {
        const tr = $("<tr></tr>")
          .append(
            $("<td></td>").append(
              $("<input/>", {
                type: "checkbox",
                disabled: true,
                checked: item.isComplete
              })
            )
          )
          .append($("<td></td>").text(item.name))
          .append(
            $("<td></td>").append(
              $("<button>Edit</button>").on("click", function() {
                editItem(item.id);
              })
            )
          )
          .append(
            $("<td></td>").append(
              $("<button>Delete</button>").on("click", function() {
                deleteItem(item.id);
              })
            )
          );

        tr.appendTo(tBody);
      });

      todos = data;
    }
  });
}
```

Add a to-do item 2.1

jQuery sends an HTTP POST request with the to-do item in the request body. The `accepts` and `contentType` options are set to `application/json` to specify the media type being received and sent. The to-do item is converted to JSON by using JSON.stringify. When the API returns a successful status code, the `getData` function is invoked to update the HTML table.

```
function addItem() {
  const item = {
    name: $("#add-name").val(),
    isComplete: false
  };

  $.ajax({
    type: "POST",
    accepts: "application/json",
    url: uri,
    contentType: "application/json",
    data: JSON.stringify(item),
    error: function(jqXHR, textStatus, errorThrown) {
      alert("Something went wrong!");
    },
    success: function(result) {
      getData();
      $("#add-name").val("");
    }
  });
}
```

Update a to-do item 2.1

Updating a to-do item is similar to adding one. The `url` changes to add the unique identifier of the item, and the `type` is `PUT`.

```
$.ajax({
  url: uri + "/" + $("#edit-id").val(),
  type: "PUT",
  accepts: "application/json",
  contentType: "application/json",
  data: JSON.stringify(item),
  success: function(result) {
    getData();
  }
});
```

Delete a to-do item 2.1

Deleting a to-do item is accomplished by setting the `type` on the AJAX call to `DELETE` and specifying the item's unique identifier in the URL.

```
$.ajax({
  url: uri + "/" + id,
  type: "DELETE",
  success: function(result) {
    getData();
  }
});
```

Add authentication support to a web API 2.1

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- IdentityServer4

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see Welcome to IdentityServer4.

Additional resources 2.1

View or download sample code for this tutorial. See how to download.

For more information, see the following resources:

- Create web APIs with ASP.NET Core
- ASP.NET Core Web API help pages with Swagger / OpenAPI
- Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8
- Routing to controller actions in ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Deploy ASP.NET Core apps to Azure App Service
- Host and deploy ASP.NET Core
- YouTube version of this tutorial

Create a web API with ASP.NET Core and MongoDB

9/22/2020 • 21 minutes to read • Edit Online

By Pratik Khandelwal and Scott Addie

This tutorial creates a web API that performs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

- Configure MongoDB
- Create a MongoDB database
- Define a MongoDB collection and schema
- Perform MongoDB CRUD operations from a web API
- Customize JSON serialization

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- .NET Core SDK 3.0 or later
- Visual Studio 2019 with the **ASP.NET and web development** workload
- MongoDB

Configure MongoDB

If using Windows, MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `Path` environment variable. This change enables MongoDB access from anywhere on your development machine.

Use the mongo Shell in the following steps to create a database, make collections, and store documents. For more information on mongo Shell commands, see Working with the mongo Shell.

1. Choose a directory on your development machine for storing the data. For example, *C:\BooksData* on Windows. Create the directory if it doesn't exist. The mongo Shell doesn't create new directories.

2. Open a command shell. Run the following command to connect to MongoDB on default port 27017. Remember to replace `<data_directory_path>` with the directory you chose in the previous step.

   ```
   mongod --dbpath <data_directory_path>
   ```

3. Open another command shell instance. Connect to the default test database by running the following command:

   ```
   mongo
   ```

4. Run the following in a command shell:

```
use BookstoreDb
```

   If it doesn't already exist, a database named *BookstoreDb* is created. If the database does exist, its connection is opened for transactions.

5. Create a `Books` collection using following command:

```
db.createCollection('Books')
```

   The following result is displayed:

```
{ "ok" : 1 }
```

6. Define a schema for the `Books` collection and insert two documents using the following command:

```
db.Books.insertMany([{'Name':'Design Patterns','Price':54.93,'Category':'Computers','Author':'Ralph
Johnson'}, {'Name':'Clean Code','Price':43.15,'Category':'Computers','Author':'Robert C. Martin'}])
```

   The following result is displayed:

```
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5bfd996f7b8e48dc15ff215d"),
    ObjectId("5bfd996f7b8e48dc15ff215e")
  ]
}
```

   NOTE

   The ID's shown in this article will not match the IDs when you run this sample.

7. View the documents in the database using the following command:

```
db.Books.find({}).pretty()
```

   The following result is displayed:

```
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215d"),
  "Name" : "Design Patterns",
  "Price" : 54.93,
  "Category" : "Computers",
  "Author" : "Ralph Johnson"
}
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215e"),
  "Name" : "Clean Code",
  "Price" : 43.15,
  "Category" : "Computers",
  "Author" : "Robert C. Martin"
}
```

The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

The database is ready. You can start creating the ASP.NET Core web API.

Create the ASP.NET Core web API project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web Application** project type, and select **Next**.
3. Name the project *BooksApi*, and select **Create**.
4. Select the **.NET Core** target framework and **ASP.NET Core 3.0**. Select the **API** project template, and select **Create**.
5. Visit the NuGet Gallery: MongoDB.Driver to determine the latest stable version of the .NET driver for MongoDB. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

```
Install-Package MongoDB.Driver -Version {VERSION}
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BooksApi.Models
{
    public class Book
    {
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        [BsonElement("Name")]
        public string BookName { get; set; }

        public decimal Price { get; set; }

        public string Category { get; set; }

        public string Author { get; set; }
    }
}
```

In the preceding class, the `Id` property:

- Is required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Is annotated with `[BsonId]` to designate this property as the document's primary key.
- Is annotated with `[BsonRepresentation(BsonType.ObjectId)]` to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the `[BsonElement]` attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to *appsettings.json*:

```
{
  "BookstoreDatabaseSettings": {
    "BooksCollectionName": "Books",
    "ConnectionString": "mongodb://localhost:27017",
    "DatabaseName": "BookstoreDb"
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

2. Add a *BookstoreDatabaseSettings.cs* file to the *Models* directory with the following code:

```
namespace BooksApi.Models
{
    public class BookstoreDatabaseSettings : IBookstoreDatabaseSettings
    {
        public string BooksCollectionName { get; set; }
        public string ConnectionString { get; set; }
        public string DatabaseName { get; set; }
    }

    public interface IBookstoreDatabaseSettings
    {
        string BooksCollectionName { get; set; }
        string ConnectionString { get; set; }
        string DatabaseName { get; set; }
    }
}
```

The preceding `BookstoreDatabaseSettings` class is used to store the *appsettings.json* file's `BookstoreDatabaseSettings` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    // requires using Microsoft.Extensions.Options
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddControllers();
}
```

In the preceding code:

- The configuration instance to which the *appsettings.json* file's `BookstoreDatabaseSettings` section binds is registered in the Dependency Injection (DI) container. For example, a `BookstoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookstoreDatabaseSettings:ConnectionString` property in *appsettings.json*.
- The `IBookstoreDatabaseSettings` interface is registered in DI with a singleton service lifetime. When injected, the interface instance resolves to a `BookstoreDatabaseSettings` object.

4. Add the following code to the top of *Startup.cs* to resolve the `BookstoreDatabaseSettings` and `IBookstoreDatabaseSettings` references:

```
using BooksApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.
2. Add a `BookService` class to the *Services* directory with the following code:

```
using BooksApi.Models;
using MongoDB.Driver;
using System.Collections.Generic;
using System.Linq;

namespace BooksApi.Services
{
    public class BookService
    {
        private readonly IMongoCollection<Book> _books;

        public BookService(IBookstoreDatabaseSettings settings)
        {
            var client = new MongoClient(settings.ConnectionString);
            var database = client.GetDatabase(settings.DatabaseName);

            _books = database.GetCollection<Book>(settings.BooksCollectionName);
        }

        public List<Book> Get() =>
            _books.Find(book => true).ToList();

        public Book Get(string id) =>
            _books.Find<Book>(book => book.Id == id).FirstOrDefault();

        public Book Create(Book book)
        {
            _books.InsertOne(book);
            return book;
        }

        public void Update(string id, Book bookIn) =>
            _books.ReplaceOne(book => book.Id == id, bookIn);

        public void Remove(Book bookIn) =>
            _books.DeleteOne(book => book.Id == bookIn.Id);

        public void Remove(string id) =>
            _books.DeleteOne(book => book.Id == id);
    }
}
```

In the preceding code, an `IBookstoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the *appsettings.json* configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddControllers();
}
```

In the preceding code, the `BookService` class is registered with DI to support constructor injection in consuming classes. The singleton service lifetime is most appropriate because `BookService` takes a direct

dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of *Startup.cs* to resolve the `BookService` reference:

```
using BooksApi.Services;
```

The `BookService` class uses the following `MongoDB.Driver` members to perform CRUD operations against the database:

- MongoClient: Reads the server instance for performing database operations. The constructor of this class is provided the MongoDB connection string:

```
public BookService(IBookstoreDatabaseSettings settings)
{
    var client = new MongoClient(settings.ConnectionString);
    var database = client.GetDatabase(settings.DatabaseName);

    _books = database.GetCollection<Book>(settings.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for performing operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Perform CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOne: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOne: Inserts the provided object as a new document in the collection.
- ReplaceOne: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

```
using BooksApi.Models;
using BooksApi.Services;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace BooksApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly BookService _bookService;

        public BooksController(BookService bookService)
        {
            _bookService = bookService;
        }
```

```
        [HttpGet]
        public ActionResult<List<Book>> Get() =>
            _bookService.Get();

        [HttpGet("{id:length(24)}", Name = "GetBook")]
        public ActionResult<Book> Get(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            return book;
        }

        [HttpPost]
        public ActionResult<Book> Create(Book book)
        {
            _bookService.Create(book);

            return CreatedAtRoute("GetBook", new { id = book.Id.ToString() }, book);
        }

        [HttpPut("{id:length(24)}")]
        public IActionResult Update(string id, Book bookIn)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Update(id, bookIn);

            return NoContent();
        }

        [HttpDelete("{id:length(24)}")]
        public IActionResult Delete(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Remove(book.Id);

            return NoContent();
        }
    }
}
```

The preceding web API controller:

- Uses the `BookService` class to perform CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtRoute in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.
2. Navigate to `http://localhost:<port>/api/books` to test the controller's parameterless `Get` action method. The following JSON response is displayed:

```
[
  {
    "id":"5bfd996f7b8e48dc15ff215d",
    "bookName":"Design Patterns",
    "price":54.93,
    "category":"Computers",
    "author":"Ralph Johnson"
  },
  {
    "id":"5bfd996f7b8e48dc15ff215e",
    "bookName":"Clean Code",
    "price":43.15,
    "category":"Computers",
    "author":"Robert C. Martin"
  }
]
```

3. Navigate to `http://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. The following JSON response is displayed:

```
{
  "id":"{ID}",
  "bookName":"Clean Code",
  "price":43.15,
  "category":"Computers",
  "author":"Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. JSON.NET has been removed from ASP.NET shared framework. Add a package reference to `Microsoft.AspNetCore.Mvc.NewtonsoftJson`.
2. In `Startup.ConfigureServices`, chain the following highlighted code on to the `AddControllers` method call:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddControllers()
        .AddNewtonsoftJson(options => options.UseMemberCasing());
}
```

With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author`.

3. In *Models/Book.cs*, annotate the `BookName` property with the following `[JsonProperty]` attribute:

```
[BsonElement("Name")]
[JsonProperty("Name")]
public string BookName { get; set; }
```

The `[JsonProperty]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

4. Add the following code to the top of *Models/Book.cs* to resolve the `[JsonProperty]` attribute reference:

```
using Newtonsoft.Json;
```

5. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

This tutorial creates a web API that performs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

- Configure MongoDB
- Create a MongoDB database
- Define a MongoDB collection and schema
- Perform MongoDB CRUD operations from a web API
- Customize JSON serialization

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- .NET Core SDK 2.2
- Visual Studio 2019 with the **ASP.NET and web development** workload
- MongoDB

Configure MongoDB

If using Windows, MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `Path` environment variable. This change enables MongoDB access from anywhere on your development machine.

Use the mongo Shell in the following steps to create a database, make collections, and store documents. For more information on mongo Shell commands, see Working with the mongo Shell.

1. Choose a directory on your development machine for storing the data. For example, *C:\BooksData* on Windows. Create the directory if it doesn't exist. The mongo Shell doesn't create new directories.

2. Open a command shell. Run the following command to connect to MongoDB on default port 27017. Remember to replace `<data_directory_path>` with the directory you chose in the previous step.

   ```
   mongod --dbpath <data_directory_path>
   ```

3. Open another command shell instance. Connect to the default test database by running the following command:

   ```
   mongo
   ```

4. Run the following in a command shell:

   ```
   use BookstoreDb
   ```

   If it doesn't already exist, a database named *BookstoreDb* is created. If the database does exist, its connection is opened for transactions.

5. Create a `Books` collection using following command:

   ```
   db.createCollection('Books')
   ```

   The following result is displayed:

   ```
   { "ok" : 1 }
   ```

6. Define a schema for the `Books` collection and insert two documents using the following command:

   ```
   db.Books.insertMany([{'Name':'Design Patterns','Price':54.93,'Category':'Computers','Author':'Ralph
   Johnson'}, {'Name':'Clean Code','Price':43.15,'Category':'Computers','Author':'Robert C. Martin'}])
   ```

   The following result is displayed:

   ```
   {
     "acknowledged" : true,
     "insertedIds" : [
       ObjectId("5bfd996f7b8e48dc15ff215d"),
       ObjectId("5bfd996f7b8e48dc15ff215e")
     ]
   }
   ```

> **NOTE**
> The ID's shown in this article will not match the IDs when you run this sample.

7. View the documents in the database using the following command:

```
db.Books.find({}).pretty()
```

   The following result is displayed:

```
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215d"),
  "Name" : "Design Patterns",
  "Price" : 54.93,
  "Category" : "Computers",
  "Author" : "Ralph Johnson"
}
{
  "_id" : ObjectId("5bfd996f7b8e48dc15ff215e"),
  "Name" : "Clean Code",
  "Price" : 43.15,
  "Category" : "Computers",
  "Author" : "Robert C. Martin"
}
```

   The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

The database is ready. You can start creating the ASP.NET Core web API.

Create the ASP.NET Core web API project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web Application** project type, and select **Next**.
3. Name the project *BooksApi*, and select **Create**.
4. Select the **.NET Core** target framework and **ASP.NET Core 2.2**. Select the **API** project template, and select **Create**.
5. Visit the NuGet Gallery: MongoDB.Driver to determine the latest stable version of the .NET driver for MongoDB. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

```
Install-Package MongoDB.Driver -Version {VERSION}
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BooksApi.Models
{
    public class Book
    {
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }

        [BsonElement("Name")]
        public string BookName { get; set; }

        public decimal Price { get; set; }

        public string Category { get; set; }

        public string Author { get; set; }
    }
}
```

In the preceding class, the `Id` property:

- Is required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Is annotated with `[BsonId]` to designate this property as the document's primary key.
- Is annotated with `[BsonRepresentation(BsonType.ObjectId)]` to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the `[BsonElement]` attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to *appsettings.json*:

```
{
  "BookstoreDatabaseSettings": {
    "BooksCollectionName": "Books",
    "ConnectionString": "mongodb://localhost:27017",
    "DatabaseName": "BookstoreDb"
  },
  "Logging": {
    "IncludeScopes": false,
    "Debug": {
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Warning"
      }
    }
  }
}
```

2. Add a *BookstoreDatabaseSettings.cs* file to the *Models* directory with the following code:

```
namespace BooksApi.Models
{
    public class BookstoreDatabaseSettings : IBookstoreDatabaseSettings
    {
        public string BooksCollectionName { get; set; }
        public string ConnectionString { get; set; }
        public string DatabaseName { get; set; }
    }

    public interface IBookstoreDatabaseSettings
    {
        string BooksCollectionName { get; set; }
        string ConnectionString { get; set; }
        string DatabaseName { get; set; }
    }
}
```

The preceding `BookstoreDatabaseSettings` class is used to store the *appsettings.json* file's `BookstoreDatabaseSettings` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

In the preceding code:

- The configuration instance to which the *appsettings.json* file's `BookstoreDatabaseSettings` section binds is registered in the Dependency Injection (DI) container. For example, a `BookstoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookstoreDatabaseSettings:ConnectionString` property in *appsettings.json*.
- The `IBookstoreDatabaseSettings` interface is registered in DI with a singleton service lifetime. When injected, the interface instance resolves to a `BookstoreDatabaseSettings` object.

4. Add the following code to the top of *Startup.cs* to resolve the `BookstoreDatabaseSettings` and `IBookstoreDatabaseSettings` references:

```
using BooksApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.
2. Add a `BookService` class to the *Services* directory with the following code:

```
using BooksApi.Models;
using MongoDB.Driver;
using System.Collections.Generic;
using System.Linq;

namespace BooksApi.Services
{
    public class BookService
    {
        private readonly IMongoCollection<Book> _books;

        public BookService(IBookstoreDatabaseSettings settings)
        {
            var client = new MongoClient(settings.ConnectionString);
            var database = client.GetDatabase(settings.DatabaseName);

            _books = database.GetCollection<Book>(settings.BooksCollectionName);
        }

        public List<Book> Get() =>
            _books.Find(book => true).ToList();

        public Book Get(string id) =>
            _books.Find<Book>(book => book.Id == id).FirstOrDefault();

        public Book Create(Book book)
        {
            _books.InsertOne(book);
            return book;
        }

        public void Update(string id, Book bookIn) =>
            _books.ReplaceOne(book => book.Id == id, bookIn);

        public void Remove(Book bookIn) =>
            _books.DeleteOne(book => book.Id == bookIn.Id);

        public void Remove(string id) =>
            _books.DeleteOne(book => book.Id == id);
    }
}
```

In the preceding code, an `IBookstoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the *appsettings.json* configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

In the preceding code, the `BookService` class is registered with DI to support constructor injection in

consuming classes. The singleton service lifetime is most appropriate because `BookService` takes a direct dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of *Startup.cs* to resolve the `BookService` reference:

```
using BooksApi.Services;
```

The `BookService` class uses the following `MongoDB.Driver` members to perform CRUD operations against the database:

- MongoClient: Reads the server instance for performing database operations. The constructor of this class is provided the MongoDB connection string:

```
public BookService(IBookstoreDatabaseSettings settings)
{
    var client = new MongoClient(settings.ConnectionString);
    var database = client.GetDatabase(settings.DatabaseName);

    _books = database.GetCollection<Book>(settings.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for performing operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Perform CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOne: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOne: Inserts the provided object as a new document in the collection.
- ReplaceOne: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

```
using BooksApi.Models;
using BooksApi.Services;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace BooksApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase
    {
        private readonly BookService _bookService;

        public BooksController(BookService bookService)
        {
            _bookService = bookService;
```

```
        }

        [HttpGet]
        public ActionResult<List<Book>> Get() =>
            _bookService.Get();

        [HttpGet("{id:length(24)}", Name = "GetBook")]
        public ActionResult<Book> Get(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            return book;
        }

        [HttpPost]
        public ActionResult<Book> Create(Book book)
        {
            _bookService.Create(book);

            return CreatedAtRoute("GetBook", new { id = book.Id.ToString() }, book);
        }

        [HttpPut("{id:length(24)}")]
        public IActionResult Update(string id, Book bookIn)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Update(id, bookIn);

            return NoContent();
        }

        [HttpDelete("{id:length(24)}")]
        public IActionResult Delete(string id)
        {
            var book = _bookService.Get(id);

            if (book == null)
            {
                return NotFound();
            }

            _bookService.Remove(book.Id);

            return NoContent();
        }
    }
}
```

The preceding web API controller:

- Uses the `BookService` class to perform CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtRoute in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtRoute` also

adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.
2. Navigate to `http://localhost:<port>/api/books` to test the controller's parameterless `Get` action method. The following JSON response is displayed:

```
[
  {
    "id":"5bfd996f7b8e48dc15ff215d",
    "bookName":"Design Patterns",
    "price":54.93,
    "category":"Computers",
    "author":"Ralph Johnson"
  },
  {
    "id":"5bfd996f7b8e48dc15ff215e",
    "bookName":"Clean Code",
    "price":43.15,
    "category":"Computers",
    "author":"Robert C. Martin"
  }
]
```

3. Navigate to `http://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. The following JSON response is displayed:

```
{
  "id":"{ID}",
  "bookName":"Clean Code",
  "price":43.15,
  "category":"Computers",
  "author":"Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. In `Startup.ConfigureServices`, chain the following highlighted code on to the `AddMvc` method call:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<BookstoreDatabaseSettings>(
        Configuration.GetSection(nameof(BookstoreDatabaseSettings)));

    services.AddSingleton<IBookstoreDatabaseSettings>(sp =>
        sp.GetRequiredService<IOptions<BookstoreDatabaseSettings>>().Value);

    services.AddSingleton<BookService>();

    services.AddMvc()
            .AddJsonOptions(options => options.UseMemberCasing())
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author`.

2. In *Models/Book.cs*, annotate the `BookName` property with the following `[JsonProperty]` attribute:

```
[BsonElement("Name")]
[JsonProperty("Name")]
public string BookName { get; set; }
```

The `[JsonProperty]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

3. Add the following code to the top of *Models/Book.cs* to resolve the `[JsonProperty]` attribute reference:

```
using Newtonsoft.Json;
```

4. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- IdentityServer4

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see Welcome to IdentityServer4.

Next steps

For more information on building ASP.NET Core web APIs, see the following resources:

- YouTube version of this article
- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API

ASP.NET Core web API help pages with Swagger / OpenAPI

9/22/2020 • 2 minutes to read • Edit Online

By Christoph Nienaber and Rico Suter

When consuming a web API, understanding its various methods can be challenging for a developer. Swagger, also known as OpenAPI, solves the problem of generating useful documentation and help pages for web APIs. It provides benefits such as interactive documentation, client SDK generation, and API discoverability.

In this article, the Swashbuckle.AspNetCore and NSwag .NET Swagger implementations are showcased:

- **Swashbuckle.AspNetCore** is an open source project for generating Swagger documents for ASP.NET Core Web APIs.
- **NSwag** is another open source project for generating Swagger documents and integrating Swagger UI or ReDoc into ASP.NET Core web APIs. Additionally, NSwag offers approaches to generate C# and TypeScript client code for your API.

What is Swagger / OpenAPI?

Swagger is a language-agnostic specification for describing REST APIs. The Swagger project was donated to the OpenAPI Initiative, where it's now referred to as OpenAPI. Both names are used interchangeably; however, OpenAPI is preferred. It allows both computers and humans to understand the capabilities of a service without any direct access to the implementation (source code, network access, documentation). One goal is to minimize the amount of work needed to connect disassociated services. Another goal is to reduce the amount of time needed to accurately document a service.

OpenAPI specification (openapi.json)

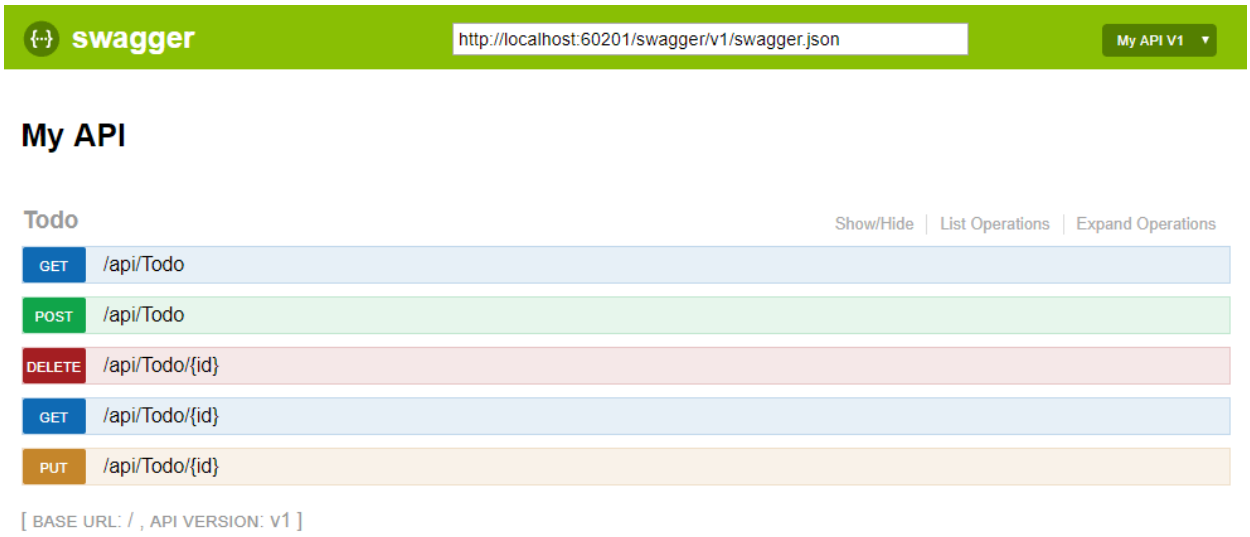
The core to the OpenAPI flow is the specification—by default, a document named *openapi.json*. It's generated by the OpenAPI tool chain (or third-party implementations of it) based on your service. It describes the capabilities of your API and how to access it with HTTP. It drives the Swagger UI and is used by the tool chain to enable discovery and client code generation. Here's an example of an OpenAPI specification, reduced for brevity:

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "API V1",
    "version": "v1"
  },
  "paths": {
    "/api/Todo": {
      "get": {
        "tags": [
          "Todo"
        ],
        "operationId": "ApiTodoGet",
        "responses": {
          "200": {
            "description": "Success",
            "content": {
              "text/plain": {
                "schema": {
                  "type": "array"
```

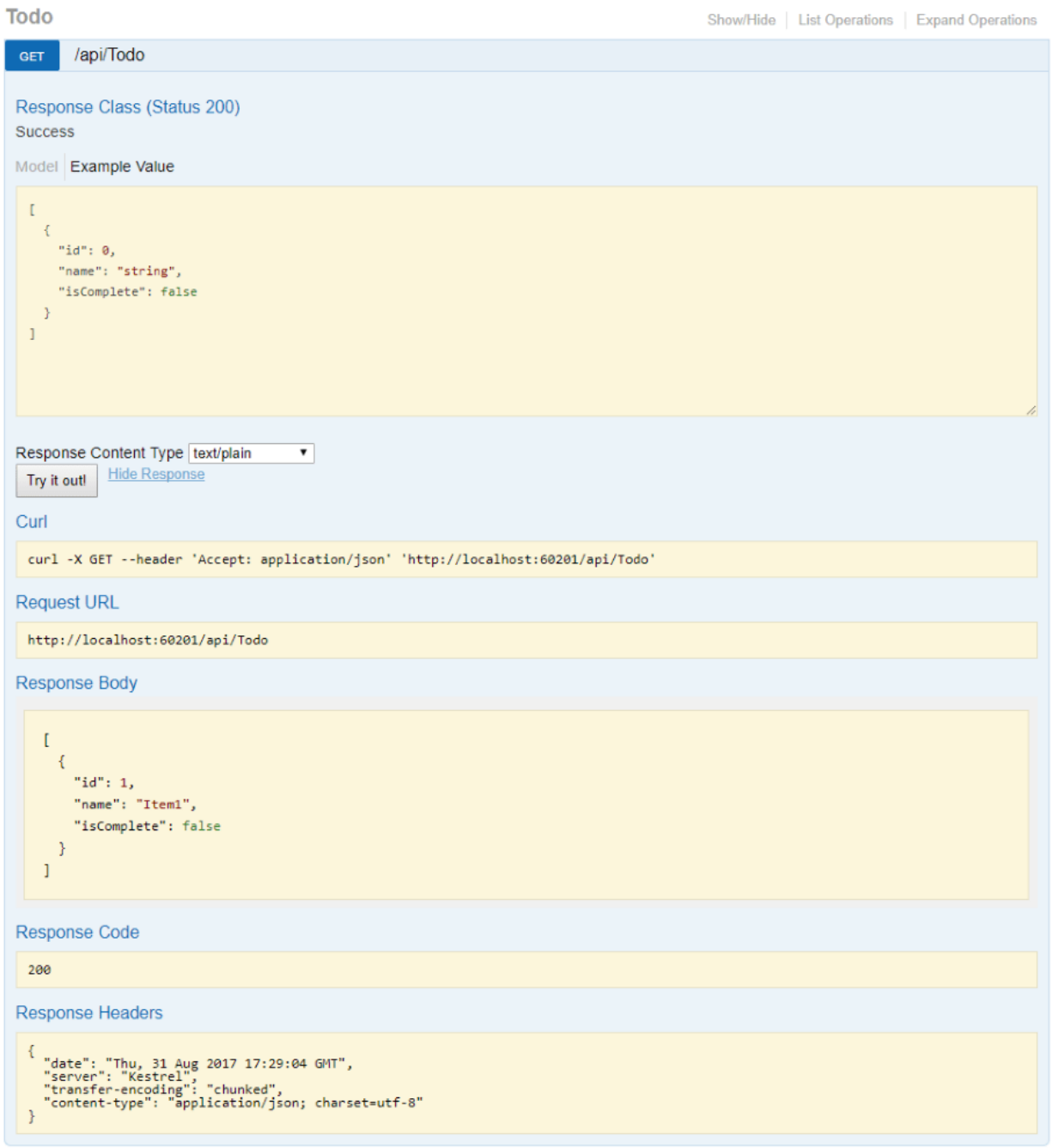
```
                "type": "array",
                "items": {
                  "$ref": "#/components/schemas/ToDoItem"
                }
              }
            },
            "application/json": {
              "schema": {
                "type": "array",
                "items": {
                  "$ref": "#/components/schemas/ToDoItem"
                }
              }
            },
            "text/json": {
              "schema": {
                "type": "array",
                "items": {
                  "$ref": "#/components/schemas/ToDoItem"
                }
              }
            }
          }
        }
      },
      "post": {
        …
      }
    },
    "/api/Todo/{id}": {
      "get": {
        …
      },
      "put": {
        …
      },
      "delete": {
        …
      }
    }
  },
  "components": {
    "schemas": {
      "ToDoItem": {
        "type": "object",
        "properties": {
          "id": {
            "type": "integer",
            "format": "int32"
          },
          "name": {
            "type": "string",
            "nullable": true
          },
          "isCompleted": {
            "type": "boolean"
          }
        },
        "additionalProperties": false
      }
    }
  }
}
```


Swagger UI

Swagger UI offers a web-based UI that provides information about the service, using the generated OpenAPI specification. Both Swashbuckle and NSwag include an embedded version of Swagger UI, so that it can be hosted in your ASP.NET Core app using a middleware registration call. The web UI looks like this:



Each public action method in your controllers can be tested from the UI. Click a method name to expand the section. Add any necessary parameters, and click **Try it out!**.



> **NOTE**
>
> The Swagger UI version used for the screenshots is version 2. For a version 3 example, see Petstore example.

Next steps

- Get started with Swashbuckle
- Get started with NSwag

Get started with Swashbuckle and ASP.NET Core

9/22/2020 • 14 minutes to read • Edit Online

By Shayne Boyer and Scott Addie

View or download sample code (how to download)

There are three main components to Swashbuckle:

- Swashbuckle.AspNetCore.Swagger: a Swagger object model and middleware to expose `SwaggerDocument` objects as JSON endpoints.
- Swashbuckle.AspNetCore.SwaggerGen: a Swagger generator that builds `SwaggerDocument` objects directly from your routes, controllers, and models. It's typically combined with the Swagger endpoint middleware to automatically expose Swagger JSON.
- Swashbuckle.AspNetCore.SwaggerUI: an embedded version of the Swagger UI tool. It interprets Swagger JSON to build a rich, customizable experience for describing the web API functionality. It includes built-in test harnesses for the public methods.

Package installation

Swashbuckle can be added with the following approaches:

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- .NET Core CLI
- From the **Package Manager Console** window:
  - Go to **View** > **Other Windows** > **Package Manager Console**
  - Navigate to the directory in which the *TodoApi.csproj* file exists
  - Execute the following command:

    ```
    Install-Package Swashbuckle.AspNetCore -Version 5.5.0
    ```

- From the **Manage NuGet Packages** dialog:
  - Right-click the project in **Solution Explorer** > **Manage NuGet Packages**
  - Set the **Package source** to "nuget.org"
  - Ensure the "Include prerelease" option is enabled
  - Enter "Swashbuckle.AspNetCore" in the search box
  - Select the latest "Swashbuckle.AspNetCore" package from the **Browse** tab and click **Install**

Add and configure Swagger middleware

Add the Swagger generator to the services collection in the `Startup.ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddControllers();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen();
}
```

In the `Startup.Configure` method, enable the middleware for serving the generated JSON document and the Swagger UI:

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

NOTE

Swashbuckle relies on MVC's Microsoft.AspNetCore.Mvc.ApiExplorer to discover the routes and endpoints. If the project calls AddMvc, routes and endpoints are discovered automatically. When calling AddMvcCore, the AddApiExplorer method must be explicitly called. For more information, see Swashbuckle, ApiExplorer, and Routing.

The preceding `UseSwaggerUI` method call enables the Static File Middleware. If targeting .NET Framework or .NET Core 1.x, add the Microsoft.AspNetCore.StaticFiles NuGet package to the project.

Launch the app, and navigate to `http://localhost:<port>/swagger/v1/swagger.json`. The generated document describing the endpoints appears as shown in OpenAPI specification (openapi.json).

The Swagger UI can be found at `http://localhost:<port>/swagger`. Explore the API via Swagger UI and incorporate it in other programs.

TIP

To serve the Swagger UI at the app's root (`http://localhost:<port>/`), set the `RoutePrefix` property to an empty string:

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    c.RoutePrefix = string.Empty;
});
```

If using directories with IIS or a reverse proxy, set the Swagger endpoint to a relative path using the `./` prefix. For example, `./swagger/v1/swagger.json`. Using `/swagger/v1/swagger.json` instructs the app to look for the JSON file at the true root of the URL (plus the route prefix, if used). For example, use `http://localhost:<port>/<route_prefix>/swagger/v1/swagger.json` instead of `http://localhost:<port>/<virtual_directory>/<route_prefix>/swagger/v1/swagger.json`.

NOTE

By default, Swashbuckle generates and exposes Swagger JSON in version 3.0 of the specification—officially called the OpenAPI Specification. To support backwards compatibility, you can opt into exposing JSON in the 2.0 format instead. This 2.0 format is important for integrations such as Microsoft Power Apps and Microsoft Flow that currently support OpenAPI version 2.0. To opt into the 2.0 format, set the `SerializeAsV2` property in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger(c =>
    {
        c.SerializeAsV2 = true;
    });

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Customize and extend

Swagger provides options for documenting the object model and customizing the UI to match your theme.

In the `Startup` class, add the following namespaces:

```
using System;
using System.Reflection;
using System.IO;
```

API info and description

The configuration action passed to the `AddSwaggerGen` method adds information such as the author, license, and description:

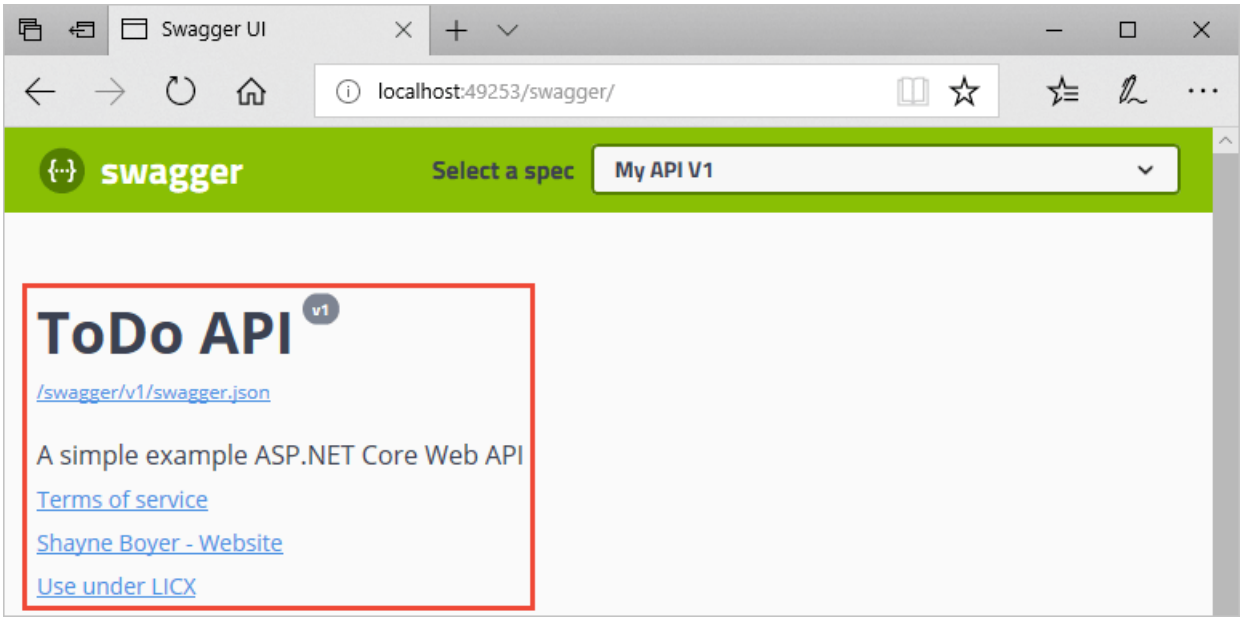
In the `Startup` class, import the following namespace to use the `OpenApiInfo` class:

```
using Microsoft.OpenApi.Models;
```

Using the `OpenApiInfo` class, modify the information displayed in the UI:

```
// Register the Swagger generator, defining 1 or more Swagger documents
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "A simple example ASP.NET Core Web API",
        TermsOfService = new Uri("https://example.com/terms"),
        Contact = new OpenApiContact
        {
            Name = "Shayne Boyer",
            Email = string.Empty,
            Url = new Uri("https://twitter.com/spboyer"),
        },
        License = new OpenApiLicense
        {
            Name = "Use under LICX",
            Url = new Uri("https://example.com/license"),
        }
    });
});
```

The Swagger UI displays the version's information:



XML comments

XML comments can be enabled with the following approaches:

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- .NET Core CLI

- Right-click the project in **Solution Explorer** and select **Edit <project_name>.csproj**.
- Manually add the highlighted lines to the *.csproj* file:

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

- Right-click the project in **Solution Explorer** and select **Properties**.
- Check the **XML documentation file** box under the **Output** section of the **Build** tab.

Enabling XML comments provides debug information for undocumented public types and members. Undocumented types and members are indicated by the warning message. For example, the following message indicates a violation of warning code 1591:

```
warning CS1591: Missing XML comment for publicly visible type or member 'TodoController.GetAll()'
```

To suppress warnings project-wide, define a semicolon-delimited list of warning codes to ignore in the project file. Appending the warning codes to `$(NoWarn);` applies the C# default values too.

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

```
<PropertyGroup>
  <DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

To suppress warnings only for specific members, enclose the code in #pragma warning preprocessor directives. This approach is useful for code that shouldn't be exposed via the API docs. In the following example, warning code CS1591 is ignored for the entire `Program` class. Enforcement of the warning code is restored at the close of the class definition. Specify multiple warning codes with a comma-delimited list.

```
namespace TodoApi
{
#pragma warning disable CS1591
    public class Program
    {
        public static void Main(string[] args) =>
            BuildWebHost(args).Run();

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
#pragma warning restore CS1591
}
```

Configure Swagger to use the XML file that's generated with the preceding instructions. For Linux or non-Windows operating systems, file names and paths can be case-sensitive. For example, a *TodoApi.XML* file is valid on Windows but not CentOS.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddControllers();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = new Uri("https://example.com/terms"),
            Contact = new OpenApiContact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = new Uri("https://twitter.com/spboyer"),
            },
            License = new OpenApiLicense
            {
                Name = "Use under LICX",
                Url = new Uri("https://example.com/license"),
            }
        });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = new Uri("https://example.com/terms"),
            Contact = new OpenApiContact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = new Uri("https://twitter.com/spboyer"),
            },
            License = new OpenApiLicense
            {
                Name = "Use under LICX",
                Url = new Uri("https://example.com/license"),
            }
        });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = new Uri("https://example.com/terms"),
            Contact = new OpenApiContact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = new Uri("https://twitter.com/spboyer"),
            },
            License = new OpenApiLicense
            {
                Name = "Use under LICX",
                Url = new Uri("https://example.com/license"),
            }
        });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo
        {
            Version = "v1",
            Title = "ToDo API",
            Description = "A simple example ASP.NET Core Web API",
            TermsOfService = new Uri("https://example.com/terms"),
            Contact = new OpenApiContact
            {
                Name = "Shayne Boyer",
                Email = string.Empty,
                Url = new Uri("https://twitter.com/spboyer"),
            },
            License = new OpenApiLicense
            {
                Name = "Use under LICX",
                Url = new Uri("https://example.com/license"),
            }
        });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetEntryAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

In the preceding code, Reflection is used to build an XML file name matching that of the web API project. The AppContext.BaseDirectory property is used to construct a path to the XML file. Some Swagger features (for example, schemata of input parameters or HTTP methods and response codes from the respective attributes) work without the use of an XML documentation file. For most features, namely method summaries and the descriptions of parameters and response codes, the use of an XML file is mandatory.

Adding triple-slash comments to an action enhances the Swagger UI by adding the description to the section header. Add a <summary> element above the `Delete` action:

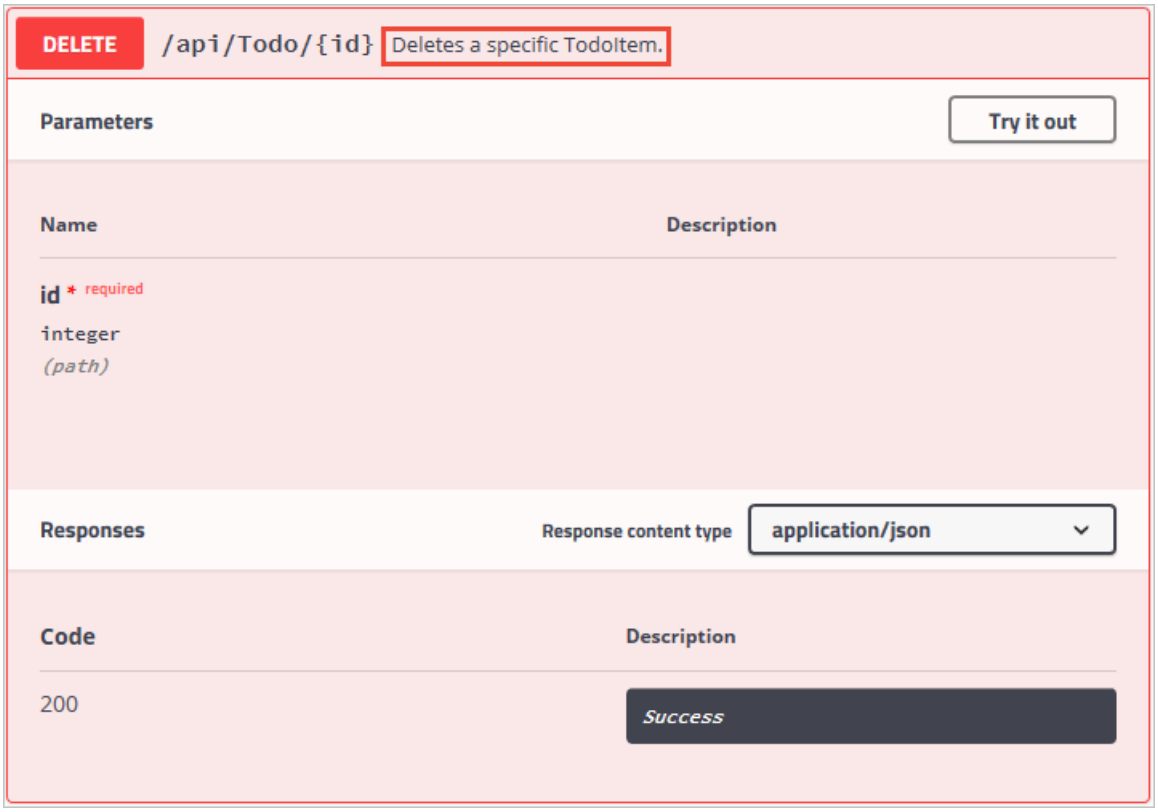
```
/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
[HttpDelete("{id}")]
public IActionResult Delete(long id)
{
    var todo = _context.TodoItems.Find(id);

    if (todo == null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(todo);
    _context.SaveChanges();

    return NoContent();
}
```

The Swagger UI displays the inner text of the preceding code's `<summary>` element:



The UI is driven by the generated JSON schema:

```
"delete": {
    "tags": [
        "Todo"
    ],
    "summary": "Deletes a specific TodoItem.",
    "operationId": "ApiTodoByIdDelete",
    "consumes": [],
    "produces": [],
    "parameters": [
        {
            "name": "id",
            "in": "path",
            "description": "",
            "required": true,
            "type": "integer",
            "format": "int64"
        }
    ],
    "responses": {
        "200": {
            "description": "Success"
        }
    }
}
```

Add a <remarks> element to the `Create` action method documentation. It supplements information specified in the `<summary>` element and provides a more robust Swagger UI. The `<remarks>` element content can consist of text, JSON, or XML.

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

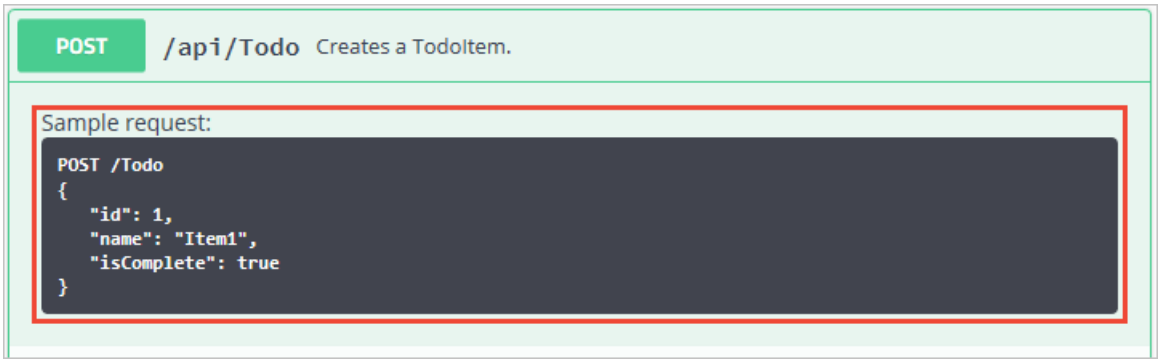
```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

Notice the UI enhancements with these additional comments:



Data annotations

Mark the model with attributes, found in the System.ComponentModel.DataAnnotations namespace, to help drive the Swagger UI components.

Add the `[Required]` attribute to the `Name` property of the `TodoItem` class:

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }

        [Required]
        public string Name { get; set; }

        [DefaultValue(false)]
        public bool IsComplete { get; set; }
    }
}
```

The presence of this attribute changes the UI behavior and alters the underlying JSON schema:

```
"definitions": {
    "TodoItem": {
        "required": [
            "name"
        ],
        "type": "object",
        "properties": {
            "id": {
                "format": "int64",
                "type": "integer"
            },
            "name": {
                "type": "string"
            },
            "isComplete": {
                "default": false,
                "type": "boolean"
            }
        }
    }
},
```

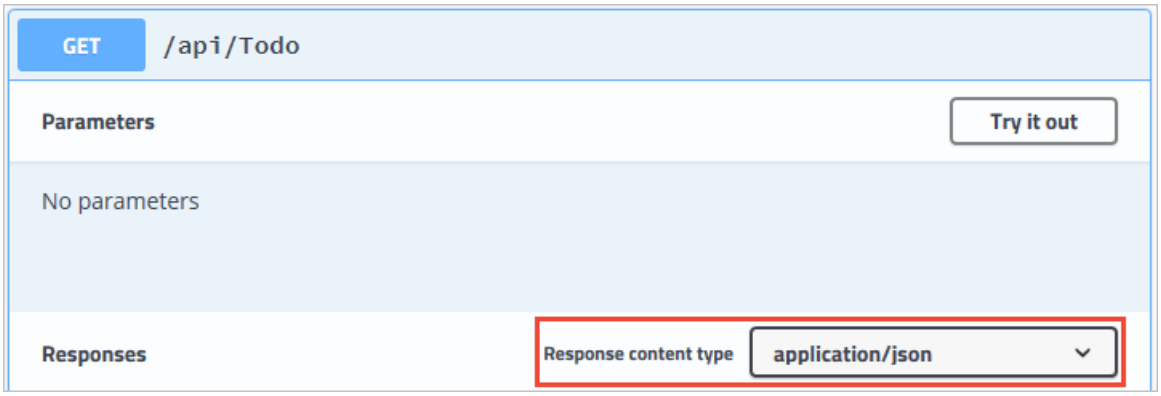
Add the `[Produces("application/json")]` attribute to the API controller. Its purpose is to declare that the controller's actions support a response content type of *application/json*:

```
[Produces("application/json")]
[Route("api/[controller]")]
public class TodoController : ControllerBase
{
    private readonly TodoContext _context;
```

```
[Produces("application/json")]
[Route("api/[controller]")]
[ApiController]
public class TodoController : ControllerBase
{
    private readonly TodoContext _context;
```

```
[Produces("application/json")]
[Route("api/[controller]")]
[ApiController]
public class TodoController : ControllerBase
{
    private readonly TodoContext _context;
```

The **Response Content Type** drop-down selects this content type as the default for the controller's GET actions:



As the usage of data annotations in the web API increases, the UI and API help pages become more descriptive and useful.

Describe response types

Developers consuming a web API are most concerned with what's returned—specifically response types and error codes (if not standard). The response types and error codes are denoted in the XML comments and data annotations.

The `Create` action returns an HTTP 201 status code on success. An HTTP 400 status code is returned when the posted request body is null. Without proper documentation in the Swagger UI, the consumer lacks knowledge of these expected outcomes. Fix that problem by adding the highlighted lines in the following example:

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(typeof(TodoItem), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

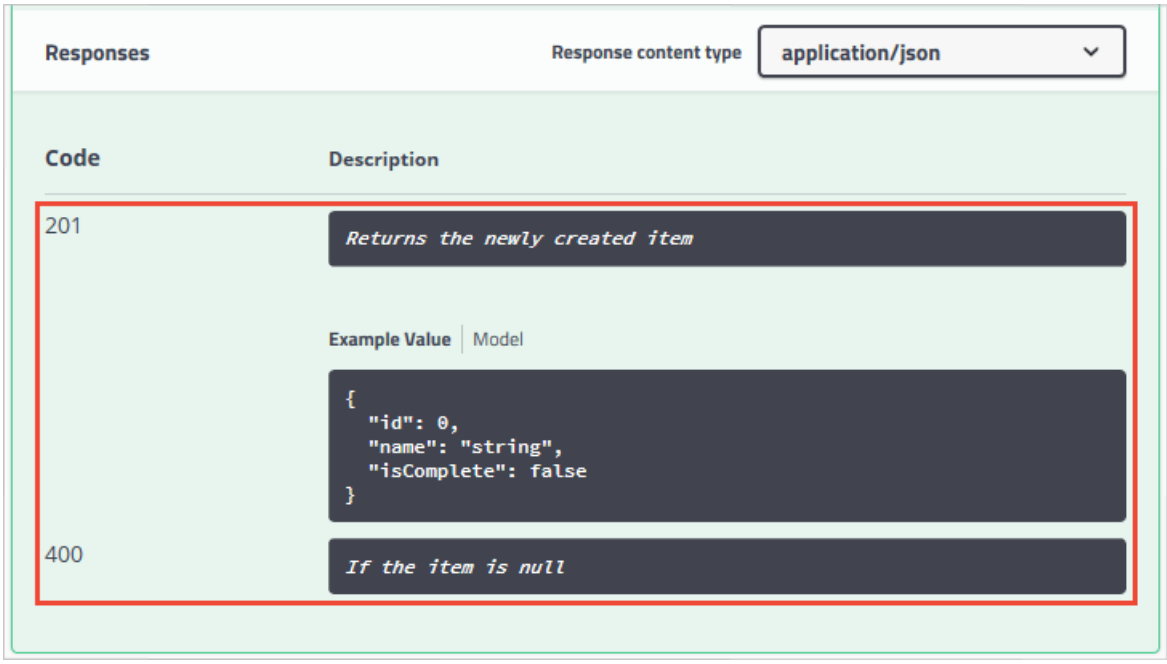
```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

The Swagger UI now clearly documents the expected HTTP response codes:



In ASP.NET Core 2.2 or later, conventions can be used as an alternative to explicitly decorating individual actions with `[ProducesResponseType]`. For more information, see Use web API conventions.

To support the `[ProducesResponseType]` decoration, the Swashbuckle.AspNetCore.Annotations package offers extensions to enable and enrich the response, schema, and parameter metadata.

Customize the UI

The default UI is both functional and presentable. However, API documentation pages should represent your brand or theme. Branding the Swashbuckle components requires adding the resources to serve static files and building the folder structure to host those files.

If targeting .NET Framework or .NET Core 1.x, add the Microsoft.AspNetCore.StaticFiles NuGet package to the project:

```
<PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
```

The preceding NuGet package is already installed if targeting .NET Core 2.x and using the metapackage.

Enable Static File Middleware:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseMvc();
}
```

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

To inject additional CSS stylesheets, add them to the project's *wwwroot* folder and specify the relative path in the middleware options:

```
app.UseSwaggerUI(c =>
{
     c.InjectStylesheet("/swagger-ui/custom.css");
}
```

Get started with NSwag and ASP.NET Core

9/22/2020 • 6 minutes to read • Edit Online

By Christoph Nienaber, Rico Suter, and Dave Brock

View or download sample code (how to download)

View or download sample code (how to download)

NSwag offers the following capabilities:

- The ability to utilize the Swagger UI and Swagger generator.
- Flexible code generation capabilities.

With NSwag, you don't need an existing API—you can use third-party APIs that incorporate Swagger and generate a client implementation. NSwag allows you to expedite the development cycle and easily adapt to API changes.

Register the NSwag middleware

Register the NSwag middleware to:

- Generate the Swagger specification for the implemented web API.
- Serve the Swagger UI to browse and test the web API.

To use the NSwag ASP.NET Core middleware, install the NSwag.AspNetCore NuGet package. This package contains the middleware to generate and serve the Swagger specification, Swagger UI (v2 and v3), and ReDoc UI.

Use one of the following approaches to install the NSwag NuGet package:

- Visual Studio
- Visual Studio for Mac
- .NET Core CLI

- From the **Package Manager Console** window:
  - Go to **View** > **Other Windows** > **Package Manager Console**
  - Navigate to the directory in which the *TodoApi.csproj* file exists
  - Execute the following command:

    ```
    Install-Package NSwag.AspNetCore
    ```

- From the **Manage NuGet Packages** dialog:
  - Right-click the project in **Solution Explorer** > **Manage NuGet Packages**
  - Set the **Package source** to "nuget.org"
  - Enter "NSwag.AspNetCore" in the search box
  - Select the "NSwag.AspNetCore" package from the **Browse** tab and click **Install**

Add and configure Swagger middleware

Add and configure Swagger in your ASP.NET Core app by performing the following steps:

- In the `Startup.ConfigureServices` method, register the required Swagger services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt =>
        opt.UseInMemoryDatabase("TodoList"));
    services.AddMvc();

    // Register the Swagger services
    services.AddSwaggerDocument();
}
```

- In the `Startup.Configure` method, enable the middleware for serving the generated Swagger specification and the Swagger UI:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();

    // Register the Swagger generator and the Swagger UI middlewares
    app.UseOpenApi();
    app.UseSwaggerUi3();

    app.UseMvc();
}
```

- Launch the app. Navigate to:
  - `http://localhost:<port>/swagger` to view the Swagger UI.
  - `http://localhost:<port>/swagger/v1/swagger.json` to view the Swagger specification.

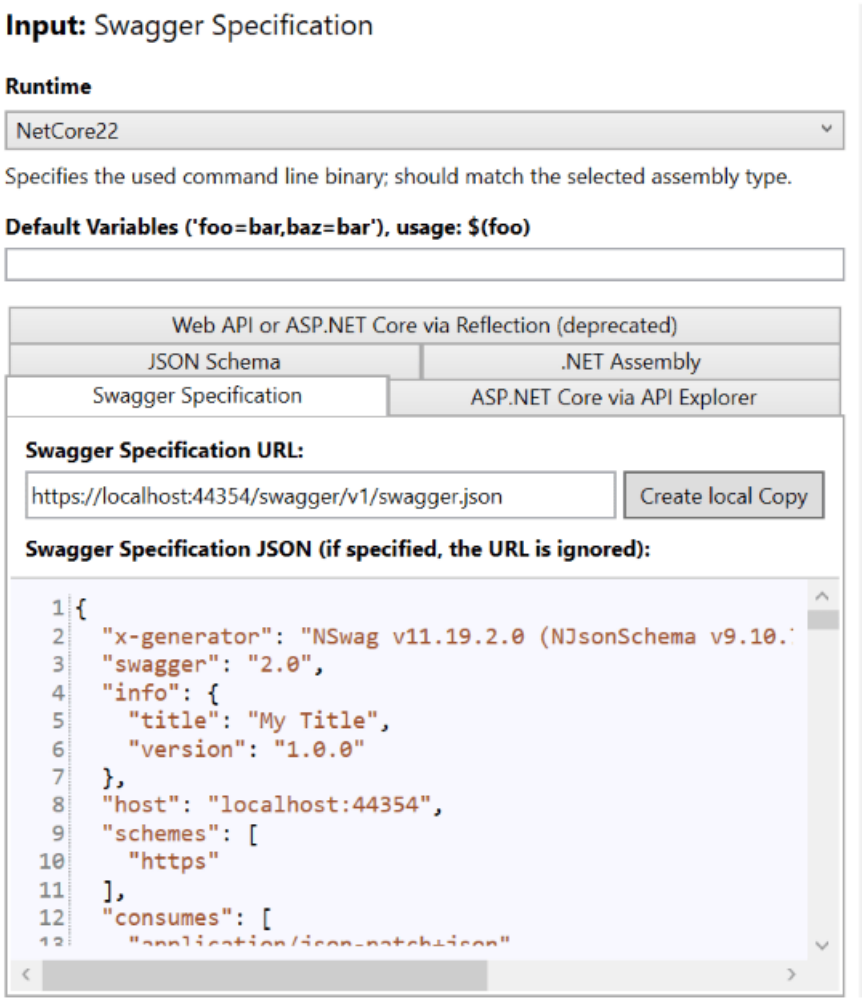
Code generation

You can take advantage of NSwag's code generation capabilities by choosing one of the following options:

- NSwagStudio: A Windows desktop app for generating API client code in C# or TypeScript.
- The NSwag.CodeGeneration.CSharp or NSwag.CodeGeneration.TypeScript NuGet packages for code generation inside your project.
- NSwag from the command line.
- The NSwag.MSBuild NuGet package.
- The Unchase OpenAPI (Swagger) Connected Service: A Visual Studio Connected Service for generating API client code in C# or TypeScript. Also generates C# controllers for OpenAPI services with NSwag.

Generate code with NSwagStudio

- Install NSwagStudio by following the instructions at the NSwagStudio GitHub repository. On the NSwag release page you can download an xcopy version which can be started without installation and admin privileges.
- Launch NSwagStudio and enter the *swagger.json* file URL in the **Swagger Specification URL** text box. For example, *http://localhost:44354/swagger/v1/swagger.json*.
- Click the **Create local Copy** button to generate a JSON representation of your Swagger specification.



- In the **Outputs** area, click the **CSharp Client** check box. Depending on your project, you can also choose **TypeScript Client** or **CSharp Web API Controller**. If you select **CSharp Web API Controller**, a service specification rebuilds the service, serving as a reverse generation.
- Click **Generate Outputs** to produce a complete C# client implementation of the *TodoApi.NSwag* project. To see the generated client code, click the **CSharp Client** tab:

```
//----------------------
// <auto-generated>
//     Generated using the NSwag toolchain v12.0.9.0 (NJsonSchema v9.13.10.0 (Newtonsoft.Json v11.0.0.0))
(http://NSwag.org)
// </auto-generated>
//----------------------

namespace MyNamespace
{
    #pragma warning disable

    [System.CodeDom.Compiler.GeneratedCode("NSwag", "12.0.9.0 (NJsonSchema v9.13.10.0 (Newtonsoft.Json
v11.0.0.0))")]
    public partial class TodoClient
    {
        private string _baseUrl = "https://localhost:44354";
        private System.Net.Http.HttpClient _httpClient;
        private System.Lazy<Newtonsoft.Json.JsonSerializerSettings> _settings;

        public TodoClient(System.Net.Http.HttpClient httpClient)
        {
            _httpClient = httpClient;
            _settings = new System.Lazy<Newtonsoft.Json.JsonSerializerSettings>(() =>
            {
                var settings = new Newtonsoft.Json.JsonSerializerSettings();
                UpdateJsonSerializerSettings(settings);
                return settings;
            });
        }

        public string BaseUrl
        {
            get { return _baseUrl; }
            set { _baseUrl = value; }
        }

        // code omitted for brevity
```

TIP

The C# client code is generated based on selections in the **Settings** tab. Modify the settings to perform tasks such as default namespace renaming and synchronous method generation.

- Copy the generated C# code into a file in the client project that will consume the API.
- Start consuming the web API:

```
 var todoClient = new TodoClient();

// Gets all to-dos from the API
 var allTodos = await todoClient.GetAllAsync();

 // Create a new TodoItem, and save it via the API.
var createdTodo = await todoClient.CreateAsync(new TodoItem());

// Get a single to-do by ID
var foundTodo = await todoClient.GetByIdAsync(1);
```

Customize API documentation

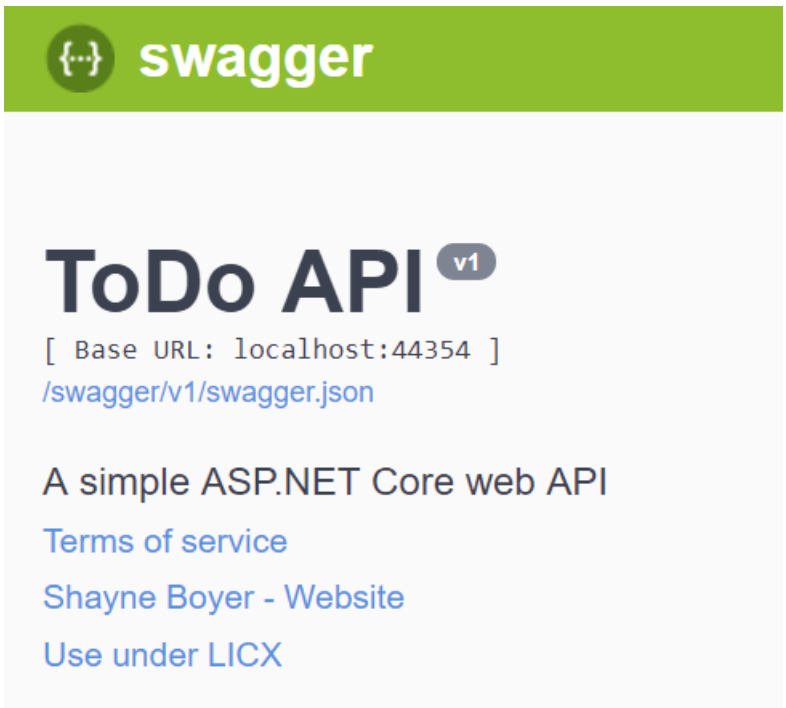
Swagger provides options for documenting the object model to ease consumption of the web API.

API info and description

In the `Startup.ConfigureServices` method, a configuration action passed to the `AddSwaggerDocument` method adds information such as the author, license, and description:

```
services.AddSwaggerDocument(config =>
{
    config.PostProcess = document =>
    {
        document.Info.Version = "v1";
        document.Info.Title = "ToDo API";
        document.Info.Description = "A simple ASP.NET Core web API";
        document.Info.TermsOfService = "None";
        document.Info.Contact = new NSwag.OpenApiContact
        {
            Name = "Shayne Boyer",
            Email = string.Empty,
            Url = "https://twitter.com/spboyer"
        };
        document.Info.License = new NSwag.OpenApiLicense
        {
            Name = "Use under LICX",
            Url = "https://example.com/license"
        };
    };
});
```

The Swagger UI displays the version's information:



XML comments

To enable XML comments, perform the following steps:

- Visual Studio
- Visual Studio for Mac
- .NET Core CLI

- Right-click the project in **Solution Explorer** and select **Edit <project_name>.csproj**.
- Manually add the highlighted lines to the *.csproj* file:

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

- Right-click the project in **Solution Explorer** and select **Properties**
- Check the **XML documentation file** box under the **Output** section of the **Build** tab

Data annotations

Because NSwag uses Reflection, and the recommended return type for web API actions is IActionResult, it can't infer what your action is doing and what it returns.

Consider the following example:

```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

The preceding action returns `IActionResult`, but inside the action it's returning either CreatedAtRoute or BadRequest. Use data annotations to tell clients which HTTP status codes this action is known to return. Mark the action with the following attributes:

```
[ProducesResponseType(typeof(TodoItem), StatusCodes.Status201Created)]  // Created
[ProducesResponseType(StatusCodes.Status400BadRequest)]                 // BadRequest
```

Because NSwag uses Reflection, and the recommended return type for web API actions is ActionResult<T>, it can only infer the return type defined by `T`. You can't automatically infer other possible return types.

Consider the following example:

```
[HttpPost]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

The preceding action returns `ActionResult<T>`. Inside the action, it's returning CreatedAtRoute. Since the controller has the `[ApiController]` attribute, a BadRequest response is possible, too. For more information, see Automatic HTTP 400 responses. Use data annotations to tell clients which HTTP status codes this action is known to return. Mark the action with the following attributes:

```
[ProducesResponseType(StatusCodes.Status201Created)]     // Created
[ProducesResponseType(StatusCodes.Status400BadRequest)]  // BadRequest
```

In ASP.NET Core 2.2 or later, you can use conventions instead of explicitly decorating individual actions with `[ProducesResponseType]`. For more information, see Use web API conventions.

The Swagger generator can now accurately describe this action, and generated clients know what they receive when calling the endpoint. As a recommendation, mark all actions with these attributes.

For guidelines on what HTTP responses your API actions should return, see the RFC 7231 specification.

Develop ASP.NET Core apps using OpenAPI tools

9/22/2020 • 2 minutes to read • Edit Online

By Ryan Brandenburg

Microsoft.dotnet-openapi is a .NET Core Global Tool for managing OpenAPI references within a project.

Installation

To install `Microsoft.dotnet-openapi`, run the following command:

```
dotnet tool install -g Microsoft.dotnet-openapi
```

Add

Adding an OpenAPI reference using any of the commands on this page adds an `<OpenApiReference />` element similar to the following to the *.csproj* file:

```
<OpenApiReference Include="openapi.json" />
```

The preceding reference is required for the app to call the generated client code.

Add File

Options

SHORT OPTION	LONG OPTION	DESCRIPTION	EXAMPLE
-p	--updateProject	The project to operate on.	dotnet openapi add file --*updateProject* .*Ref.csproj* .\OpenAPI.json
-c	--code-generator	The code generator to apply to the reference. Options are `NSwagCSharp` and `NSwagTypeScript`. If `--code-generator` is not specified the tooling defaults to `NSwagCSharp`.	dotnet openapi add file .\OpenApi.json --code-generator
-h	--help	Show help information	dotnet openapi add file --help

Arguments

ARGUMENT	DESCRIPTION	EXAMPLE
source-file	The source to create a reference from. Must be an OpenAPI file.	dotnet openapi add file .*OpenAPI.json*

Add URL

Options

SHORT OPTION	LONG OPTION	DESCRIPTION	EXAMPLE
-p	--updateProject	The project to operate on.	dotnet openapi add url --*updateProject .\Ref.csproj* `https://contoso.com/openapi.json`
-o	--output-file	Where to place the local copy of the OpenAPI file.	dotnet openapi add url `https://contoso.com/openapi.json` *--output-file myclient.json*
-c	--code-generator	The code generator to apply to the reference. Options are `NSwagCSharp` and `NSwagTypeScript`.	dotnet openapi add file .\OpenApi.json --code-generator
-h	--help	Show help information	dotnet openapi add url --help

Arguments

ARGUMENT	DESCRIPTION	EXAMPLE
source-URL	The source to create a reference from. Must be a URL.	dotnet openapi add url `https://contoso.com/openapi.json`

Remove

Removes the OpenAPI reference matching the given filename from the *.csproj* file. When the OpenAPI reference is removed, clients won't be generated. Local *.json* and *.yaml* files are deleted.

Options

SHORT OPTION	LONG OPTION	DESCRIPTION	EXAMPLE
-p	--updateProject	The project to operate on.	dotnet openapi remove --*updateProject .\Ref.csproj* .\OpenAPI.json
-h	--help	Show help information	dotnet openapi remove --help

Arguments

ARGUMENT	DESCRIPTION	EXAMPLE
source-file	The source to remove the reference to.	dotnet openapi remove *.\OpenAPI.json*

Refresh

Refreshes the local version of a file that was downloaded using the latest content from the download URL.

Options

SHORT OPTION	LONG OPTION	DESCRIPTION	EXAMPLE

SHORT OPTION	LONG OPTION	DESCRIPTION	EXAMPLE
-p	--updateProject	The project to operate on.	dotnet openapi refresh --*updateProject .\Ref.csproj* `https://contoso.com/openapi.json`
-h	--help	Show help information	dotnet openapi refresh --help

Arguments

ARGUMENT	DESCRIPTION	EXAMPLE
source-URL	The URL to refresh the reference from.	dotnet openapi refresh `https://contoso.com/openapi.json`

Controller action return types in ASP.NET Core web API

9/22/2020 • 6 minutes to read • Edit Online

By Scott Addie

View or download sample code (how to download)

ASP.NET Core offers the following options for web API controller action return types:

- Specific type
- IActionResult
- ActionResult<T>

- Specific type
- IActionResult

This document explains when it's most appropriate to use each return type.

Specific type

The simplest action returns a primitive or complex data type (for example, `string` or a custom object type). Consider the following action, which returns a collection of custom `Product` objects:

```
[HttpGet]
public List<Product> Get() =>
    _repository.GetProducts();
```

Without known conditions to safeguard against during action execution, returning a specific type could suffice. The preceding action accepts no parameters, so parameter constraints validation isn't needed.

When multiple return types are possible, it's common to mix an ActionResult return type with the primitive or complex return type. Either IActionResult or ActionResult<T> are necessary to accommodate this type of action. Several samples of multiple return types are provided in this document.

Return IEnumerable<T> or IAsyncEnumerable<T>

In ASP.NET Core 2.2 and earlier, returning IEnumerable<T> from an action results in synchronous collection iteration by the serializer. The result is the blocking of calls and a potential for thread pool starvation. To illustrate, imagine that Entity Framework (EF) Core is being used for the web API's data access needs. The following action's return type is synchronously enumerated during serialization:

```
public IEnumerable<Product> GetOnSaleProducts() =>
    _context.Products.Where(p => p.IsOnSale);
```

To avoid synchronous enumeration and blocking waits on the database in ASP.NET Core 2.2 and earlier, invoke `ToListAsync`:

```
public async Task<IEnumerable<Product>> GetOnSaleProducts() =>
    await _context.Products.Where(p => p.IsOnSale).ToListAsync();
```

In ASP.NET Core 3.0 and later, returning `IAsyncEnumerable<T>` from an action:

- No longer results in synchronous iteration.
- Becomes as efficient as returning IEnumerable<T>.

ASP.NET Core 3.0 and later buffers the result of the following action before providing it to the serializer:

```
public IEnumerable<Product> GetOnSaleProducts() =>
    _context.Products.Where(p => p.IsOnSale);
```

Consider declaring the action signature's return type as `IAsyncEnumerable<T>` to guarantee the asynchronous iteration. Ultimately, the iteration mode is based on the underlying concrete type being returned. MVC automatically buffers any concrete type that implements `IAsyncEnumerable<T>`.

Consider the following action, which returns sale-priced product records as `IEnumerable<Product>`:

```
[HttpGet("syncsale")]
public IEnumerable<Product> GetOnSaleProducts()
{
    var products = _repository.GetProducts();

    foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

The `IAsyncEnumerable<Product>` equivalent of the preceding action is:

```
[HttpGet("asyncsale")]
public async IAsyncEnumerable<Product> GetOnSaleProductsAsync()
{
    var products = _repository.GetProductsAsync();

    await foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

Both of the preceding actions are non-blocking as of ASP.NET Core 3.0.

IActionResult type

The IActionResult return type is appropriate when multiple `ActionResult` return types are possible in an action. The `ActionResult` types represent various HTTP status codes. Any non-abstract class deriving from `ActionResult` qualifies as a valid return type. Some common return types in this category are BadRequestResult (400), NotFoundResult (404), and OkObjectResult (200). Alternatively, convenience methods in the ControllerBase class can be used to return `ActionResult` types from an action. For example, `return BadRequest();` is a shorthand form of `return new BadRequestResult();`.

Because there are multiple return types and paths in this type of action, liberal use of the `[ProducesResponseType]`

attribute is necessary. This attribute produces more descriptive response details for web API help pages generated by tools like Swagger. `[ProducesResponseType]` indicates the known types and HTTP status codes to be returned by the action.

Synchronous action

Consider the following synchronous action in which there are two possible return types:

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IActionResult GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return Ok(product);
}
```

```
[HttpGet("{id}")]
[ProducesResponseType(typeof(Product), StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IActionResult GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return Ok(product);
}
```

In the preceding action:

- A 404 status code is returned when the product represented by `id` doesn't exist in the underlying data store. The NotFound convenience method is invoked as shorthand for `return new NotFoundResult();`.
- A 200 status code is returned with the `Product` object when the product does exist. The Ok convenience method is invoked as shorthand for `return new OkObjectResult(product);`.

Asynchronous action

Consider the following asynchronous action in which there are two possible return types:

```
[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> CreateAsync(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    await _repository.AddProductAsync(product);

    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

```
[HttpPost]
[Consumes("application/json")]
[ProducesResponseType(typeof(Product), StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> CreateAsync([FromBody] Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    await _repository.AddProductAsync(product);

    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

In the preceding action:

- A 400 status code is returned when the product description contains "XYZ Widget". The BadRequest convenience method is invoked as shorthand for `return new BadRequestResult();`.
- A 201 status code is generated by the CreatedAtAction convenience method when a product is created. An alternative to calling `CreatedAtAction` is `return new CreatedAtActionResult(nameof(GetById), "Products", new { id = product.Id }, product);`. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

For example, the following model indicates that requests must include the `Name` and `Description` properties. Failure to provide `Name` and `Description` in the request causes model validation to fail.

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Required]
    public string Description { get; set; }
}
```

If the `[ApiController]` attribute in ASP.NET Core 2.1 or later is applied, model validation errors result in a 400 status code. For more information, see Automatic HTTP 400 responses.

ActionResult<T> type

ASP.NET Core 2.1 introduced the ActionResult<T> return type for web API controller actions. It enables you to return a type deriving from ActionResult or return a specific type. `ActionResult<T>` offers the following benefits over the IActionResult type:

- The `[ProducesResponseType]` attribute's `Type` property can be excluded. For example, `[ProducesResponseType(200, Type = typeof(Product))]` is simplified to `[ProducesResponseType(200)]`. The action's expected return type is instead inferred from the `T` in `ActionResult<T>`.
- Implicit cast operators support the conversion of both `T` and `ActionResult` to `ActionResult<T>`. `T` converts to ObjectResult, which means `return new ObjectResult(T);` is simplified to `return T;`.

C# doesn't support implicit cast operators on interfaces. Consequently, conversion of the interface to a concrete type is necessary to use `ActionResult<T>`. For example, use of `IEnumerable` in the following example doesn't

work:

```
[HttpGet]
public ActionResult<IEnumerable<Product>> Get() =>
    _repository.GetProducts();
```

One option to fix the preceding code is to return `_repository.GetProducts().ToList();`.

Most actions have a specific return type. Unexpected conditions can occur during action execution, in which case the specific type isn't returned. For example, an action's input parameter may fail model validation. In such a case, it's common to return the appropriate `ActionResult` type instead of the specific type.

Synchronous action

Consider a synchronous action in which there are two possible return types:

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public ActionResult<Product> GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return product;
}
```

In the preceding action:

- A 404 status code is returned when the product doesn't exist in the database.
- A 200 status code is returned with the corresponding `Product` object when the product does exist. Before ASP.NET Core 2.1, the `return product;` line had to be `return Ok(product);`.

Asynchronous action

Consider an asynchronous action in which there are two possible return types:

```
[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<Product>> CreateAsync(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    await _repository.AddProductAsync(product);

    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

In the preceding action:

- A 400 status code (BadRequest) is returned by the ASP.NET Core runtime when:
  - The `[ApiController]` attribute has been applied and model validation fails.
  - The product description contains "XYZ Widget".

- A 201 status code is generated by the CreatedAtAction method when a product is created. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

Additional resources

- Handle requests with controllers in ASP.NET Core MVC
- Model validation in ASP.NET Core MVC
- ASP.NET Core Web API help pages with Swagger / OpenAPI

JsonPatch in ASP.NET Core web API

9/22/2020 • 13 minutes to read • Edit Online

By Tom Dykstra and Kirk Larkin

This article explains how to handle JSON Patch requests in an ASP.NET Core web API.

Package installation

To enable JSON Patch support in your app, complete the following steps:

1. Install the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` NuGet package.
2. Update the project's `Startup.ConfigureServices` method to call AddNewtonsoftJson. For example:

```
services
    .AddControllersWithViews()
    .AddNewtonsoftJson();
```

`AddNewtonsoftJson` is compatible with the MVC service registration methods:

- AddRazorPages
- AddControllersWithViews
- AddControllers

JSON Patch, AddNewtonsoftJson, and System.Text.Json

`AddNewtonsoftJson` replaces the `System.Text.Json`-based input and output formatters used for formatting **all** JSON content. To add support for JSON Patch using `Newtonsoft.Json`, while leaving the other formatters unchanged, update the project's `Startup.ConfigureServices` method as follows:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
    {
        options.InputFormatters.Insert(0, GetJsonPatchInputFormatter());
    });
}

private static NewtonsoftJsonPatchInputFormatter GetJsonPatchInputFormatter()
{
    var builder = new ServiceCollection()
        .AddLogging()
        .AddMvc()
        .AddNewtonsoftJson()
        .Services.BuildServiceProvider();

    return builder
        .GetRequiredService<IOptions<MvcOptions>>()
        .Value
        .InputFormatters
        .OfType<NewtonsoftJsonPatchInputFormatter>()
        .First();
}
```

The preceding code requires the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` package and the following `using` statements:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Formatters;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using System.Linq;
```

PATCH HTTP request method

The PUT and PATCH methods are used to update an existing resource. The difference between them is that PUT replaces the entire resource, while PATCH specifies only the changes.

JSON Patch

JSON Patch is a format for specifying updates to be applied to a resource. A JSON Patch document has an array of *operations*. Each operation identifies a particular type of change. Examples of such changes include adding an array element or replacing a property value.

For example, the following JSON documents represent a resource, a JSON Patch document for the resource, and the result of applying the Patch operations.

Resource example

```
{
  "customerName": "John",
  "orders": [
    {
      "orderName": "Order0",
      "orderType": null
    },
    {
      "orderName": "Order1",
      "orderType": null
    }
  ]
}
```

JSON patch example

```
[
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "add",
    "path": "/orders/-",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

In the preceding JSON:

- The `op` property indicates the type of operation.
- The `path` property indicates the element to update.
- The `value` property provides the new value.

Resource after patch

Here's the resource after applying the preceding JSON Patch document:

```
{
  "customerName": "Barry",
  "orders": [
    {
      "orderName": "Order0",
      "orderType": null
    },
    {
      "orderName": "Order1",
      "orderType": null
    },
    {
      "orderName": "Order2",
      "orderType": null
    }
  ]
}
```

The changes made by applying a JSON Patch document to a resource are atomic. If any operation in the list fails, no operation in the list is applied.

Path syntax

The path property of an operation object has slashes between levels. For example, `"/address/zipCode"`.

Zero-based indexes are used to specify array elements. The first element of the `addresses` array would be at `/addresses/0`. To `add` to the end of an array, use a hyphen (`-`) rather than an index number: `/addresses/-`.

Operations

The following table shows supported operations as defined in the JSON Patch specification:

OPERATION	NOTES

OPERATION	NOTES
`add`	Add a property or array element. For existing property: set value.
`remove`	Remove a property or array element.
`replace`	Same as `remove` followed by `add` at same location.
`move`	Same as `remove` from source followed by `add` to destination using value from source.
`copy`	Same as `add` to destination using value from source.
`test`	Return success status code if value at `path` = provided `value`.

JSON Patch in ASP.NET Core

The ASP.NET Core implementation of JSON Patch is provided in the Microsoft.AspNetCore.JsonPatch NuGet package.

Action method code

In an API controller, an action method for JSON Patch:

- Is annotated with the `HttpPatch` attribute.
- Accepts a `JsonPatchDocument<T>`, typically with `[FromBody]`.
- Calls `ApplyTo` on the patch document to apply the changes.

Here's an example:

```
[HttpPatch]
public IActionResult JsonPatchWithModelState(
    [FromBody] JsonPatchDocument<Customer> patchDoc)
{
    if (patchDoc != null)
    {
        var customer = CreateCustomer();

        patchDoc.ApplyTo(customer, ModelState);

        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        return new ObjectResult(customer);
    }
    else
    {
        return BadRequest(ModelState);
    }
}
```

This code from the sample app works with the following `Customer` model:

```
public class Customer
{
    public string CustomerName { get; set; }
    public List<Order> Orders { get; set; }
}
```

```
public class Order
{
    public string OrderName { get; set; }
    public string OrderType { get; set; }
}
```

The sample action method:

- Constructs a `Customer`.
- Applies the patch.
- Returns the result in the body of the response.

In a real app, the code would retrieve the data from a store such as a database and update the database after applying the patch.

Model state

The preceding action method example calls an overload of `ApplyTo` that takes model state as one of its parameters. With this option, you can get error messages in responses. The following example shows the body of a 400 Bad Request response for a `test` operation:

```
{
    "Customer": [
        "The current value 'John' at path 'customerName' is not equal to the test value 'Nancy'."
    ]
}
```

Dynamic objects

The following action method example shows how to apply a patch to a dynamic object:

```
[HttpPatch]
public IActionResult JsonPatchForDynamic([FromBody]JsonPatchDocument patch)
{
    dynamic obj = new ExpandoObject();
    patch.ApplyTo(obj);

    return Ok(obj);
}
```

The add operation

- If `path` points to an array element: inserts new element before the one specified by `path`.
- If `path` points to a property: sets the property value.
- If `path` points to a nonexistent location:
  - If the resource to patch is a dynamic object: adds a property.
  - If the resource to patch is a static object: the request fails.

The following sample patch document sets the value of `CustomerName` and adds an `Order` object to the end of the `Orders` array.

```
[
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "add",
    "path": "/orders/-",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

The remove operation

- If `path` points to an array element: removes the element.
- If `path` points to a property:
  - If resource to patch is a dynamic object: removes the property.
  - If resource to patch is a static object:
    - If the property is nullable: sets it to null.
    - If the property is non-nullable, sets it to `default<T>`.

The following sample patch document sets `CustomerName` to null and deletes `Orders[0]`:

```
[
  {
    "op": "remove",
    "path": "/customerName"
  },
  {
    "op": "remove",
    "path": "/orders/0"
  }
]
```

The replace operation

This operation is functionally the same as a `remove` followed by an `add`.

The following sample patch document sets the value of `CustomerName` and replaces `Orders[0]` with a new `Order` object:

```
[
  {
    "op": "replace",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "replace",
    "path": "/orders/0",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

The move operation

- If `path` points to an array element: copies `from` element to location of `path` element, then runs a `remove` operation on the `from` element.
- If `path` points to a property: copies value of `from` property to `path` property, then runs a `remove` operation on the `from` property.
- If `path` points to a nonexistent property:
  - If the resource to patch is a static object: the request fails.
  - If the resource to patch is a dynamic object: copies `from` property to location indicated by `path`, then runs a `remove` operation on the `from` property.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Sets `Orders[0].OrderName` to null.
- Moves `Orders[1]` to before `Orders[0]`.

```
[
  {
    "op": "move",
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "move",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The copy operation

This operation is functionally the same as a `move` operation without the final `remove` step.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Inserts a copy of `Orders[1]` before `Orders[0]`.

```
[
  {
    "op": "copy",
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "copy",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The test operation

If the value at the location indicated by `path` is different from the value provided in `value`, the request fails. In that case, the whole PATCH request fails even if all other operations in the patch document would otherwise succeed.

The `test` operation is commonly used to prevent an update when there's a concurrency conflict.

The following sample patch document has no effect if the initial value of `CustomerName` is "John", because the test fails:

```
[
  {
    "op": "test",
    "path": "/customerName",
    "value": "Nancy"
  },
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  }
]
```

Get the code

View or download sample code. (How to download).

To test the sample, run the app and send HTTP requests with the following settings:

- URL: `http://localhost:{port}/jsonpatch/jsonpatchwithmodelstate`
- HTTP method: `PATCH`
- Header: `Content-Type: application/json-patch+json`
- Body: Copy and paste one of the JSON patch document samples from the *JSON* project folder.

Additional resources

- IETF RFC 5789 PATCH method specification
- IETF RFC 6902 JSON Patch specification
- IETF RFC 6901 JSON Patch path format spec
- JSON Patch documentation. Includes links to resources for creating JSON Patch documents.
- ASP.NET Core JSON Patch source code

This article explains how to handle JSON Patch requests in an ASP.NET Core web API.

PATCH HTTP request method

The PUT and PATCH methods are used to update an existing resource. The difference between them is that PUT replaces the entire resource, while PATCH specifies only the changes.

JSON Patch

JSON Patch is a format for specifying updates to be applied to a resource. A JSON Patch document has an array of *operations*. Each operation identifies a particular type of change, such as add an array element or replace a property value.

For example, the following JSON documents represent a resource, a JSON patch document for the resource, and the result of applying the patch operations.

Resource example

```
{
  "customerName": "John",
  "orders": [
    {
      "orderName": "Order0",
      "orderType": null
    },
    {
      "orderName": "Order1",
      "orderType": null
    }
  ]
}
```

JSON patch example

```
[
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "add",
    "path": "/orders/-",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

In the preceding JSON:

- The `op` property indicates the type of operation.
- The `path` property indicates the element to update.
- The `value` property provides the new value.

Resource after patch

Here's the resource after applying the preceding JSON Patch document:

```
{
  "customerName": "Barry",
  "orders": [
    {
      "orderName": "Order0",
      "orderType": null
    },
    {
      "orderName": "Order1",
      "orderType": null
    },
    {
      "orderName": "Order2",
      "orderType": null
    }
  ]
}
```

The changes made by applying a JSON Patch document to a resource are atomic: if any operation in the list fails, no operation in the list is applied.

Path syntax

The path property of an operation object has slashes between levels. For example, `"/address/zipCode"`.

Zero-based indexes are used to specify array elements. The first element of the `addresses` array would be at `/addresses/0`. To `add` to the end of an array, use a hyphen (-) rather than an index number: `/addresses/-`.

Operations

The following table shows supported operations as defined in the JSON Patch specification:

OPERATION	NOTES
`add`	Add a property or array element. For existing property: set value.
`remove`	Remove a property or array element.
`replace`	Same as `remove` followed by `add` at same location.
`move`	Same as `remove` from source followed by `add` to destination using value from source.
`copy`	Same as `add` to destination using value from source.
`test`	Return success status code if value at `path` = provided `value`.

JsonPatch in ASP.NET Core

The ASP.NET Core implementation of JSON Patch is provided in the Microsoft.AspNetCore.JsonPatch NuGet package. The package is included in the Microsoft.AspnetCore.App metapackage.

Action method code

In an API controller, an action method for JSON Patch:

- Is annotated with the `HttpPatch` attribute.
- Accepts a `JsonPatchDocument<T>`, typically with `[FromBody]`.
- Calls `ApplyTo` on the patch document to apply the changes.

Here's an example:

```
[HttpPatch]
public IActionResult JsonPatchWithModelState(
    [FromBody] JsonPatchDocument<Customer> patchDoc)
{
    if (patchDoc != null)
    {
        var customer = CreateCustomer();

        patchDoc.ApplyTo(customer, ModelState);

        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        return new ObjectResult(customer);
    }
    else
    {
        return BadRequest(ModelState);
    }
}
```

This code from the sample app works with the following `Customer` model.

```
public class Customer
{
    public string CustomerName { get; set; }
    public List<Order> Orders { get; set; }
}
```

```
public class Order
{
    public string OrderName { get; set; }
    public string OrderType { get; set; }
}
```

The sample action method:

- Constructs a `Customer`.
- Applies the patch.
- Returns the result in the body of the response.

In a real app, the code would retrieve the data from a store such as a database and update the database after applying the patch.

Model state

The preceding action method example calls an overload of `ApplyTo` that takes model state as one of its parameters. With this option, you can get error messages in responses. The following example shows the body of a 400 Bad Request response for a `test` operation:

```
{
    "Customer": [
        "The current value 'John' at path 'customerName' is not equal to the test value 'Nancy'."
    ]
}
```

Dynamic objects

The following action method example shows how to apply a patch to a dynamic object.

```
[HttpPatch]
public IActionResult JsonPatchForDynamic([FromBody]JsonPatchDocument patch)
{
    dynamic obj = new ExpandoObject();
    patch.ApplyTo(obj);

    return Ok(obj);
}
```

The add operation

- If `path` points to an array element: inserts new element before the one specified by `path`.
- If `path` points to a property: sets the property value.
- If `path` points to a nonexistent location:
  - If the resource to patch is a dynamic object: adds a property.
  - If the resource to patch is a static object: the request fails.

The following sample patch document sets the value of `CustomerName` and adds an `Order` object to the end of the `Orders` array.

```
[
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "add",
    "path": "/orders/-",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

The remove operation

- If `path` points to an array element: removes the element.
- If `path` points to a property:
  - If resource to patch is a dynamic object: removes the property.
  - If resource to patch is a static object:
    - If the property is nullable: sets it to null.
    - If the property is non-nullable, sets it to `default<T>`.

The following sample patch document sets `CustomerName` to null and deletes `Orders[0]`.

```
[
  {
    "op": "remove",
    "path": "/customerName"
  },
  {
    "op": "remove",
    "path": "/orders/0"
  }
]
```

The replace operation

This operation is functionally the same as a `remove` followed by an `add`.

The following sample patch document sets the value of `CustomerName` and replaces `Orders[0]` with a new `Order` object.

```
[
  {
    "op": "replace",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "replace",
    "path": "/orders/0",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

The move operation

- If `path` points to an array element: copies `from` element to location of `path` element, then runs a `remove` operation on the `from` element.
- If `path` points to a property: copies value of `from` property to `path` property, then runs a `remove` operation on the `from` property.
- If `path` points to a nonexistent property:
  - If the resource to patch is a static object: the request fails.
  - If the resource to patch is a dynamic object: copies `from` property to location indicated by `path`, then runs a `remove` operation on the `from` property.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Sets `Orders[0].OrderName` to null.
- Moves `Orders[1]` to before `Orders[0]`.

```
[
  {
    "op": "move",
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "move",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The copy operation

This operation is functionally the same as a `move` operation without the final `remove` step.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Inserts a copy of `Orders[1]` before `Orders[0]`.

```
[
  {
    "op": "copy",
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "copy",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The test operation

If the value at the location indicated by `path` is different from the value provided in `value`, the request fails. In that case, the whole PATCH request fails even if all other operations in the patch document would otherwise succeed.

The `test` operation is commonly used to prevent an update when there's a concurrency conflict.

The following sample patch document has no effect if the initial value of `CustomerName` is "John", because the test fails:

```
[
  {
    "op": "test",
    "path": "/customerName",
    "value": "Nancy"
  },
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  }
]
```

Get the code

View or download sample code. (How to download).

To test the sample, run the app and send HTTP requests with the following settings:

- URL: `http://localhost:{port}/jsonpatch/jsonpatchwithmodelstate`
- HTTP method: `PATCH`
- Header: `Content-Type: application/json-patch+json`
- Body: Copy and paste one of the JSON patch document samples from the *JSON* project folder.

Additional resources

- IETF RFC 5789 PATCH method specification
- IETF RFC 6902 JSON Patch specification
- IETF RFC 6901 JSON Patch path format spec
- JSON Patch documentation. Includes links to resources for creating JSON Patch documents.
- ASP.NET Core JSON Patch source code

Format response data in ASP.NET Core Web API

9/22/2020 • 7 minutes to read • Edit Online

By Rick Anderson and Steve Smith

ASP.NET Core MVC has support for formatting response data. Response data can be formatted using specific formats or in response to client requested format.

View or download sample code (how to download)

Format-specific Action Results

Some action result types are specific to a particular format, such as JsonResult and ContentResult. Actions can return results that are formatted in a particular format, regardless of client preferences. For example, returning `JsonResult` returns JSON-formatted data. Returning `ContentResult` or a string returns plain-text-formatted string data.

An action isn't required to return any specific type. ASP.NET Core supports any object return value. Results from actions that return objects that are not IActionResult types are serialized using the appropriate IOutputFormatter implementation. For more information, see Controller action return types in ASP.NET Core web API.

The built-in helper method Ok returns JSON-formatted data:

```
// GET: api/authors
[HttpGet]
public ActionResult Get()
{
    return Ok(_authors.List());
}
```

The sample download returns the list of authors. Using the F12 browser developer tools or Postman with the previous code:

- The response header containing **content-type:** `application/json; charset=utf-8` is displayed.
- The request headers are displayed. For example, the `Accept` header. The `Accept` header is ignored by the preceding code.

To return plain text formatted data, use ContentResult and the Content helper:

```
// GET api/authors/about
[HttpGet("About")]
public ContentResult About()
{
    return Content("An API listing authors of docs.asp.net.");
}
```

In the preceding code, the `Content-Type` returned is `text/plain`. Returning a string delivers `Content-Type` of `text/plain`:

```
// GET api/authors/version
[HttpGet("version")]
public string Version()
{
    return "Version 1.0.0";
}
```

For actions with multiple return types, return `IActionResult`. For example, returning different HTTP status codes based on the result of operations performed.

Content negotiation

Content negotiation occurs when the client specifies an Accept header. The default format used by ASP.NET Core is JSON. Content negotiation is:

- Implemented by ObjectResult.
- Built into the status code-specific action results returned from the helper methods. The action results helper methods are based on `ObjectResult`.

When a model type is returned, the return type is `ObjectResult`.

The following action method uses the `Ok` and `NotFound` helper methods:

```
// GET: api/authors/search?namelike=th
[HttpGet("Search")]
public IActionResult Search(string namelike)
{
    var result = _authors.GetByNameSubstring(namelike);
    if (!result.Any())
    {
        return NotFound(namelike);
    }
    return Ok(result);
}
```

By default, ASP.NET Core supports `application/json`, `text/json`, and `text/plain` media types. Tools such as Fiddler or Postman can set the `Accept` request header to specify the return format. When the `Accept` header contains a type the server supports, that type is returned. The next section shows how to add additional formatters.

Controller actions can return POCOs (Plain Old CLR Objects). When a POCO is returned, the runtime automatically creates an `ObjectResult` that wraps the object. The client gets the formatted serialized object. If the object being returned is `null`, a `204 No Content` response is returned.

Returning an object type:

```
// GET api/authors/RickAndMSFT
[HttpGet("{alias}")]
public Author Get(string alias)
{
    return _authors.GetByAlias(alias);
}
```

In the preceding code, a request for a valid author alias returns a `200 OK` response with the author's data. A request for an invalid alias returns a `204 No Content` response.

The Accept header

Content *negotiation* takes place when an `Accept` header appears in the request. When a request contains an accept header, ASP.NET Core:

- Enumerates the media types in the accept header in preference order.
- Tries to find a formatter that can produce a response in one of the formats specified.

If no formatter is found that can satisfy the client's request, ASP.NET Core:

- Returns `406 Not Acceptable` if MvcOptions has been set, or -
- Tries to find the first formatter that can produce a response.

If no formatter is configured for the requested format, the first formatter that can format the object is used. If no `Accept` header appears in the request:

- The first formatter that can handle the object is used to serialize the response.
- There isn't any negotiation taking place. The server is determining what format to return.

If the Accept header contains `*/*`, the Header is ignored unless `RespectBrowserAcceptHeader` is set to true on MvcOptions.

Browsers and content negotiation

Unlike typical API clients, web browsers supply `Accept` headers. Web browser specify many formats, including wildcards. By default, when the framework detects that the request is coming from a browser:

- The `Accept` header is ignored.
- The content is returned in JSON, unless otherwise configured.

This provides a more consistent experience across browsers when consuming APIs.

To configure an app to honor browser accept headers, set RespectBrowserAcceptHeader to `true`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
    {
        options.RespectBrowserAcceptHeader = true; // false by default
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.RespectBrowserAcceptHeader = true; // false by default
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

Configure formatters

Apps that need to support additional formats can add the appropriate NuGet packages and configure support. There are separate formatters for input and output. Input formatters are used by Model Binding. Output formatters are used to format responses. For information on creating a custom formatter, see Custom Formatters.

Add XML format support

XML formatters implemented using XmlSerializer are configured by calling AddXmlSerializerFormatters:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers()
        .AddXmlSerializerFormatters();
}
```

The preceding code serializes results using `XmlSerializer`.

When using the preceding code, controller methods return the appropriate format based on the request's `Accept` header.

Configure System.Text.Json-based formatters

Features for the `System.Text.Json`-based formatters can be configured using `Microsoft.AspNetCore.Mvc.JsonOptions.SerializerOptions`.

```
services.AddControllers().AddJsonOptions(options =>
{
    // Use the default property (Pascal) casing.
    options.JsonSerializerOptions.PropertyNamingPolicy = null;

    // Configure a custom converter.
    options.JsonSerializerOptions.Converters.Add(new MyCustomJsonConverter());
});
```

Output serialization options, on a per-action basis, can be configured using `JsonResult`. For example:

```
public IActionResult Get()
{
    return Json(model, new JsonSerializerOptions
    {
        WriteIndented = true,
    });
}
```

Add Newtonsoft.Json-based JSON format support

Prior to ASP.NET Core 3.0, the default used JSON formatters implemented using the `Newtonsoft.Json` package. In ASP.NET Core 3.0 or later, the default JSON formatters are based on `System.Text.Json`. Support for `Newtonsoft.Json` based formatters and features is available by installing the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` NuGet package and configuring it in `Startup.ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers()
        .AddNewtonsoftJson();
}
```

Some features may not work well with `System.Text.Json`-based formatters and require a reference to the `Newtonsoft.Json`-based formatters. Continue using the `Newtonsoft.Json`-based formatters if the app:

- Uses `Newtonsoft.Json` attributes. For example, `[JsonProperty]` or `[JsonIgnore]`.
- Customizes the serialization settings.
- Relies on features that `Newtonsoft.Json` provides.
- Configures `Microsoft.AspNetCore.Mvc.JsonResult.SerializerSettings`. Prior to ASP.NET Core 3.0, `JsonResult.SerializerSettings` accepts an instance of `JsonSerializerSettings` that is specific to `Newtonsoft.Json`.

- Generates OpenAPI documentation.

Features for the `Newtonsoft.Json`-based formatters can be configured using `Microsoft.AspNetCore.Mvc.MvcNewtonsoftJsonOptions.SerializerSettings`:

```
services.AddControllers().AddNewtonsoftJson(options =>
{
    // Use the default property (Pascal) casing
    options.SerializerSettings.ContractResolver = new DefaultContractResolver();

    // Configure a custom converter
    options.SerializerSettings.Converters.Add(new MyCustomJsonConverter());
});
```

Output serialization options, on a per-action basis, can be configured using `JsonResult`. For example:

```
public IActionResult Get()
{
    return Json(model, new JsonSerializerSettings
    {
        Formatting = Formatting.Indented,
    });
}
```

Add XML format support

XML formatting requires the Microsoft.AspNetCore.Mvc.Formatters.Xml NuGet package.

XML formatters implemented using XmlSerializer are configured by calling AddXmlSerializerFormatters:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
        .AddXmlSerializerFormatters();
}
```

The preceding code serializes results using `XmlSerializer`.

When using the preceding code, controller methods should return the appropriate format based on the request's `Accept` header.

Specify a format

To restrict the response formats, apply the `[Produces]` filter. Like most Filters, `[Produces]` can be applied at the action, controller, or global scope:

```
[ApiController]
[Route("[controller]")]
[Produces("application/json")]
public class WeatherForecastController : ControllerBase
{
```

The preceding `[Produces]` filter:

- Forces all actions within the controller to return JSON-formatted responses.
- If other formatters are configured and the client specifies a different format, JSON is returned.

For more information, see Filters.

Special case formatters

Some special cases are implemented using built-in formatters. By default, `string` return types are formatted as *text/plain* (*text/html* if requested via the `Accept` header). This behavior can be deleted by removing the StringOutputFormatter. Formatters are removed in the `ConfigureServices` method. Actions that have a model object return type return `204 No Content` when returning `null`. This behavior can be deleted by removing the HttpNoContentOutputFormatter. The following code removes the `StringOutputFormatter` and `HttpNoContentOutputFormatter`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
    {
        // requires using Microsoft.AspNetCore.Mvc.Formatters;
        options.OutputFormatters.RemoveType<StringOutputFormatter>();
        options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        // requires using Microsoft.AspNetCore.Mvc.Formatters;
        options.OutputFormatters.RemoveType<StringOutputFormatter>();
        options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

Without the `StringOutputFormatter`, the built-in JSON formatter formats `string` return types. If the built-in JSON formatter is removed and an XML formatter is available, the XML formatter formats `string` return types. Otherwise, `string` return types return `406 Not Acceptable`.

Without the `HttpNoContentOutputFormatter`, null objects are formatted using the configured formatter. For example:

- The JSON formatter returns a response with a body of `null`.
- The XML formatter returns an empty XML element with the attribute `xsi:nil="true"` set.

Response format URL mappings

Clients can request a particular format as part of the URL, for example:

- In the query string or part of the path.
- By using a format-specific file extension such as .xml or .json.

The mapping from request path should be specified in the route the API is using. For example:

```
[Route("api/[controller]")]
[ApiController]
[FormatFilter]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}.{format?}")]
    public Product Get(int id)
    {
```

The preceding route allows the requested format to be specified as an optional file extension. The `[FormatFilter]` attribute checks for the existence of the format value in the `RouteData` and maps the response format to the appropriate formatter when the response is created.

ROUTE	FORMATTER
`/api/products/5`	The default output formatter
`/api/products/5.json`	The JSON formatter (if configured)
`/api/products/5.xml`	The XML formatter (if configured)

Custom formatters in ASP.NET Core Web API

9/22/2020 • 5 minutes to read • Edit Online

By Kirk Larkin and Tom Dykstra.

ASP.NET Core MVC supports data exchange in Web APIs using input and output formatters. Input formatters are used by Model Binding. Output formatters are used to format responses.

The framework provides built-in input and output formatters for JSON and XML. It provides a built-in output formatter for plain text, but doesn't provide an input formatter for plain text.

This article shows how to add support for additional formats by creating custom formatters. For an example of a custom plain text input formatter, see TextPlainInputFormatter on GitHub.

View or download sample code (how to download)

When to use custom formatters

Use a custom formatter to add support for a content type that isn't handled by the built-in formatters.

Overview of how to use a custom formatter

To create a custom formatter:

- For serializing data sent to the client, create an output formatter class.
- For deserializing data received from the client, create an input formatter class.
- Add instances of formatter classes to the `InputFormatters` and `OutputFormatters` collections in MvcOptions.

How to create a custom formatter class

To create a formatter:

- Derive the class from the appropriate base class. The sample app derives from TextOutputFormatter and TextInputFormatter.
- Specify valid media types and encodings in the constructor.
- Override the CanReadType and CanWriteType methods.
- Override the ReadRequestBodyAsync and `WriteResponseBodyAsync` methods.

The following code shows the `VcardOutputFormatter` class from the sample:

```
public class VcardOutputFormatter : TextOutputFormatter
{
    public VcardOutputFormatter()
    {
        SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

        SupportedEncodings.Add(Encoding.UTF8);
        SupportedEncodings.Add(Encoding.Unicode);
    }

    protected override bool CanWriteType(Type type)
    {
        return typeof(Contact).IsAssignableFrom(type) ||
            typeof(IEnumerable<Contact>).IsAssignableFrom(type);
    }

    public override async Task WriteResponseBodyAsync(
        OutputFormatterWriteContext context, Encoding selectedEncoding)
    {
        var httpContext = context.HttpContext;
        var serviceProvider = httpContext.RequestServices;

        var logger = serviceProvider.GetRequiredService<ILogger<VcardOutputFormatter>>();
        var buffer = new StringBuilder();

        if (context.Object is IEnumerable<Contact> contacts)
        {
            foreach (var contact in contacts)
            {
                FormatVcard(buffer, contact, logger);
            }
        }
        else
        {
            FormatVcard(buffer, (Contact)context.Object, logger);
        }

        await httpContext.Response.WriteAsync(buffer.ToString());
    }

    private static void FormatVcard(
        StringBuilder buffer, Contact contact, ILogger logger)
    {
        buffer.AppendLine("BEGIN:VCARD");
        buffer.AppendLine("VERSION:2.1");
        buffer.AppendLine($"N:{contact.LastName};{contact.FirstName}");
        buffer.AppendLine($"FN:{contact.FirstName} {contact.LastName}");
        buffer.AppendLine($"UID:{contact.Id}");
        buffer.AppendLine("END:VCARD");

        logger.LogInformation("Writing {FirstName} {LastName}",
            contact.FirstName, contact.LastName);
    }
}
```

Derive from the appropriate base class

For text media types (for example, vCard), derive from the TextInputFormatter or TextOutputFormatter base class.

```
public class VcardOutputFormatter : TextOutputFormatter
```

For binary types, derive from the InputFormatter or OutputFormatter base class.

Specify valid media types and encodings

In the constructor, specify valid media types and encodings by adding to the `SupportedMediaTypes` and `SupportedEncodings` collections.

```
public VcardOutputFormatter()
{
    SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
```

A formatter class can **not** use constructor injection for its dependencies. For example, `ILogger<VcardOutputFormatter>` cannot be added as a parameter to the constructor. To access services, use the context object that gets passed in to the methods. A code example in this article and the sample show how to do this.

Override CanReadType and CanWriteType

Specify the type to deserialize into or serialize from by overriding the `CanReadType` or `CanWriteType` methods. For example, creating vCard text from a `Contact` type and vice versa.

```
protected override bool CanWriteType(Type type)
{
    return typeof(Contact).IsAssignableFrom(type) ||
        typeof(IEnumerable<Contact>).IsAssignableFrom(type);
}
```

The CanWriteResult method

In some scenarios, `CanWriteResult` must be overridden rather than `CanWriteType`. Use `CanWriteResult` if the following conditions are true:

- The action method returns a model class.
- There are derived classes which might be returned at runtime.
- The derived class returned by the action must be known at runtime.

For example, suppose the action method:

- Signature returns a `Person` type.
- Can return a `Student` or `Instructor` type that derives from `Person`.

For the formatter to handle only `Student` objects, check the type of Object in the context object provided to the `CanWriteResult` method. When the action method returns `IActionResult`:

- It's not necessary to use `CanWriteResult`.
- The `CanWriteType` method receives the runtime type.

Override ReadRequestBodyAsync and WriteResponseBodyAsync

Deserialization or serialization is performed in `ReadRequestBodyAsync` or `WriteResponseBodyAsync`. The following example shows how to get services from the dependency injection container. Services can't be obtained from constructor parameters.

```
public override async Task WriteResponseBodyAsync(
    OutputFormatterWriteContext context, Encoding selectedEncoding)
{
    var httpContext = context.HttpContext;
    var serviceProvider = httpContext.RequestServices;

    var logger = serviceProvider.GetRequiredService<ILogger<VcardOutputFormatter>>();
    var buffer = new StringBuilder();

    if (context.Object is IEnumerable<Contact> contacts)
    {
        foreach (var contact in contacts)
        {
            FormatVcard(buffer, contact, logger);
        }
    }
    else
    {
        FormatVcard(buffer, (Contact)context.Object, logger);
    }

    await httpContext.Response.WriteAsync(buffer.ToString());
}

private static void FormatVcard(
    StringBuilder buffer, Contact contact, ILogger logger)
{
    buffer.AppendLine("BEGIN:VCARD");
    buffer.AppendLine("VERSION:2.1");
    buffer.AppendLine($"N:{contact.LastName};{contact.FirstName}");
    buffer.AppendLine($"FN:{contact.FirstName} {contact.LastName}");
    buffer.AppendLine($"UID:{contact.Id}");
    buffer.AppendLine("END:VCARD");

    logger.LogInformation("Writing {FirstName} {LastName}",
        contact.FirstName, contact.LastName);
}
```

How to configure MVC to use a custom formatter

To use a custom formatter, add an instance of the formatter class to the `InputFormatters` or `OutputFormatters` collection.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
    {
        options.InputFormatters.Insert(0, new VcardInputFormatter());
        options.OutputFormatters.Insert(0, new VcardOutputFormatter());
    });
}
```

```
services.AddMvc(options =>
{
    options.InputFormatters.Insert(0, new VcardInputFormatter());
    options.OutputFormatters.Insert(0, new VcardOutputFormatter());
})
.SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Formatters are evaluated in the order you insert them. The first one takes precedence.

The complete `VcardInputFormatter` class

The following code shows the `VcardInputFormatter` class from the sample:

```
public class VcardInputFormatter : TextInputFormatter
{
    public VcardInputFormatter()
    {
        SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

        SupportedEncodings.Add(Encoding.UTF8);
        SupportedEncodings.Add(Encoding.Unicode);
    }

    protected override bool CanReadType(Type type)
    {
        return type == typeof(Contact);
    }

    public override async Task<InputFormatterResult> ReadRequestBodyAsync(
        InputFormatterContext context, Encoding effectiveEncoding)
    {
        var httpContext = context.HttpContext;
        var serviceProvider = httpContext.RequestServices;

        var logger = serviceProvider.GetRequiredService<ILogger<VcardInputFormatter>>();

        using var reader = new StreamReader(httpContext.Request.Body, effectiveEncoding);
        string nameLine = null;

        try
        {
            await ReadLineAsync("BEGIN:VCARD", reader, context, logger);
            await ReadLineAsync("VERSION:", reader, context, logger);

            nameLine = await ReadLineAsync("N:", reader, context, logger);

            var split = nameLine.Split(";".ToCharArray());
            var contact = new Contact
            {
                LastName = split[0].Substring(2),
                FirstName = split[1]
            };

            await ReadLineAsync("FN:", reader, context, logger);
            await ReadLineAsync("END:VCARD", reader, context, logger);

            logger.LogInformation("nameLine = {nameLine}", nameLine);

            return await InputFormatterResult.SuccessAsync(contact);
        }
        catch
        {
            logger.LogError("Read failed: nameLine = {nameLine}", nameLine);
            return await InputFormatterResult.FailureAsync();
        }
    }

    private static async Task<string> ReadLineAsync(
        string expectedText, StreamReader reader, InputFormatterContext context,
        ILogger logger)
    {
        var line = await reader.ReadLineAsync();

        if (!line.StartsWith(expectedText))
        {
            var errorMessage = $"Looked for '{expectedText}' and got '{line}'";
```

```
            context.ModelState.TryAddModelError(context.ModelName, errorMessage);
            logger.LogError(errorMessage);

            throw new Exception(errorMessage);
        }

        return line;
    }
}
```

Test the app

Run the sample app for this article, which implements basic vCard input and output formatters. The app reads and writes vCards similar to the following:

```
BEGIN:VCARD
VERSION:2.1
N:Davolio;Nancy
FN:Nancy Davolio
END:VCARD
```

To see vCard output, run the app and send a Get request with Accept header `text/vcard` to `https://localhost:5001/api/contacts`.

To add a vCard to the in-memory collection of contacts:

- Send a `Post` request to `/api/contacts` with a tool like Postman.
- Set the `Content-Type` header to `text/vcard`.
- Set `vCard` text in the body, formatted like the preceding example.

Additional resources

- Format response data in ASP.NET Core Web API
- Manage Protobuf references with dotnet-grpc

Use web API analyzers

9/22/2020 • 2 minutes to read • Edit Online

ASP.NET Core 2.2 and later provides an MVC analyzers package intended for use with web API projects. The analyzers work with controllers annotated with ApiControllerAttribute, while building on web API conventions.

The analyzers package notifies you of any controller action that:

- Returns an undeclared status code.
- Returns an undeclared success result.
- Documents a status code that isn't returned.
- Includes an explicit model validation check.

Reference the analyzer package

In ASP.NET Core 3.0 or later, the analyzers are included in the .NET Core SDK. To enable the analyzer in your project, include the `IncludeOpenAPIAnalyzers` property in the project file:

```
<PropertyGroup>
 <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
</PropertyGroup>
```

Package installation

Install the Microsoft.AspNetCore.Mvc.Api.Analyzers NuGet package with one of the following approaches:

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code
- .NET Core CLI

From the **Package Manager Console** window:

- Go to **View** > **Other Windows** > **Package Manager Console**.
- Navigate to the directory in which the *ApiConventions.csproj* file exists.
- Execute the following command:

```
Install-Package Microsoft.AspNetCore.Mvc.Api.Analyzers
```

Analyzers for web API conventions

OpenAPI documents contain status codes and response types that an action may return. In ASP.NET Core MVC, attributes such as ProducesResponseTypeAttribute and ProducesAttribute are used to document an action. ASP.NET Core Web API help pages with Swagger / OpenAPI goes into further detail on documenting your web API.

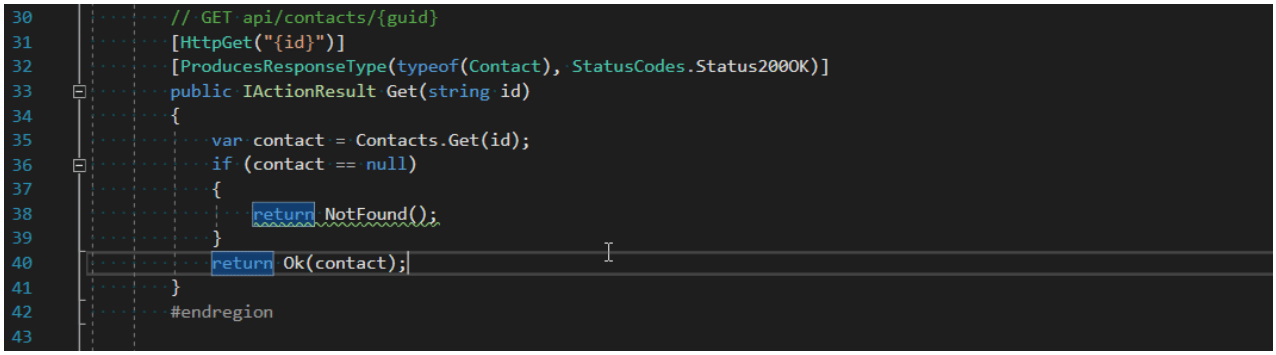
One of the analyzers in the package inspects controllers annotated with ApiControllerAttribute and identifies actions that don't entirely document their responses. Consider the following example:

```
// GET api/contacts/{guid}
[HttpGet("{id}", Name = "GetById")]
[ProducesResponseType(typeof(Contact), StatusCodes.Status200OK)]
public IActionResult Get(string id)
{
    var contact = _contacts.Get(id);

    if (contact == null)
    {
        return NotFound();
    }

    return Ok(contact);
}
```

The preceding action documents the HTTP 200 success return type but doesn't document the HTTP 404 failure status code. The analyzer reports the missing documentation for the HTTP 404 status code as a warning. An option to fix the problem is provided.



Additional resources

- Use web API conventions
- ASP.NET Core Web API help pages with Swagger / OpenAPI
- Create web APIs with ASP.NET Core

Use web API conventions

9/22/2020 • 3 minutes to read • Edit Online

By Pranav Krishnamoorthy and Scott Addie

ASP.NET Core 2.2 and later includes a way to extract common API documentation and apply it to multiple actions, controllers, or all controllers within an assembly. Web API conventions are a substitute for decorating individual actions with `[ProducesResponseType]`.

A convention allows you to:

- Define the most common return types and status codes returned from a specific type of action.
- Identify actions that deviate from the defined standard.

ASP.NET Core MVC 2.2 and later includes a set of default conventions in Microsoft.AspNetCore.Mvc.DefaultApiConventions. The conventions are based on the controller (*ValuesController.cs*) provided in the ASP.NET Core **API** project template. If your actions follow the patterns in the template, you should be successful using the default conventions. If the default conventions don't meet your needs, see Create web API conventions.

At runtime, Microsoft.AspNetCore.Mvc.ApiExplorer understands conventions. `ApiExplorer` is MVC's abstraction to communicate with OpenAPI (also known as Swagger) document generators. Attributes from the applied convention are associated with an action and are included in the action's OpenAPI documentation. API analyzers also understand conventions. If your action is unconventional (for example, it returns a status code that isn't documented by the applied convention), a warning encourages you to document the status code.

View or download sample code (how to download)

Apply web API conventions

Conventions don't compose; each action may be associated with exactly one convention. More specific conventions take precedence over less specific conventions. The selection is non-deterministic when two or more conventions of the same priority apply to an action. The following options exist to apply a convention to an action, from the most specific to the least specific:

1. `Microsoft.AspNetCore.Mvc.ApiConventionMethodAttribute` — Applies to individual actions and specifies the convention type and the convention method that applies.

   In the following example, the default convention type's `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` convention method is applied to the `Update` action:

```
// PUT api/contactsconvention/{guid}
[HttpPut("{id}")]
[ApiConventionMethod(typeof(DefaultApiConventions),
                     nameof(DefaultApiConventions.Put))]
public IActionResult Update(string id, Contact contact)
{
    var contactToUpdate = _contacts.Get(id);

    if (contactToUpdate == null)
    {
        return NotFound();
    }

    _contacts.Update(contact);

    return NoContent();
}
```

The `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` convention method applies the following attributes to the action:

```
[ProducesDefaultResponseType]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
```

For more information on `[ProducesDefaultResponseType]`, see Default Response.

2. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` applied to a controller — Applies the specified convention type to all actions on the controller. A convention method is marked with hints that determine the actions to which the convention method applies. For more information on hints, see Create web API conventions).

   In the following example, the default set of conventions is applied to all actions in *ContactsConventionController*.

   ```
   [ApiController]
   [ApiConventionType(typeof(DefaultApiConventions))]
   [Route("api/[controller]")]
   public class ContactsConventionController : ControllerBase
   {
   ```

3. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` applied to an assembly — Applies the specified convention type to all controllers in the current assembly. As a recommendation, apply assembly-level attributes in the *Startup.cs* file.

   In the following example, the default set of conventions is applied to all controllers in the assembly:

   ```
   [assembly: ApiConventionType(typeof(DefaultApiConventions))]
   namespace ApiConventions
   {
       public class Startup
       {
   ```

Create web API conventions

If the default API conventions don't meet your needs, create your own conventions. A convention is:

- A static type with methods.
- Capable of defining response types and naming requirements on actions.

Response types

These methods are annotated with `[ProducesResponseType]` or `[ProducesDefaultResponseType]` attributes. For example:

```
public static class MyAppConventions
{
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status404NotFound)]
    public static void Find(int id)
    {
    }
}
```

If more specific metadata attributes are absent, applying this convention to an assembly enforces that:

- The convention method applies to any action named `Find`.
- A parameter named `id` is present on the `Find` action.

Naming requirements

The `[ApiConventionNameMatch]` and `[ApiConventionTypeMatch]` attributes can be applied to the convention method that determines the actions to which they apply. For example:

```
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ApiConventionNameMatch(ApiConventionNameMatchBehavior.Prefix)]
public static void Find(
    [ApiConventionNameMatch(ApiConventionNameMatchBehavior.Suffix)]
    int id)
{ }
```

In the preceding example:

- The `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Prefix` option applied to the method indicates that the convention matches any action prefixed with "Find". Examples of matching actions include `Find`, `FindPet`, and `FindById`.
- The `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Suffix` applied to the parameter indicates that the convention matches methods with exactly one parameter ending in the suffix identifier. Examples include parameters such as `id` or `petId`. `ApiConventionTypeMatch` can be similarly applied to types to constrain the parameter type. A `params[]` argument indicates remaining parameters that don't need to be explicitly matched.

Additional resources

- Use web API analyzers
- ASP.NET Core Web API help pages with Swagger / OpenAPI

Handle errors in ASP.NET Core web APIs

9/22/2020 • 7 minutes to read • Edit Online

This article describes how to handle and customize error handling with ASP.NET Core web APIs.

View or download sample code (How to download)

Developer Exception Page

The Developer Exception Page is a useful tool to get detailed stack traces for server errors. It uses DeveloperExceptionPageMiddleware to capture synchronous and asynchronous exceptions from the HTTP pipeline and to generate error responses. To illustrate, consider the following controller action:

```
[HttpGet("{city}")]
public WeatherForecast Get(string city)
{
    if (!string.Equals(city?.TrimEnd(), "Redmond", StringComparison.OrdinalIgnoreCase))
    {
        throw new ArgumentException(
            $"We don't offer a weather forecast for {city}.", nameof(city));
    }

    return GetWeather().First();
}
```

Run the following `curl` command to test the preceding action:

```
curl -i https://localhost:5001/weatherforecast/chicago
```

In ASP.NET Core 3.0 and later, the Developer Exception Page displays a plain-text response if the client doesn't request HTML-formatted output. The following output appears:

```
HTTP/1.1 500 Internal Server Error
Transfer-Encoding: chunked
Content-Type: text/plain
Server: Microsoft-IIS/10.0
X-Powered-By: ASP.NET
Date: Fri, 27 Sep 2019 16:13:16 GMT

System.ArgumentException: We don't offer a weather forecast for chicago. (Parameter 'city')
   at WebApiSample.Controllers.WeatherForecastController.Get(String city) in
C:\working_folder\aspnet\AspNetCore.Docs\aspnetcore\web-api\handle-
errors\samples\3.x\Controllers\WeatherForecastController.cs:line 34
   at lambda_method(Closure , Object , Object[] )
   at Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(Object target, Object[] parameters)
   at
Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.SyncObjectResultExecutor.Execute(IActionResultTyp
eMapper mapper, ObjectMethodExecutor executor, Object controller, Object[] arguments)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
<InvokeActionMethodAsync>g__Logged|12_1(ControllerActionInvoker invoker)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.
<InvokeNextActionFilterAsync>g__Awaited|10_0(ControllerActionInvoker invoker, Task lastTask, State next, Scope
scope, Object state, Boolean isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Rethrow(ActionExecutedContextSealed
context)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope& scope, Object&
state, Boolean& isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync()
--- End of stack trace from previous location where exception was thrown ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.
<InvokeFilterPipelineAsync>g__Awaited|19_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope,
Object state, Boolean isCompleted)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Logged|17_1(ResourceInvoker
invoker)
   at Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g__AwaitRequestTask|6_0(Endpoint endpoint, Task
requestTask, ILogger logger)
   at Microsoft.AspNetCore.Authorization.AuthorizationMiddleware.Invoke(HttpContext context)
   at Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware.Invoke(HttpContext context)

HEADERS
=======
Accept: */*
Host: localhost:44312
User-Agent: curl/7.55.1
```

To display an HTML-formatted response instead, set the `Accept` HTTP request header to the `text/html` media type. For example:

```
curl -i -H "Accept: text/html" https://localhost:5001/weatherforecast/chicago
```

Consider the following excerpt from the HTTP response:

In ASP.NET Core 2.2 and earlier, the Developer Exception Page displays an HTML-formatted response. For example, consider the following excerpt from the HTTP response:

```
HTTP/1.1 500 Internal Server Error
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
Server: Microsoft-IIS/10.0
X-Powered-By: ASP.NET
Date: Fri, 27 Sep 2019 16:55:37 GMT

<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <meta charset="utf-8" />
        <title>Internal Server Error</title>
        <style>
            body {
    font-family: 'Segoe UI', Tahoma, Arial, Helvetica, sans-serif;
    font-size: .813em;
    color: #222;
    background-color: #fff;
}
```

The HTML-formatted response becomes useful when testing via tools like Postman. The following screen capture shows both the plain-text and the HTML-formatted responses in Postman:

WARNING

Enable the Developer Exception Page **only when the app is running in the Development environment**. You don't want to share detailed exception information publicly when the app runs in production. For more information on configuring environments, see Use multiple environments in ASP.NET Core.

Exception handler

In non-development environments, Exception Handling Middleware can be used to produce an error payload:

1. In `Startup.Configure`, invoke UseExceptionHandler to use the middleware:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }

    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

2. Configure a controller action to respond to the `/error` route:

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error")]
    public IActionResult Error() => Problem();
}
```

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error")]
    public ActionResult Error([FromServices] IHostingEnvironment webHostEnvironment)
    {
        var feature = HttpContext.Features.Get<IExceptionHandlerPathFeature>();
        var ex = feature?.Error;
        var isDev = webHostEnvironment.IsDevelopment();
        var problemDetails = new ProblemDetails
        {
            Status = (int)HttpStatusCode.InternalServerError,
            Instance = feature?.Path,
            Title = isDev ? $"{ex.GetType().Name}: {ex.Message}" : "An error occurred.",
            Detail = isDev ? ex.StackTrace : null,
        };

        return StatusCode(problemDetails.Status.Value, problemDetails);
    }
}
```

The preceding `Error` action sends an RFC 7807-compliant payload to the client.

Exception Handling Middleware can also provide more detailed content-negotiated output in the local development environment. Use the following steps to produce a consistent payload format across development and production environments:

1. In `Startup.Configure`, register environment-specific Exception Handling Middleware instances:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseExceptionHandler("/error-local-development");
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseExceptionHandler("/error-local-development");
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
}
```

   In the preceding code, the middleware is registered with:

   - A route of `/error-local-development` in the Development environment.
   - A route of `/error` in environments that aren't Development.

2. Apply attribute routing to controller actions:

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error-local-development")]
    public IActionResult ErrorLocalDevelopment(
        [FromServices] IWebHostEnvironment webHostEnvironment)
    {
        if (webHostEnvironment.EnvironmentName != "Development")
        {
            throw new InvalidOperationException(
                "This shouldn't be invoked in non-development environments.");
        }

        var context = HttpContext.Features.Get<IExceptionHandlerFeature>();

        return Problem(
            detail: context.Error.StackTrace,
            title: context.Error.Message);
    }

    [Route("/error")]
    public IActionResult Error() => Problem();
}
```

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error-local-development")]
    public IActionResult ErrorLocalDevelopment(
        [FromServices] IHostingEnvironment webHostEnvironment)
    {
        if (!webHostEnvironment.IsDevelopment())
        {
            throw new InvalidOperationException(
                "This shouldn't be invoked in non-development environments.");
        }

        var feature = HttpContext.Features.Get<IExceptionHandlerPathFeature>();
        var ex = feature?.Error;

        var problemDetails = new ProblemDetails
        {
            Status = (int)HttpStatusCode.InternalServerError,
            Instance = feature?.Path,
            Title = ex.GetType().Name,
            Detail = ex.StackTrace,
        };

        return StatusCode(problemDetails.Status.Value, problemDetails);
    }

    [Route("/error")]
    public ActionResult Error(
        [FromServices] IHostingEnvironment webHostEnvironment)
    {
        var feature = HttpContext.Features.Get<IExceptionHandlerPathFeature>();
        var ex = feature?.Error;
        var isDev = webHostEnvironment.IsDevelopment();
        var problemDetails = new ProblemDetails
        {
            Status = (int)HttpStatusCode.InternalServerError,
            Instance = feature?.Path,
            Title = isDev ? $"{ex.GetType().Name}: {ex.Message}" : "An error occurred.",
            Detail = isDev ? ex.StackTrace : null,
        };

        return StatusCode(problemDetails.Status.Value, problemDetails);
    }
}
```

Use exceptions to modify the response

The contents of the response can be modified from outside of the controller. In ASP.NET 4.x Web API, one way to do this was using the HttpResponseException type. ASP.NET Core doesn't include an equivalent type. Support for `HttpResponseException` can be added with the following steps:

1. Create a well-known exception type named `HttpResponseException`:

```
public class HttpResponseException : Exception
{
    public int Status { get; set; } = 500;

    public object Value { get; set; }
}
```

2. Create an action filter named `HttpResponseExceptionFilter`:

```
public class HttpResponseExceptionFilter : IActionFilter, IOrderedFilter
{
    public int Order { get; set; } = int.MaxValue - 10;

    public void OnActionExecuting(ActionExecutingContext context) { }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        if (context.Exception is HttpResponseException exception)
        {
            context.Result = new ObjectResult(exception.Value)
            {
                StatusCode = exception.Status,
            };
            context.ExceptionHandled = true;
        }
    }
}
```

In the preceding filter, the magic number 10 is subtracted from the maximum integer value. Subtracting this number allows other filters to run at the very end of the pipeline.

3. In `Startup.ConfigureServices`, add the action filter to the filters collection:

```
services.AddControllers(options =>
    options.Filters.Add(new HttpResponseExceptionFilter()));
```

```
services.AddMvc(options =>
        options.Filters.Add(new HttpResponseExceptionFilter()))
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

```
services.AddMvc(options =>
        options.Filters.Add(new HttpResponseExceptionFilter()))
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

Validation failure error response

For web API controllers, MVC responds with a ValidationProblemDetails response type when model validation fails. MVC uses the results of InvalidModelStateResponseFactory to construct the error response for a validation failure. The following example uses the factory to change the default response type to SerializableError in `Startup.ConfigureServices`:

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            var result = new BadRequestObjectResult(context.ModelState);

            // TODO: add `using System.Net.Mime;` to resolve MediaTypeNames
            result.ContentTypes.Add(MediaTypeNames.Application.Json);
            result.ContentTypes.Add(MediaTypeNames.Application.Xml);

            return result;
        };
    });
```

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            var result = new BadRequestObjectResult(context.ModelState);

            // TODO: add `using System.Net.Mime;` to resolve MediaTypeNames
            result.ContentTypes.Add(MediaTypeNames.Application.Json);
            result.ContentTypes.Add(MediaTypeNames.Application.Xml);

            return result;
        };
    });
```

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

services.Configure<ApiBehaviorOptions>(options =>
{
    options.InvalidModelStateResponseFactory = context =>
    {
        var result = new BadRequestObjectResult(context.ModelState);

        // TODO: add `using using System.Net.Mime;` to resolve MediaTypeNames
        result.ContentTypes.Add(MediaTypeNames.Application.Json);
        result.ContentTypes.Add(MediaTypeNames.Application.Xml);

        return result;
    };
});
```

Client error response

An *error result* is defined as a result with an HTTP status code of 400 or higher. For web API controllers, MVC transforms an error result to a result with ProblemDetails.

> **IMPORTANT**
> ASP.NET Core 2.1 generates a problem details response that's nearly RFC 7807-compliant. If 100 percent compliance is important, upgrade the project to ASP.NET Core 2.2 or later.

The error response can be configured in one of the following ways:

1. Implement ProblemDetailsFactory
2. Use ApiBehaviorOptions.ClientErrorMapping

Implement `ProblemDetailsFactory`

MVC uses Microsoft.AspNetCore.Mvc.Infrastructure.ProblemDetailsFactory to produce all instances of ProblemDetails and ValidationProblemDetails. This includes client error responses, validation failure error responses, and the ControllerBase.Problem and ControllerBase.ValidationProblem helper methods.

To customize the problem details response, register a custom implementation of ProblemDetailsFactory in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection serviceCollection)
{
    services.AddControllers();
    services.AddTransient<ProblemDetailsFactory, CustomProblemDetailsFactory>();
}
```

The error response can be configured as outlined in the Use ApiBehaviorOptions.ClientErrorMapping section.

Use ApiBehaviorOptions.ClientErrorMapping

Use the ClientErrorMapping property to configure the contents of the `ProblemDetails` response. For example, the following code in `Startup.ConfigureServices` updates the `type` property for 404 responses:

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
```

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[404].Link =
            "https://httpstatuses.com/404";
    });
```

Test web APIs with the HTTP REPL

9/22/2020 • 19 minutes to read • Edit Online

By Scott Addie

The HTTP Read-Eval-Print Loop (REPL) is:

- A lightweight, cross-platform command-line tool that's supported everywhere .NET Core is supported.
- Used for making HTTP requests to test ASP.NET Core web APIs (and non-ASP.NET Core web APIs) and view their results.
- Capable of testing web APIs hosted in any environment, including localhost and Azure App Service.

The following HTTP verbs are supported:

- DELETE
- GET
- HEAD
- OPTIONS
- PATCH
- POST
- PUT

To follow along, view or download the sample ASP.NET Core web API (how to download).

Prerequisites

- .NET Core 2.1 SDK or later

Installation

To install the HTTP REPL, run the following command:

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

A .NET Core Global Tool is installed from the Microsoft.dotnet-httprepl NuGet package.

Usage

After successful installation of the tool, run the following command to start the HTTP REPL:

```
httprepl
```

To view the available HTTP REPL commands, run one of the following commands:

```
httprepl -h
```

```
httprepl --help
```

The following output is displayed:

```
Usage:
  httprepl [<BASE_ADDRESS>] [options]

Arguments:
  <BASE_ADDRESS> - The initial base address for the REPL.

Options:
  -h|--help - Show help information.

Once the REPL starts, these commands are valid:

Setup Commands:
Use these commands to configure the tool for your API server

connect        Configures the directory structure and base address of the api server
set header     Sets or clears a header for all requests. e.g. `set header content-type application/json`

HTTP Commands:
Use these commands to execute requests against your application.

GET            get - Issues a GET request
POST           post - Issues a POST request
PUT            put - Issues a PUT request
DELETE         delete - Issues a DELETE request
PATCH          patch - Issues a PATCH request
HEAD           head - Issues a HEAD request
OPTIONS        options - Issues a OPTIONS request

Navigation Commands:
The REPL allows you to navigate your URL space and focus on specific APIs that you are working on.

set base       Set the base URI. e.g. `set base http://locahost:5000`
ls             Show all endpoints for the current path
cd             Append the given directory to the currently selected path, or move up a path when using `cd ..`

Shell Commands:
Use these commands to interact with the REPL shell.

clear          Removes all text from the shell
echo [on/off]  Turns request echoing on or off, show the request that was made when using request commands
exit           Exit the shell

REPL Customization Commands:
Use these commands to customize the REPL behavior.

pref [get/set] Allows viewing or changing preferences, e.g. 'pref set editor.command.default 'C:\\Program
Files\\Microsoft VS Code\\Code.exe'`
run            Runs the script at the given path. A script is a set of commands that can be typed with one
command per line
ui             Displays the Swagger UI page, if available, in the default browser

Use `help <COMMAND>` for more detail on an individual command. e.g. `help get`.
For detailed tool info, see https://aka.ms/http-repl-doc.
```

The HTTP REPL offers command completion. Pressing the Tab key iterates through the list of commands that complete the characters or API endpoint that you typed. The following sections outline the available CLI commands.

Connect to the web API

Connect to a web API by running the following command:

```
httprepl <ROOT URI>
```

`<ROOT URI>` is the base URI for the web API. For example:

```
httprepl https://localhost:5001
```

Alternatively, run the following command at any time while the HTTP REPL is running:

```
connect <ROOT URI>
```

For example:

```
(Disconnected)~ connect https://localhost:5001
```

Manually point to the Swagger document for the web API

The connect command above will attempt to find the Swagger document automatically. If for some reason it is unable to do so, you can specify the URI of the Swagger document for the web API by using the `--swagger` option:

```
connect <ROOT URI> --swagger <SWAGGER URI>
```

For example:

```
(Disconnected)~ connect https://localhost:5001 --swagger /swagger/v1/swagger.json
```

Navigate the web API

View available endpoints

To list the different endpoints (controllers) at the current path of the web API address, run the `ls` or `dir` command:

```
https://localhot:5001/~ ls
```

The following output format is displayed:

```
.       []
Fruits  [get|post]
People  [get|post]

https://localhost:5001/~
```

The preceding output indicates that there are two controllers available: `Fruits` and `People`. Both controllers support parameterless HTTP GET and POST operations.

Navigating into a specific controller reveals more detail. For example, the following command's output shows the `Fruits` controller also supports HTTP GET, PUT, and DELETE operations. Each of these operations expects an `id` parameter in the route:

```
https://localhost:5001/fruits~ ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/fruits~
```

Alternatively, run the `ui` command to open the web API's Swagger UI page in a browser. For example:

```
https://localhost:5001/~ ui
```

Navigate to an endpoint

To navigate to a different endpoint on the web API, run the `cd` command:

```
https://localhost:5001/~ cd people
```

The path following the `cd` command is case insensitive. The following output format is displayed:

```
/people    [get|post]

https://localhost:5001/people~
```

Customize the HTTP REPL

The HTTP REPL's default colors can be customized. Additionally, a default text editor can be defined. The HTTP REPL preferences are persisted across the current session and are honored in future sessions. Once modified, the preferences are stored in the following file:

- Linux
- macOS
- Windows

%HOME%/.httpreplprefs

The *.httpreplprefs* file is loaded on startup and not monitored for changes at runtime. Manual modifications to the file take effect only after restarting the tool.

View the settings

To view the available settings, run the `pref get` command. For example:

```
https://localhost:5001/~ pref get
```

The preceding command displays the available key-value pairs:

```
colors.json=Green
colors.json.arrayBrace=BoldCyan
colors.json.comma=BoldYellow
colors.json.name=BoldMagenta
colors.json.nameSeparator=BoldWhite
colors.json.objectBrace=Cyan
colors.protocol=BoldGreen
colors.status=BoldYellow
```

Set color preferences

Response colorization is currently supported for JSON only. To customize the default HTTP REPL tool coloring, locate the key corresponding to the color to be changed. For instructions on how to find the keys, see the View the settings section. For example, change the `colors.json` key value from `Green` to `White` as follows:

```
https://localhost:5001/people~ pref set colors.json White
```

Only the allowed colors may be used. Subsequent HTTP requests display output with the new coloring.

When specific color keys aren't set, more generic keys are considered. To demonstrate this fallback behavior, consider the following example:

- If `colors.json.name` doesn't have a value, `colors.json.string` is used.
- If `colors.json.string` doesn't have a value, `colors.json.literal` is used.
- If `colors.json.literal` doesn't have a value, `colors.json` is used.
- If `colors.json` doesn't have a value, the command shell's default text color (`AllowedColors.None`) is used.

Set indentation size

Response indentation size customization is currently supported for JSON only. The default size is two spaces. For example:

```
[
  {
    "id": 1,
    "name": "Apple"
  },
  {
    "id": 2,
    "name": "Orange"
  },
  {
    "id": 3,
    "name": "Strawberry"
  }
]
```

To change the default size, set the `formatting.json.indentSize` key. For example, to always use four spaces:

```
pref set formatting.json.indentSize 4
```

Subsequent responses honor the setting of four spaces:

```
[
    {
        "id": 1,
        "name": "Apple"
    },
    {
        "id": 2,
        "name": "Orange"
    },
    {
        "id": 3,
        "name": "Strawberry"
    }
]
```

Set the default text editor

By default, the HTTP REPL has no text editor configured for use. To test web API methods requiring an HTTP request body, a default text editor must be set. The HTTP REPL tool launches the configured text editor for the sole purpose of composing the request body. Run the following command to set your preferred text editor as the default:

```
pref set editor.command.default "<EXECUTABLE>"
```

In the preceding command, `<EXECUTABLE>` is the full path to the text editor's executable file. For example, run the following command to set Visual Studio Code as the default text editor:

- Linux
- macOS
- Windows

```
pref set editor.command.default "/usr/bin/code"
```

To launch the default text editor with specific CLI arguments, set the `editor.command.default.arguments` key. For example, assume Visual Studio Code is the default text editor and that you always want the HTTP REPL to open Visual Studio Code in a new session with extensions disabled. Run the following command:

```
pref set editor.command.default.arguments "--disable-extensions --new-window"
```

Set the Swagger search paths

By default, the HTTP REPL has a set of relative paths that it uses to find the Swagger document when executing the `connect` command without the `--swagger` option. These relative paths are combined with the root and base paths specified in the `connect` command. The default relative paths are:

- *swagger.json*
- *swagger/v1/swagger.json*
- */swagger.json*
- */swagger/v1/swagger.json*

To use a different set of search paths in your environment, set the `swagger.searchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

```
pref set swagger.searchPaths "swagger/v2/swagger.json|swagger/v3/swagger.json"
```

Test HTTP GET requests

Synopsis

```
get <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--response:body] [--response:headers] [-s|--
streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

The following options are available for the `get` command:

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

  - `{header}={value}`
  - `{header}:{value}`

- `--response`

  Specifies a file to which the entire HTTP response (including headers and body) should be written. For example, `--response "C:\response.txt"`. The file is created if it doesn't exist.

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP GET request:

1. Run the `get` command on an endpoint that supports it:

   ```
   https://localhost:5001/people~ get
   ```

   The preceding command displays the following output format:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 21 Jun 2019 03:38:45 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "name": "Scott Hunter"
  },
  {
    "id": 2,
    "name": "Scott Hanselman"
  },
  {
    "id": 3,
    "name": "Scott Guthrie"
  }
]


https://localhost:5001/people~
```

2. Retrieve a specific record by passing a parameter to the `get` command:

```
https://localhost:5001/people~ get 2
```

The preceding command displays the following output format:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 21 Jun 2019 06:17:57 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 2,
    "name": "Scott Hanselman"
  }
]


https://localhost:5001/people~
```

Test HTTP POST requests

Synopsis

```
post <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

  - `{header}={value}`
  - `{header}:{value}`

- `--response`

  Specifies a file to which the entire HTTP response (including headers and body) should be written. For example, `--response "C:\response.txt"`. The file is created if it doesn't exist.

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{"id":2,"name":"Cherry"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Example

To issue an HTTP POST request:

1. Run the `post` command on an endpoint that supports it:

   ```
   https://localhost:5001/people~ post -h Content-Type=application/json
   ```

   In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

   ```
   {
     "id": 0,
     "name": ""
   }
   ```

> **TIP**
> To set the default text editor, see the Set the default text editor section.

2. Modify the JSON template to satisfy model validation requirements:

```
{
  "id": 0,
  "name": "Scott Addie"
}
```

3. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Thu, 27 Jun 2019 21:24:18 GMT
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
  "name": "Scott Addie"
}


https://localhost:5001/people~
```

Test HTTP PUT requests

Synopsis

```
put <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

  - `{header}={value}`
  - `{header}:{value}`

- `--response`

  Specifies a file to which the entire HTTP response (including headers and body) should be written. For example, `--response "C:\response.txt"`. The file is created if it doesn't exist.

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{"id":2,"name":"Cherry"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Example

To issue an HTTP PUT request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   ```
   https://localhost:5001/fruits~ get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
     {
       "id": 2,
       "data": "Orange"
     },
     {
       "id": 3,
       "data": "Strawberry"
     }
   ]
   ```

2. Run the `put` command on an endpoint that supports it:

   ```
   https://localhost:5001/fruits~ put 2 -h Content-Type=application/json
   ```

   In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

```
{
  "id": 0,
  "name": ""
}
```

> **TIP**
> To set the default text editor, see the Set the default text editor section.

3. Modify the JSON template to satisfy model validation requirements:

```
{
  "id": 2,
  "name": "Cherry"
}
```

4. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

```
[main 2019-06-28T17:27:01.805Z] update#setState idle
HTTP/1.1 204 No Content
Date: Fri, 28 Jun 2019 17:28:21 GMT
Server: Kestrel
```

5. *Optional*: Issue a `get` command to see the modifications. For example, if you typed "Cherry" in the text editor, a `get` returns the following:

```
https://localhost:5001/fruits~ get
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sat, 22 Jun 2019 00:08:20 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Cherry"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]


https://localhost:5001/fruits~
```

Test HTTP DELETE requests

Synopsis

```
delete <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

  - `{header}={value}`
  - `{header}:{value}`

- `--response`

  Specifies a file to which the entire HTTP response (including headers and body) should be written. For example, `--response "C:\response.txt"`. The file is created if it doesn't exist.

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP DELETE request:

1. *Optional*: Run the `get` command to view the data before modifying it:

```
https://localhost:5001/fruits~ get
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sat, 22 Jun 2019 00:07:32 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `delete` command on an endpoint that supports it:

   ```
   https://localhost:5001/fruits~ delete 2
   ```

   The preceding command displays the following output format:

   ```
   HTTP/1.1 204 No Content
   Date: Fri, 28 Jun 2019 17:36:42 GMT
   Server: Kestrel
   ```

3. *Optional*: Issue a `get` command to see the modifications. In this example, a `get` returns the following:

   ```
   https://localhost:5001/fruits~ get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:16:30 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
     {
       "id": 3,
       "data": "Strawberry"
     }
   ]


   https://localhost:5001/fruits~
   ```

Test HTTP PATCH requests

Synopsis

```
patch <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

  - `{header}={value}`
  - `{header}:{value}`

- `--response`

  Specifies a file to which the entire HTTP response (including headers and body) should be written. For example, `--response "C:\response.txt"`. The file is created if it doesn't exist.

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{"id":2,"name":"Cherry"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Test HTTP HEAD requests

Synopsis

```
head <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

  - `{header}={value}`
  - `{header}:{value}`

- `--response`

  Specifies a file to which the entire HTTP response (including headers and body) should be written. For example, `--response "C:\response.txt"`. The file is created if it doesn't exist.

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Test HTTP OPTIONS requests

Synopsis

```
options <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

  - `{header}={value}`
  - `{header}:{value}`

- `--response`

Specifies a file to which the entire HTTP response (including headers and body) should be written. For example, `--response "C:\response.txt"`. The file is created if it doesn't exist.

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Set HTTP request headers

To set an HTTP request header, use one of the following approaches:

- Set inline with the HTTP request. For example:

  ```
  https://localhost:5001/people~ post -h Content-Type=application/json
  ```

  With the preceding approach, each distinct HTTP request header requires its own `-h` option.

- Set before sending the HTTP request. For example:

  ```
  https://localhost:5001/people~ set header Content-Type application/json
  ```

  When setting the header before sending a request, the header remains set for the duration of the command shell session. To clear the header, provide an empty value. For example:

  ```
  https://localhost:5001/people~ set header Content-Type
  ```

Test secured endpoints

The HTTP REPL supports the testing of secured endpoints in two ways: via the default credentials of the logged in user or through the use of HTTP request headers.

Default credentials

Consider a scenario in which the web API you're testing is hosted in IIS and is secured with Windows authentication. You want the credentials of the user running the tool to flow across to the HTTP endpoints being tested. To pass the default credentials of the logged in user:

1. Set the `httpClient.useDefaultCredentials` preference to `true`:

   ```
   pref set httpClient.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

HTTP request headers

Examples of supported authentication and authorization schemes include basic authentication, JWT bearer tokens,

and digest authentication. For example, you can send a bearer token to an endpoint with the following command:

```
set header Authorization "bearer <TOKEN VALUE>"
```

To access an Azure-hosted endpoint or to use the Azure REST API, you need a bearer token. Use the following steps to obtain a bearer token for your Azure subscription via the Azure CLI. The HTTP REPL sets the bearer token in an HTTP request header and retrieves a list of Azure App Service Web Apps.

1. Log in to Azure:

   ```
   az login
   ```

2. Get your subscription ID with the following command:

   ```
   az account show --query id
   ```

3. Copy your subscription ID and run the following command:

   ```
   az account set --subscription "<SUBSCRIPTION ID>"
   ```

4. Get your bearer token with the following command:

   ```
   az account get-access-token --query accessToken
   ```

5. Connect to the Azure REST API via the HTTP REPL:

   ```
   httprepl https://management.azure.com
   ```

6. Set the `Authorization` HTTP request header:

   ```
   https://management.azure.com/> set header Authorization "bearer <ACCESS TOKEN>"
   ```

7. Navigate to the subscription:

   ```
   https://management.azure.com/> cd subscriptions/<SUBSCRIPTION ID>
   ```

8. Get a list of your subscription's Azure App Service Web Apps:

   ```
   https://management.azure.com/subscriptions/{SUBSCRIPTION ID}> get providers/Microsoft.Web/sites?api-version=2016-08-01
   ```

   The following response is displayed:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Length: 35948
Content-Type: application/json; charset=utf-8
Date: Thu, 19 Sep 2019 23:04:03 GMT
Expires: -1
Pragma: no-cache
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
x-ms-correlation-request-id: <em>xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx</em>
x-ms-original-request-ids: <em>xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx;xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx</em>
x-ms-ratelimit-remaining-subscription-reads: 11999
x-ms-request-id: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
x-ms-routing-request-id: WESTUS:xxxxxxxxxxxxxxxx:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx
{
  "value": [
    <AZURE RESOURCES LIST>
  ]
}
```

Toggle HTTP request display

By default, display of the HTTP request being sent is suppressed. It's possible to change the corresponding setting for the duration of the command shell session.

Enable request display

View the HTTP request being sent by running the `echo on` command. For example:

```
https://localhost:5001/people~ echo on
Request echoing is on
```

Subsequent HTTP requests in the current session display the request headers. For example:

```
https://localhost:5001/people~ post

[main 2019-06-28T18:50:11.930Z] update#setState idle
Request to https://localhost:5001...

POST /people HTTP/1.1
Content-Length: 41
Content-Type: application/json
User-Agent: HTTP-REPL

{
  "id": 0,
  "name": "Scott Addie"
}

Response from https://localhost:5001...

HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Fri, 28 Jun 2019 18:50:21 GMT
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
  "name": "Scott Addie"
}


https://localhost:5001/people~
```

Disable request display

Suppress display of the HTTP request being sent by running the `echo off` command. For example:

```
https://localhost:5001/people~ echo off
Request echoing is off
```

Run a script

If you frequently execute the same set of HTTP REPL commands, consider storing them in a text file. Commands in the file take the same form as those executed manually on the command line. The commands can be executed in a batched fashion using the `run` command. For example:

1. Create a text file containing a set of newline-delimited commands. To illustrate, consider a *people-script.txt* file containing the following commands:

   ```
   set base https://localhost:5001
   ls
   cd People
   ls
   get 1
   ```

2. Execute the `run` command, passing in the text file's path. For example:

   ```
   https://localhost:5001/~ run C:\http-repl-scripts\people-script.txt
   ```

   The following output appears:

```
https://localhost:5001/~ set base https://localhost:5001
Using swagger metadata from https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/~ ls
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/~ cd People
/People    [get|post]

https://localhost:5001/People~ ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/People~ get 1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 12 Jul 2019 19:20:10 GMT
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "name": "Scott Hunter"
}


https://localhost:5001/People~
```

Clear the output

To remove all output written to the command shell by the HTTP REPL tool, run the `clear` or `cls` command. To illustrate, imagine the command shell contains the following output:

```
httprepl https://localhost:5001
(Disconnected)~ set base "https://localhost:5001"
Using swagger metadata from https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/~ ls
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/~
```

Run the following command to clear the output:

```
https://localhost:5001/~ clear
```

After running the preceding command, the command shell contains only the following output:

```
https://localhost:5001/~
```

Additional resources

- REST API requests

- HTTP REPL GitHub repository

Introduction to ASP.NET Core SignalR

9/22/2020 • 2 minutes to read • Edit Online

What is SignalR?

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly.

Good candidates for SignalR:

- Apps that require high frequency updates from the server. Examples are gaming, social networks, voting, auction, maps, and GPS apps.
- Dashboards and monitoring apps. Examples include company dashboards, instant sales updates, or travel alerts.
- Collaborative apps. Whiteboard apps and team meeting software are examples of collaborative apps.
- Apps that require notifications. Social networks, email, chat, games, travel alerts, and many other apps use notifications.

SignalR provides an API for creating server-to-client remote procedure calls (RPC). The RPCs call JavaScript functions on clients from server-side .NET Core code.

Here are some features of SignalR for ASP.NET Core:

- Handles connection management automatically.
- Sends messages to all connected clients simultaneously. For example, a chat room.
- Sends messages to specific clients or groups of clients.
- Scales to handle increasing traffic.

The source is hosted in a SignalR repository on GitHub.

Transports

SignalR supports the following techniques for handling real-time communication (in order of graceful fallback):

- WebSockets
- Server-Sent Events
- Long Polling

SignalR automatically chooses the best transport method that is within the capabilities of the server and client.

Hubs

SignalR uses *hubs* to communicate between clients and servers.

A hub is a high-level pipeline that allows a client and server to call methods on each other. SignalR handles the dispatching across machine boundaries automatically, allowing clients to call methods on the server and vice versa. You can pass strongly-typed parameters to methods, which enables model binding. SignalR provides two built-in hub protocols: a text protocol based on JSON and a binary protocol based on MessagePack. MessagePack generally creates smaller messages compared to JSON. Older browsers must support XHR level 2 to provide MessagePack protocol support.

Hubs call client-side code by sending messages that contain the name and parameters of the client-side method. Objects sent as method parameters are deserialized using the configured protocol. The client tries to match the name to a method in the client-side code. When the client finds a match, it calls the method and passes to it the deserialized parameter data.

Additional resources

- Get started with SignalR for ASP.NET Core
- Supported Platforms
- Hubs
- JavaScript client

ASP.NET Core SignalR supported platforms

9/22/2020 • 2 minutes to read • Edit Online

Server system requirements

SignalR for ASP.NET Core supports any server platform that ASP.NET Core supports.

JavaScript client

The JavaScript client runs on NodeJS 8 and later versions and the following browsers:

BROWSER	VERSION
Microsoft Edge	Current†
Mozilla Firefox	Current†
Google Chrome; includes Android	Current†
Safari; includes iOS	Current†
Microsoft Internet Explorer	11

†*Current* refers to the latest version of the browser.

.NET client

The .NET client runs on any platform supported by ASP.NET Core. For example, Xamarin developers can use SignalR for building Android apps using Xamarin.Android 8.4.0.1 and later and iOS apps using Xamarin.iOS 11.14.0.4 and later.

If the server runs IIS, the WebSockets transport requires IIS 8.0 or later on Windows Server 2012 or later. Other transports are supported on all platforms.

Java client

The Java client supports Java 8 and later versions.

Unsupported clients

The following clients are available but are experimental or unofficial. They aren't currently supported and may never be.

- C++ client
- Swift client

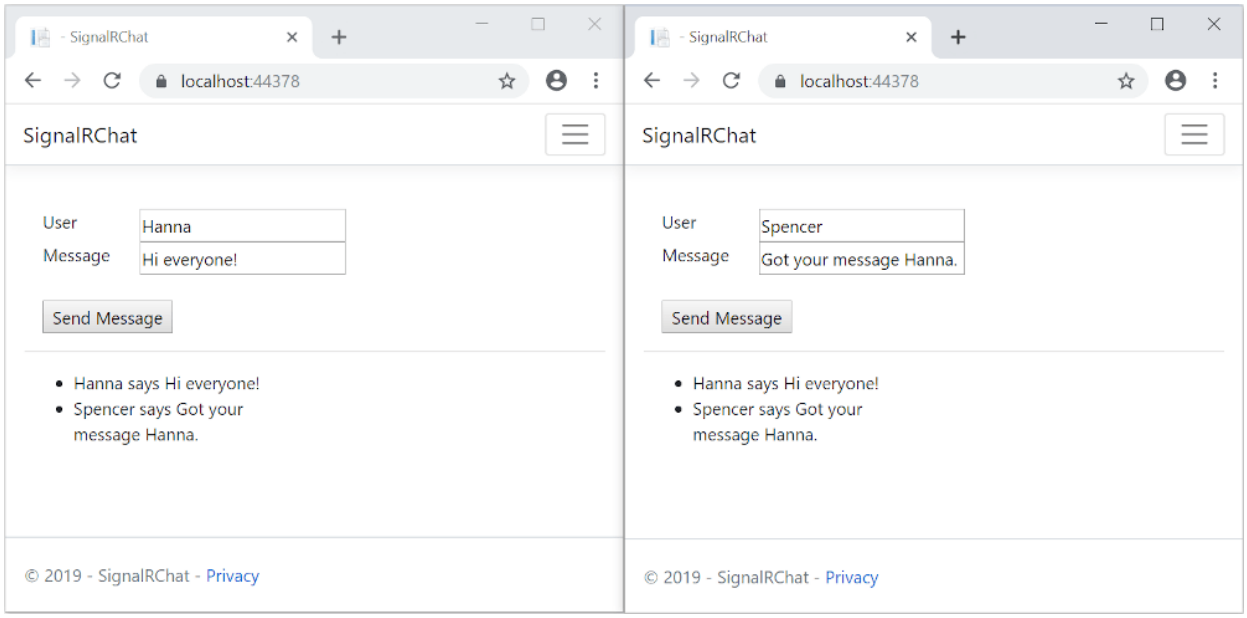
Tutorial: Get started with ASP.NET Core SignalR

9/22/2020 • 13 minutes to read • Edit Online

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- Create a web project.
- Add the SignalR client library.
- Create a SignalR hub.
- Configure the project to use SignalR.
- Add code that sends messages from any client to all connected clients.

At the end, you'll have a working chat app:



Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

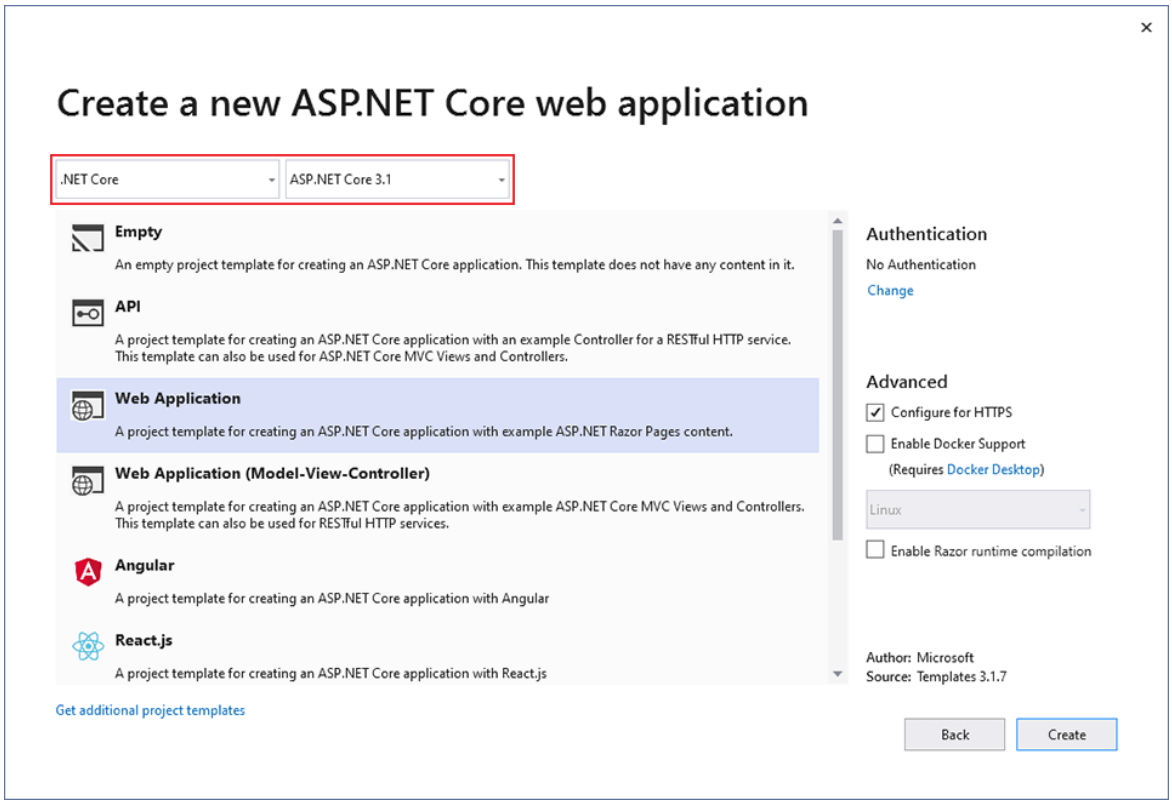
Create a web app project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the menu, select **File > New Project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web Application**, and then select **Next**.
- In the **Configure your new project** dialog, name the project *SignalRChat*, and then select **Create**.
- In the **Create a new ASP.NET Core web Application** dialog, select **.NET Core** and **ASP.NET Core**

3.1.

- Select **Web Application** to create a project that uses Razor Pages, and then select **Create**.

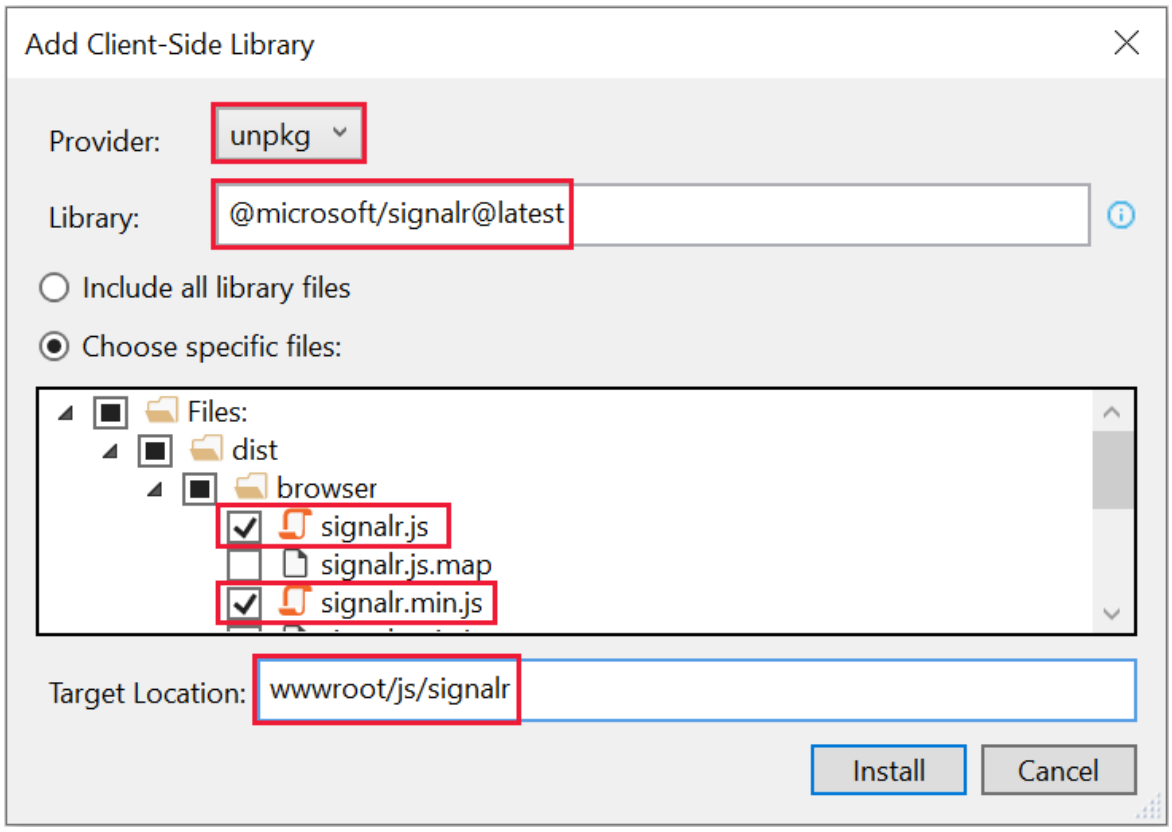


Add the SignalR client library

The SignalR server library is included in the ASP.NET Core 3.1 shared framework. The JavaScript client library isn't automatically included in the project. For this tutorial, you use Library Manager (LibMan) to get the client library from *unpkg*. unpkg is a content delivery network (CDN) that can deliver anything found in npm, the Node.js package manager.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.
- In the **Add Client-Side Library** dialog, for **Provider** select **unpkg**.
- For **Library**, enter `@microsoft/signalr@latest`.
- Select **Choose specific files**, expand the *dist/browser* folder, and select *signalr.js* and *signalr.min.js*.
- Set **Target Location** to *wwwroot/js/signalr/*, and select **Install**.



LibMan creates a *wwwroot/js/signalr* folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

- In the SignalRChat project folder, create a *Hubs* folder.
- In the *Hubs* folder, create a *ChatHub.cs* file with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

The `ChatHub` class inherits from the SignalR `Hub` class. The `Hub` class manages connections, groups, and messaging.

The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR.

- Add the following highlighted code to the *Startup.cs* file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
            services.AddSignalR();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                // The default HSTS value is 30 days. You may want to change this for production
scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
                endpoints.MapHub<ChatHub>("/chathub");
            });
        }
    }
}
```

These changes add SignalR to the ASP.NET Core dependency injection and routing systems.

Add SignalR client code

- Replace the content in *Pages\Index.cshtml* with the following code:

```
@page
    <div class="container">
        <div class="row"> </div>
        <div class="row">
            <div class="col-2">User</div>
            <div class="col-4"><input type="text" id="userInput" /></div>
        </div>
        <div class="row">
            <div class="col-2">Message</div>
            <div class="col-4"><input type="text" id="messageInput" /></div>
        </div>
        <div class="row"> </div>
        <div class="row">
            <div class="col-6">
                <input type="button" id="sendButton" value="Send Message" />
            </div>
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <hr />
        </div>
    </div>
    <div class="row">
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

  The preceding code:

  - Creates text boxes for name and message text, and a submit button.
  - Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
  - Includes script references to SignalR and the *chat.js* application code that you create in the next step.
- In the *wwwroot/js* folder, create a *chat.js* file with the following code:

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable send button until connection is established
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    var encodedMsg = user + " says " + msg;
    var li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding code:

- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

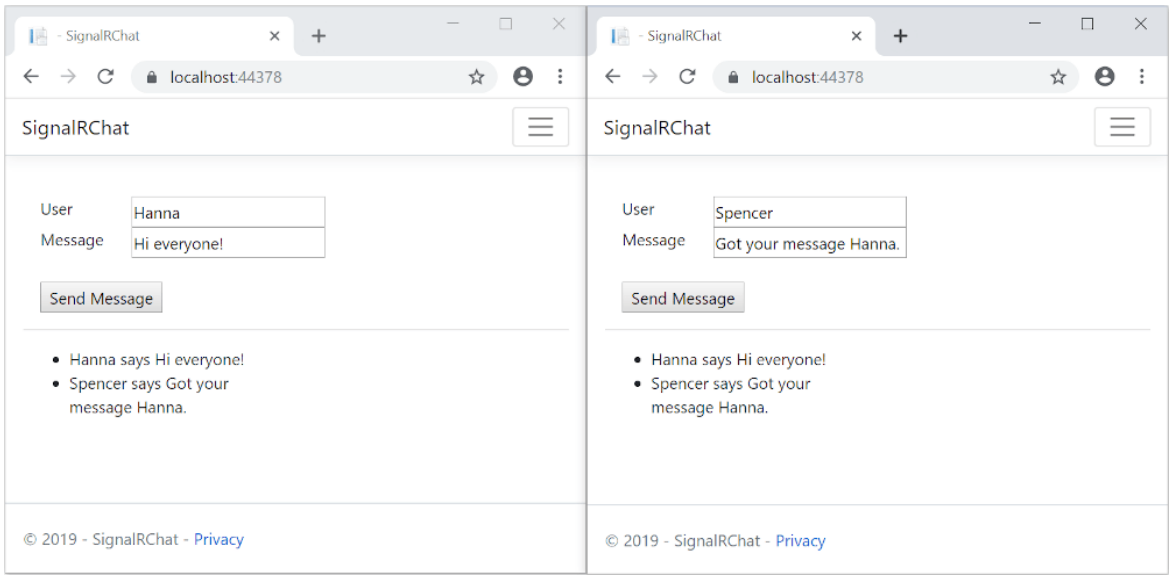
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Press **CTRL+F5** to run the app without debugging.

- Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

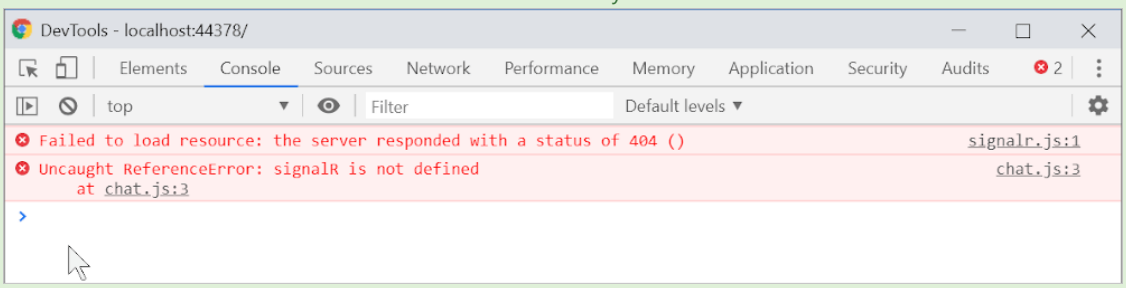
- Choose either browser, enter a name and message, and select the **Send Message** button.

  The name and message are displayed on both pages instantly.



TIP

- If the app doesn't work, open your browser developer tools (F12) and go to the console. You might see errors related to your HTML and JavaScript code. For example, suppose you put *signalr.js* in a different folder than directed. In that case the reference to that file won't work and you'll see a 404 error in the console.

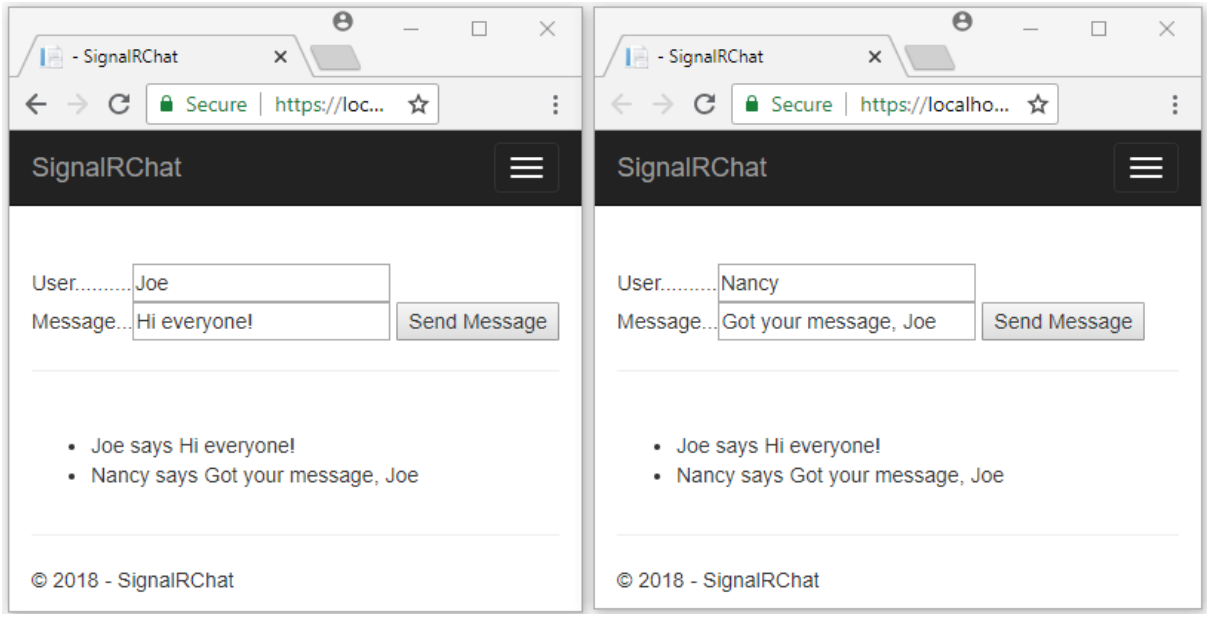


- If you get the error ERR_SPDY_INADEQUATE_TRANSPORT_SECURITY in Chrome, run these commands to update your development certificate:

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- Create a web project.
- Add the SignalR client library.
- Create a SignalR hub.
- Configure the project to use SignalR.
- Add code that sends messages from any client to all connected clients. At the end, you'll have a working chat app:



Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

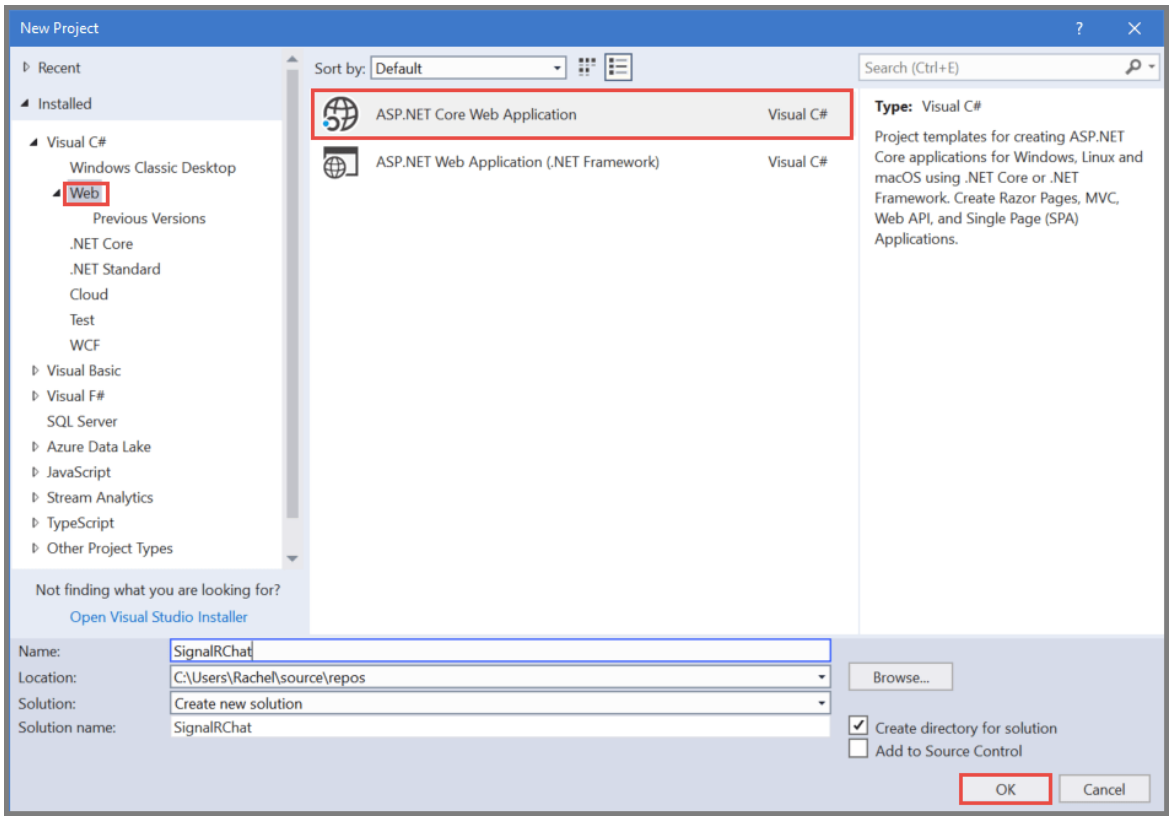
- Visual Studio 2017 version 15.9 or later with the **ASP.NET and web development** workload. You can use Visual Studio 2019, but some project creation steps differ from what's shown in the tutorial.
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

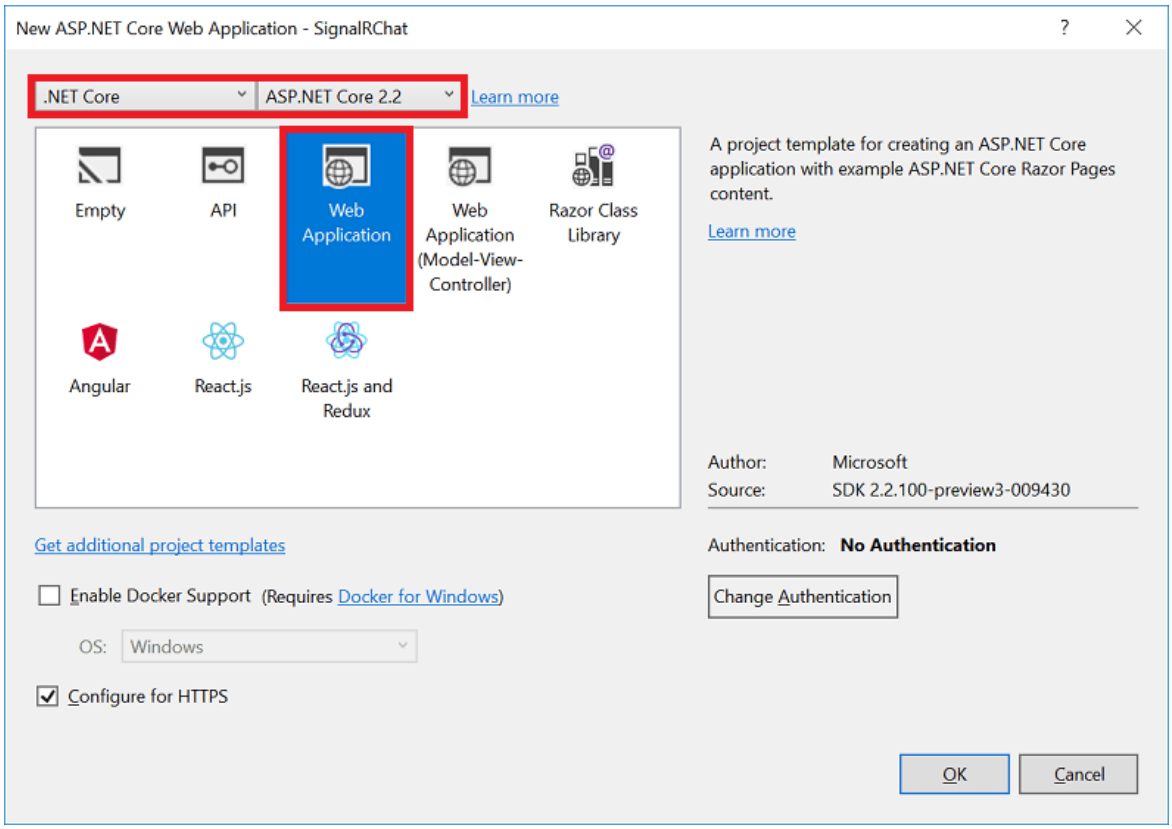
Create a web project

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- From the menu, select **File > New Project**.
- In the **New Project** dialog, select **Installed > Visual C# > Web > ASP.NET Core Web Application**. Name the project *SignalRChat*.



- Select **Web Application** to create a project that uses Razor Pages.
- Select a target framework of **.NET Core**, select **ASP.NET Core 2.2**, and click **OK**.

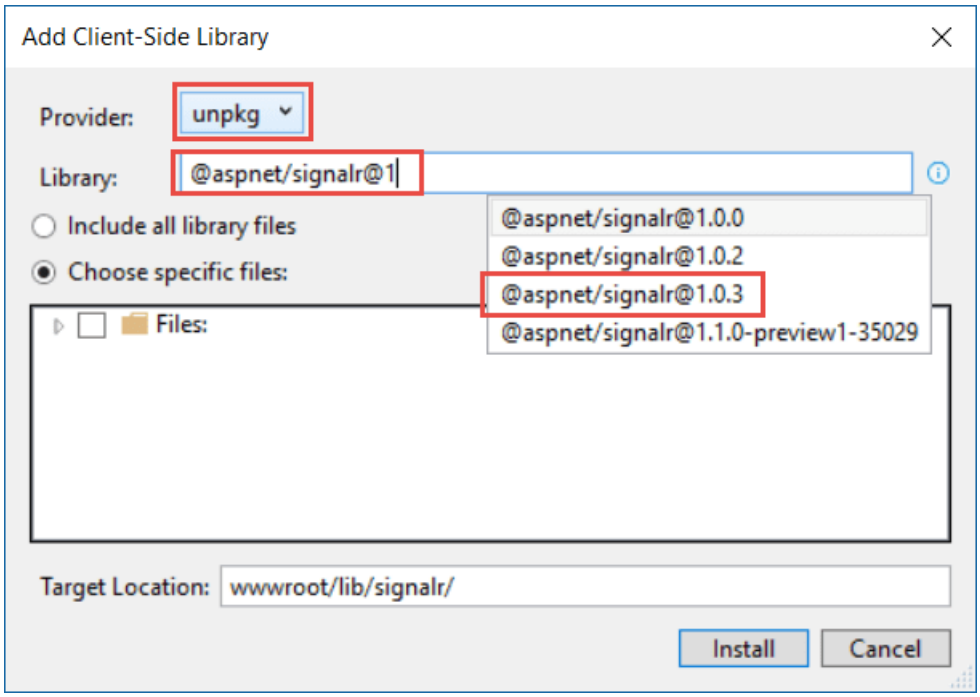


Add the SignalR client library

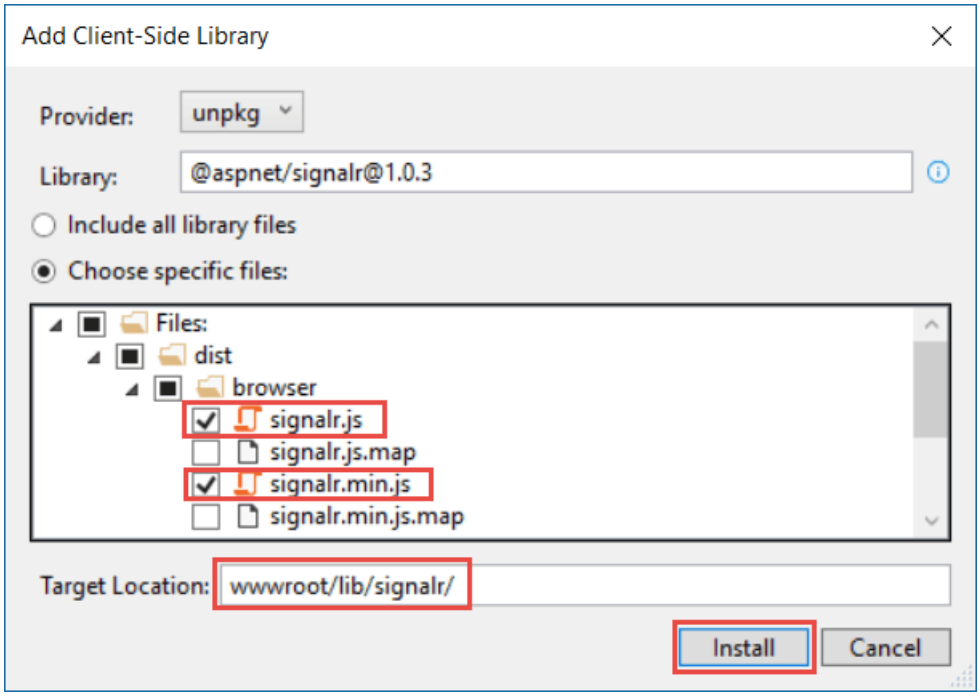
The SignalR server library is included in the `Microsoft.AspNetCore.App` metapackage. The JavaScript client library isn't automatically included in the project. For this tutorial, you use Library Manager (LibMan) to get the client library from *unpkg*. unpkg is a content delivery network (CDN) that can deliver anything found in npm, the Node.js package manager.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.
- In the **Add Client-Side Library** dialog, for **Provider** select **unpkg**.
- For **Library**, enter `@microsoft/signalr@3`, and select the latest version that isn't preview.



- Select **Choose specific files**, expand the *dist/browser* folder, and select *signalr.js* and *signalr.min.js*.
- Set **Target Location** to *wwwroot/lib/signalr/*, and select **Install**.



LibMan creates a *wwwroot/lib/signalr* folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

- In the SignalRChat project folder, create a *Hubs* folder.
- In the *Hubs* folder, create a *ChatHub.cs* file with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

  The `ChatHub` class inherits from the SignalR `Hub` class. The `Hub` class manages connections, groups, and messaging.

  The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR.

- Add the following highlighted code to the *Startup.cs* file.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                // This lambda determines whether user consent for non-essential cookies is needed
for a given request.
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });


            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

            services.AddSignalR();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            app.UseSignalR(routes =>
            {
                routes.MapHub<ChatHub>("/chathub");
            });
            app.UseMvc();
        }
    }
}
```

These changes add SignalR to the ASP.NET Core dependency injection system and the middleware pipeline.

Add SignalR client code

- Replace the content in *Pages\Index.cshtml* with the following code:

```
@page
<div class="container">
    <div class="row"> </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            User.....<input type="text" id="userInput" />
            <br />
            Message...<input type="text" id="messageInput" />
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row">
        <div class="col-12">
            <hr />
        </div>
    </div>
    <div class="row">
        <div class="col-6"> </div>
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/lib/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

  The preceding code:

  - Creates text boxes for name and message text, and a submit button.
  - Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
  - Includes script references to SignalR and the *chat.js* application code that you create in the next step.

- In the *wwwroot/js* folder, create a *chat.js* file with the following code:

```
"use strict";

var connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable send button until connection is established
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    var encodedMsg = user + " says " + msg;
    var li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});

connection.start().then(function(){
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function (event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding code:

- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

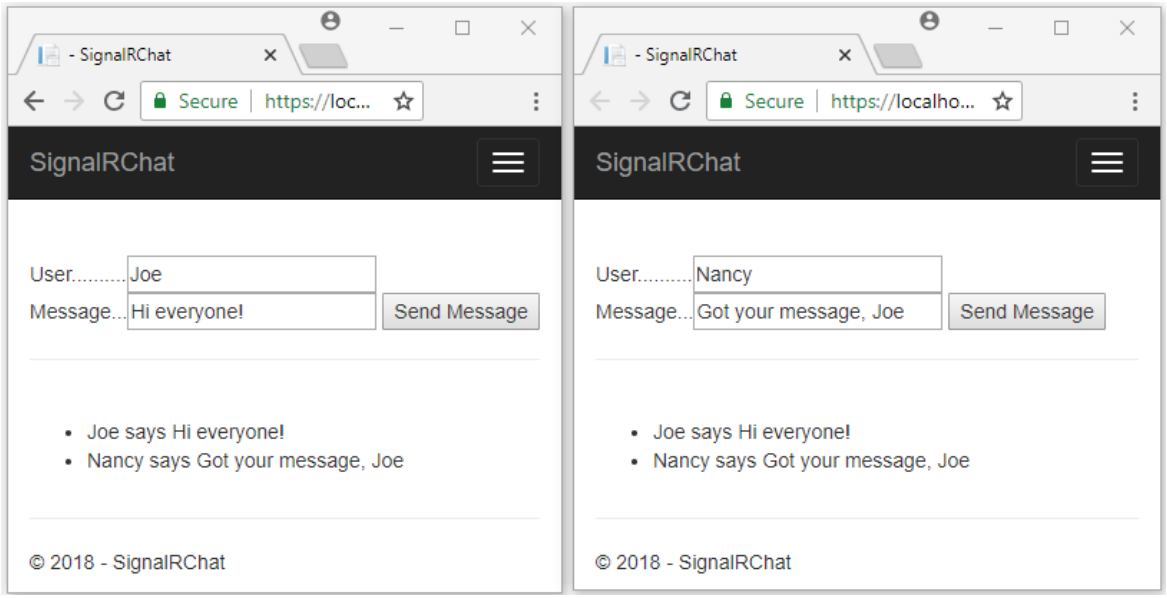
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Press **CTRL+F5** to run the app without debugging.

- Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

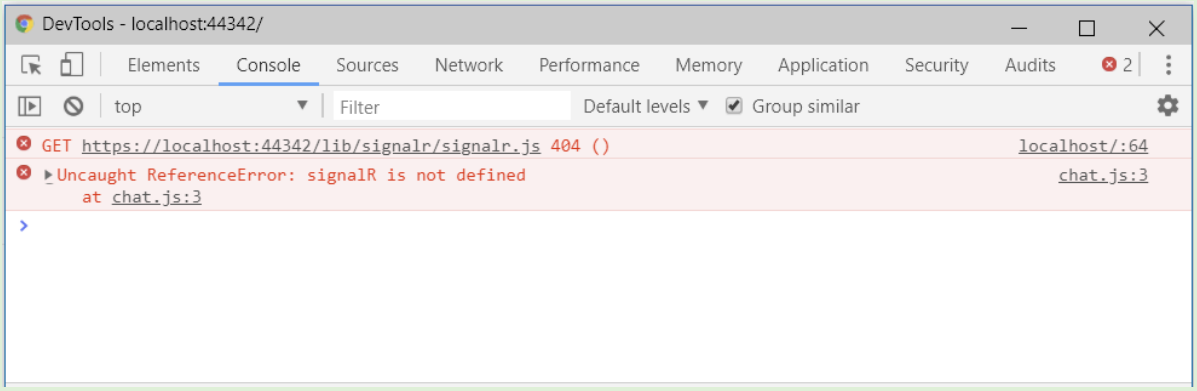
- Choose either browser, enter a name and message, and select the **Send Message** button.

  The name and message are displayed on both pages instantly.



TIP

If the app doesn't work, open your browser developer tools (F12) and go to the console. You might see errors related to your HTML and JavaScript code. For example, suppose you put *signalr.js* in a different folder than directed. In that case the reference to that file won't work and you'll see a 404 error in the console.



Additional resources

- Youtube version of this tutorial

Use ASP.NET Core SignalR with TypeScript and Webpack

9/22/2020 • 21 minutes to read • Edit Online

By Sébastien Sougnez and Scott Addie

Webpack enables developers to bundle and build the client-side resources of a web app. This tutorial demonstrates using Webpack in an ASP.NET Core SignalR web app whose client is written in TypeScript.

In this tutorial, you learn how to:

- Scaffold a starter ASP.NET Core SignalR app
- Configure the SignalR TypeScript client
- Configure a build pipeline using Webpack
- Configure the SignalR server
- Enable communication between client and server

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code

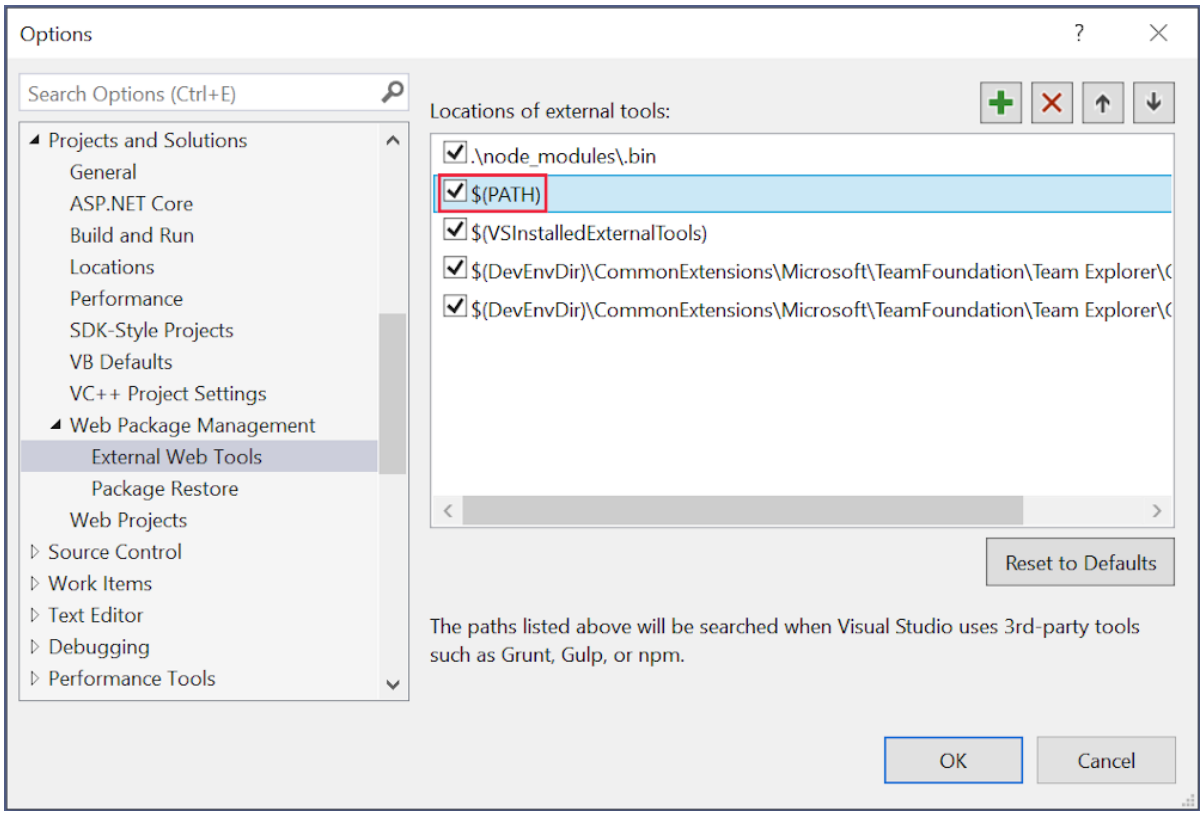
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 3.0 or later
- Node.js with npm

Create the ASP.NET Core web app

- Visual Studio
- Visual Studio Code

Configure Visual Studio to look for npm in the *PATH* environment variable. By default, Visual Studio uses the version of npm found in its installation directory. Follow these instructions in Visual Studio:

1. Launch Visual Studio. At the start window, select **Continue without code**.
2. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.
3. Select the *$(PATH)* entry from the list. Click the up arrow to move the entry to the second position in the list, and select **OK**.



Visual Studio configuration is complete.

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Web Application** template. Select **Next**.
2. Name the project *SignalRWebPack*, and select **Create**.
3. Select *.NET Core* from the target framework drop-down, and select *ASP.NET Core 3.1* from the framework selector drop-down. Select the **Empty** template, and select **Create**.

Add the `Microsoft.TypeScript.MSBuild` package to the project:

1. In **Solution Explorer** (right pane), right-click the project node and select **Manage NuGet Packages**. In the **Browse** tab, search for `Microsoft.TypeScript.MSBuild`, and then click **Install** on the right to install the package.

Visual Studio adds the NuGet package under the **Dependencies** node in **Solution Explorer**, enabling TypeScript compilation in the project.

Configure Webpack and TypeScript

The following steps configure the conversion of TypeScript to JavaScript and the bundling of client-side resources.

1. Run the following command in the project root to create a *package.json* file:

```
npm init -y
```

2. Add the highlighted property to the *package.json* file and save the file changes:

```
{
  "name": "SignalRWebPack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Setting the `private` property to `true` prevents package installation warnings in the next step.

3. Install the required npm packages. Run the following command from the project root:

```
npm i -D -E clean-webpack-plugin@3.0.0 css-loader@3.4.2 html-webpack-plugin@3.2.0 mini-css-extract-plugin@0.9.0 ts-loader@6.2.1 typescript@3.7.5 webpack@4.41.5 webpack-cli@3.3.10
```

Some command details to note:

- A version number follows the `@` sign for each package name. npm installs those specific package versions.
- The `-E` option disables npm's default behavior of writing semantic versioning range operators to *package.json*. For example, `"webpack": "4.41.5"` is used instead of `"webpack": "^4.41.5"`. This option prevents unintended upgrades to newer package versions.

See the npm-install docs for more detail.

4. Replace the `scripts` property of the *package.json* file with the following code:

```
"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

Some explanation of the scripts:

- `build`: Bundles the client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The `mode` option disables production optimizations, such as tree shaking and minification. Only use `build` in development.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET Core CLI's publish command to publish the app.

5. Create a file named *webpack.config.js*, in the project root, with the following code:

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/"
    },
    resolve: {
        extensions: [".js", ".ts"]
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader"
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"]
            }
        ]
    },
    plugins: [
        new CleanWebpackPlugin(),
        new HtmlWebpackPlugin({
            template: "./src/index.html"
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css"
        })
    ]
};
```

The preceding file configures the Webpack compilation. Some configuration details to note:

- The `output` property overrides the default value of *dist*. The bundle is instead emitted in the *wwwroot* directory.
- The `resolve.extensions` array includes *.js* to import the SignalR client JavaScript.

6. Create a new *src* directory in the project root to store the project's client-side assets.

7. Create *src/index.html* with the following markup.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR</title>
</head>
<body>
    <div id="divMessages" class="messages">
    </div>
    <div class="input-zone">
        <label id="lblMessage" for="tbMessage">Message:</label>
        <input id="tbMessage" class="input-zone-input" type="text" />
        <button id="btnSend">Send</button>
    </div>
</body>
</html>
```

The preceding HTML defines the homepage's boilerplate markup.

8. Create a new *src/css* directory. Its purpose is to store the project's *.css* files.

9. Create *src/css/main.css* with the following CSS:

```
*, *::before, *::after {
    box-sizing: border-box;
}

html, body {
    margin: 0;
    padding: 0;
}

.input-zone {
    align-items: center;
    display: flex;
    flex-direction: row;
    margin: 10px;
}

.input-zone-input {
    flex: 1;
    margin-right: 10px;
}

.message-author {
    font-weight: bold;
}

.messages {
    border: 1px solid #000;
    margin: 10px;
    max-height: 300px;
    min-height: 300px;
    overflow-y: auto;
    padding: 5px;
}
```

The preceding *main.css* file styles the app.

10. Create *src/tsconfig.json* with the following JSON:

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

The preceding code configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript.

11. Create *src/index.ts* with the following code:

```
import "./css/main.css";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.key === "Enter") {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding TypeScript retrieves references to DOM elements and attaches two event handlers:

- `keyup`: This event fires when the user types in the `tbMessage` textbox. The `send` function is called when the user presses the **Enter** key.
- `click`: This event fires when the user clicks the **Send** button. The `send` function is called.

Configure the app

1. In `Startup.Configure`, add calls to UseDefaultFiles and UseStaticFiles.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();
    app.UseDefaultFiles();
    app.UseStaticFiles();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/hub");
    });

}
```

The preceding code allows the server to locate and serve the *index.html* file. The file is served whether the user enters its full URL or the root URL of the web app.

2. At the end of `Startup.Configure`, map a */hub* route to the `ChatHub` hub. Replace the code that displays *Hello World!* with the following line:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>("/hub");
});
```

3. In `Startup.ConfigureServices`, call AddSignalR.

```
services.AddSignalR();
```

4. Create a new directory named *Hubs* in the project root *SignalRWebPack/* to store the SignalR hub.

5. Create hub *Hubs/ChatHub.cs* with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
    }
}
```

6. Add the following `using` statement at the top of the *Startup.cs* file to resolve the `ChatHub` reference:

```
using SignalRWebPack.Hubs;
```

Enable client and server communication

The app currently displays a basic form to send messages, but is not yet functional. The server is listening to a specific route but does nothing with sent messages.

1. Run the following command at the project root:

```
npm i @microsoft/signalr @types/node
```

   The preceding command installs:

   - The SignalR TypeScript client, which allows the client to send messages to the server.
   - The TypeScript type definitions for Node.js, which enables compile-time checking of Node.js types.

2. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@microsoft/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let m = document.createElement("div");

    m.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(m);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.key === "Enter") {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding code supports receiving messages from the server. The `HubConnectionBuilder` class creates a new builder for configuring the server connection. The `withUrl` function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. Any number of message names can be listened to. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new `div` element is created with the author's name and the message content in its `innerHTML` attribute. It's added to the main `div` element displaying the messages.

3. Now that the client can receive a message, configure it to send messages. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@microsoft/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let messages = document.createElement("div");

    messages.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(messages);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.key === "Enter") {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
    connection.send("newMessage", username, tbMessage.value)
        .then(() => tbMessage.value = "");
}
```

Sending a message through the WebSockets connection requires calling the `send` method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as `newMessage` is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

4. Add the `NewMessage` method to the `ChatHub` class:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
        public async Task NewMessage(long username, string message)
        {
            await Clients.All.SendAsync("messageReceived", username, message);
        }
    }
}
```

The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic `on` method to receive all the messages. A method named after the message name suffices.

In this example, the TypeScript client sends a message identified as `newMessage`. The C# `NewMessage` method

expects the data sent by the client. A call is made to SendAsync on Clients.All. The received messages are sent to all clients connected to the hub.

Test the app

Confirm that the app works with the following steps.

- Visual Studio
- Visual Studio Code

1. Run Webpack in *release* mode. Using the **Package Manager Console** window, run the following command in the project root. If you are not in the project root, enter `cd SignalRWebPack` before entering the command.

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the *wwwroot* folder.

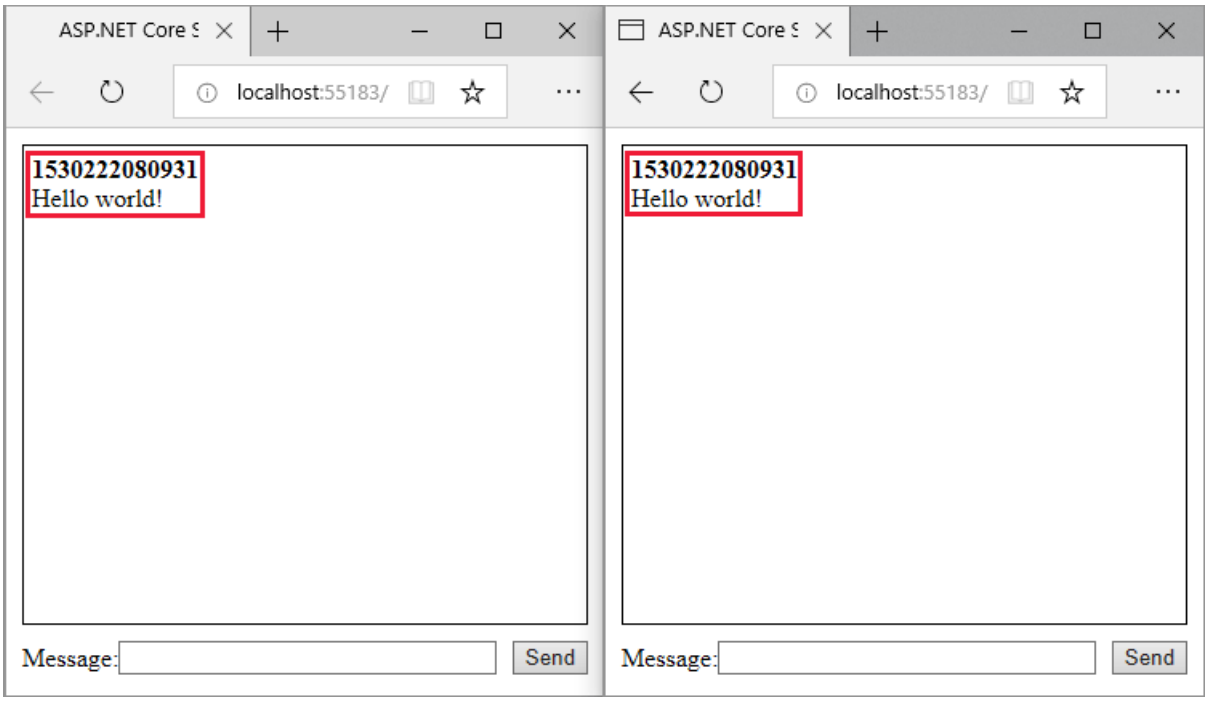
   Webpack completed the following tasks:

   - Purged the contents of the *wwwroot* directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from *src* to the *wwwroot* directory.
   - Injected the following elements into the *wwwroot/index.html* file:
     - A `<link>` tag, referencing the *wwwroot/main.<hash>.css* file. This tag is placed immediately before the closing `</head>` tag.
     - A `<script>` tag, referencing the minified *wwwroot/main.<hash>.js* file. This tag is placed immediately before the closing `</body>` tag.
2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The *wwwroot/index.html* file is served at `http://localhost:<port_number>`.

   If you get compile errors, try closing and reopening the solution.

3. Open another browser instance (any browser). Paste the URL in the address bar.

4. Choose either browser, type something in the **Message** text box, and click the **Send** button. The unique user name and message are displayed on both pages instantly.



Prerequisites

- Visual Studio
- Visual Studio Code

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later
- Node.js with npm

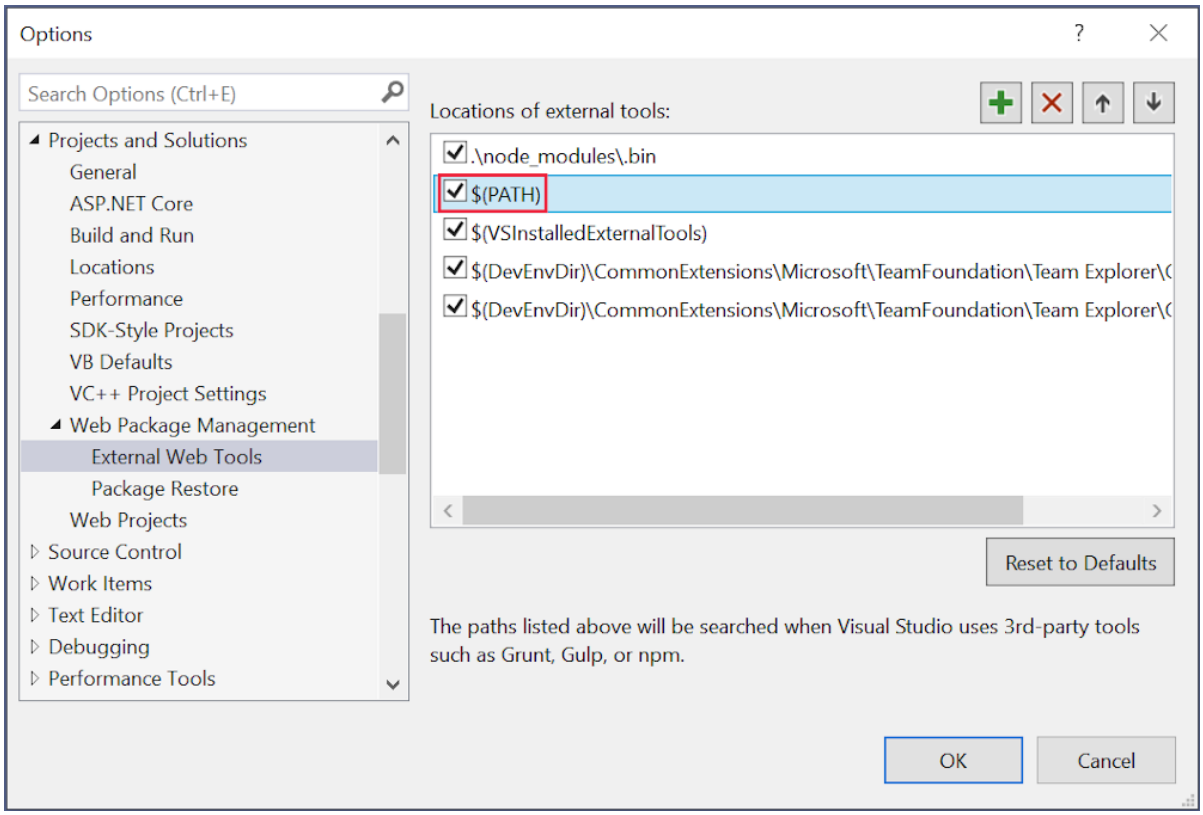
Create the ASP.NET Core web app

- Visual Studio
- Visual Studio Code

Configure Visual Studio to look for npm in the *PATH* environment variable. By default, Visual Studio uses the version of npm found in its installation directory. Follow these instructions in Visual Studio:

1. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.

2. Select the *$(PATH)* entry from the list. Click the up arrow to move the entry to the second position in the list.



Visual Studio configuration is completed. It's time to create the project.

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Web Application** template.
2. Name the project *SignalRWebPack*, and select **Create**.
3. Select *.NET Core* from the target framework drop-down, and select *ASP.NET Core 2.2* from the framework selector drop-down. Select the **Empty** template, and select **Create**.

Configure Webpack and TypeScript

The following steps configure the conversion of TypeScript to JavaScript and the bundling of client-side resources.

1. Run the following command in the project root to create a *package.json* file:

```
npm init -y
```

2. Add the highlighted property to the *package.json* file:

```
{
  "name": "SignalRWebPack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

   Setting the `private` property to `true` prevents package installation warnings in the next step.

3. Install the required npm packages. Run the following command from the project root:

```
npm install -D -E clean-webpack-plugin@1.0.1 css-loader@2.1.0 html-webpack-plugin@4.0.0-beta.5 mini-
css-extract-plugin@0.5.0 ts-loader@5.3.3 typescript@3.3.3 webpack@4.29.3 webpack-cli@3.2.3
```

Some command details to note:

- A version number follows the `@` sign for each package name. npm installs those specific package versions.
- The `-E` option disables npm's default behavior of writing semantic versioning range operators to *package.json*. For example, `"webpack": "4.29.3"` is used instead of `"webpack": "^4.29.3"`. This option prevents unintended upgrades to newer package versions.

See the npm-install docs for more detail.

4. Replace the `scripts` property of the *package.json* file with the following code:

```
"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

Some explanation of the scripts:

- `build`: Bundles the client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The `mode` option disables production optimizations, such as tree shaking and minification. Only use `build` in development.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET Core CLI's publish command to publish the app.

5. Create a file named *webpack.config.js* in the project root, with the following code:

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const CleanWebpackPlugin = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/"
    },
    resolve: {
        extensions: [".js", ".ts"]
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader"
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"]
            }
        ]
    },
    plugins: [
        new CleanWebpackPlugin(["wwwroot/*"]),
        new HtmlWebpackPlugin({
            template: "./src/index.html"
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css"
        })
    ]
};
```

The preceding file configures the Webpack compilation. Some configuration details to note:

- The `output` property overrides the default value of *dist*. The bundle is instead emitted in the *wwwroot* directory.
- The `resolve.extensions` array includes *.js* to import the SignalR client JavaScript.

6. Create a new *src* directory in the project root to store the project's client-side assets.

7. Create *src/index.html* with the following markup.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR</title>
</head>
<body>
    <div id="divMessages" class="messages">
    </div>
    <div class="input-zone">
        <label id="lblMessage" for="tbMessage">Message:</label>
        <input id="tbMessage" class="input-zone-input" type="text" />
        <button id="btnSend">Send</button>
    </div>
</body>
</html>
```

The preceding HTML defines the homepage's boilerplate markup.

8. Create a new *src/css* directory. Its purpose is to store the project's *.css* files.

9. Create *src/css/main.css* with the following markup:

```
*, *::before, *::after {
    box-sizing: border-box;
}

html, body {
    margin: 0;
    padding: 0;
}

.input-zone {
    align-items: center;
    display: flex;
    flex-direction: row;
    margin: 10px;
}

.input-zone-input {
    flex: 1;
    margin-right: 10px;
}

.message-author {
    font-weight: bold;
}

.messages {
    border: 1px solid #000;
    margin: 10px;
    max-height: 300px;
    min-height: 300px;
    overflow-y: auto;
    padding: 5px;
}
```

The preceding *main.css* file styles the app.

10. Create *src/tsconfig.json* with the following JSON:

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

The preceding code configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript.

11. Create *src/index.ts* with the following code:

```
import "./css/main.css";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding TypeScript retrieves references to DOM elements and attaches two event handlers:

- `keyup`: This event fires when the user types in the `tbMessage` textbox. The `send` function is called when the user presses the **Enter** key.
- `click`: This event fires when the user clicks the **Send** button. The `send` function is called.

Configure the ASP.NET Core app

1. The code provided in the `Startup.Configure` method displays *Hello World!*. Replace the `app.Run` method call with calls to UseDefaultFiles and UseStaticFiles.

   ```
   app.UseDefaultFiles();
   app.UseStaticFiles();
   ```

   The preceding code allows the server to locate and serve the *index.html* file, whether the user enters its full URL or the root URL of the web app.

2. Call AddSignalR in `Startup.ConfigureServices`. It adds the SignalR services to the project.

   ```
   services.AddSignalR();
   ```

3. Map a */hub* route to the `ChatHub` hub. Add the following lines at the end of `Startup.Configure`:

   ```
   app.UseSignalR(options =>
   {
       options.MapHub<ChatHub>("/hub");
   });
   ```

4. Create a new directory, called *Hubs*, in the project root. Its purpose is to store the SignalR hub, which is created in the next step.

5. Create hub *Hubs/ChatHub.cs* with the following code:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
    }
}
```

6. Add the following code at the top of the *Startup.cs* file to resolve the `ChatHub` reference:

```
using SignalRWebPack.Hubs;
```

Enable client and server communication

The app currently displays a simple form to send messages. Nothing happens when you try to do so. The server is listening to a specific route but does nothing with sent messages.

1. Run the following command at the project root:

```
npm install @aspnet/signalr
```

   The preceding command installs the SignalR TypeScript client, which allows the client to send messages to the server.

2. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let m = document.createElement("div");

    m.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(m);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
}
```

The preceding code supports receiving messages from the server. The `HubConnectionBuilder` class creates a new builder for configuring the server connection. The `withUrl` function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. You can listen to any number of message names. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new `div` element is created with the author's name and the message content in its `innerHTML` attribute. The new message is added to the main `div` element displaying the messages.

3. Now that the client can receive a message, configure it to send messages. Add the highlighted code to the *src/index.ts* file:

```
import "./css/main.css";
import * as signalR from "@aspnet/signalr";

const divMessages: HTMLDivElement = document.querySelector("#divMessages");
const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message: string) => {
    let messageContainer = document.createElement("div");

    messageContainer.innerHTML =
        `<div class="message-author">${username}</div><div>${message}</div>`;

    divMessages.appendChild(messageContainer);
    divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch(err => document.write(err));

tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
    if (e.keyCode === 13) {
        send();
    }
});

btnSend.addEventListener("click", send);

function send() {
    connection.send("newMessage", username, tbMessage.value)
              .then(() => tbMessage.value = "");
}
```

Sending a message through the WebSockets connection requires calling the `send` method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as `newMessage` is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

4. Add the `NewMessage` method to the `ChatHub` class:

```
using Microsoft.AspNetCore.SignalR;
using System.Threading.Tasks;

namespace SignalRWebPack.Hubs
{
    public class ChatHub : Hub
    {
        public async Task NewMessage(long username, string message)
        {
            await Clients.All.SendAsync("messageReceived", username, message);
        }
    }
}
```

The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic `on` method to receive all the messages. A method named after the message name suffices.

In this example, the TypeScript client sends a message identified as `newMessage`. The C# `NewMessage` method

expects the data sent by the client. A call is made to SendAsync on Clients.All. The received messages are sent to all clients connected to the hub.

Test the app

Confirm that the app works with the following steps.

- Visual Studio
- Visual Studio Code

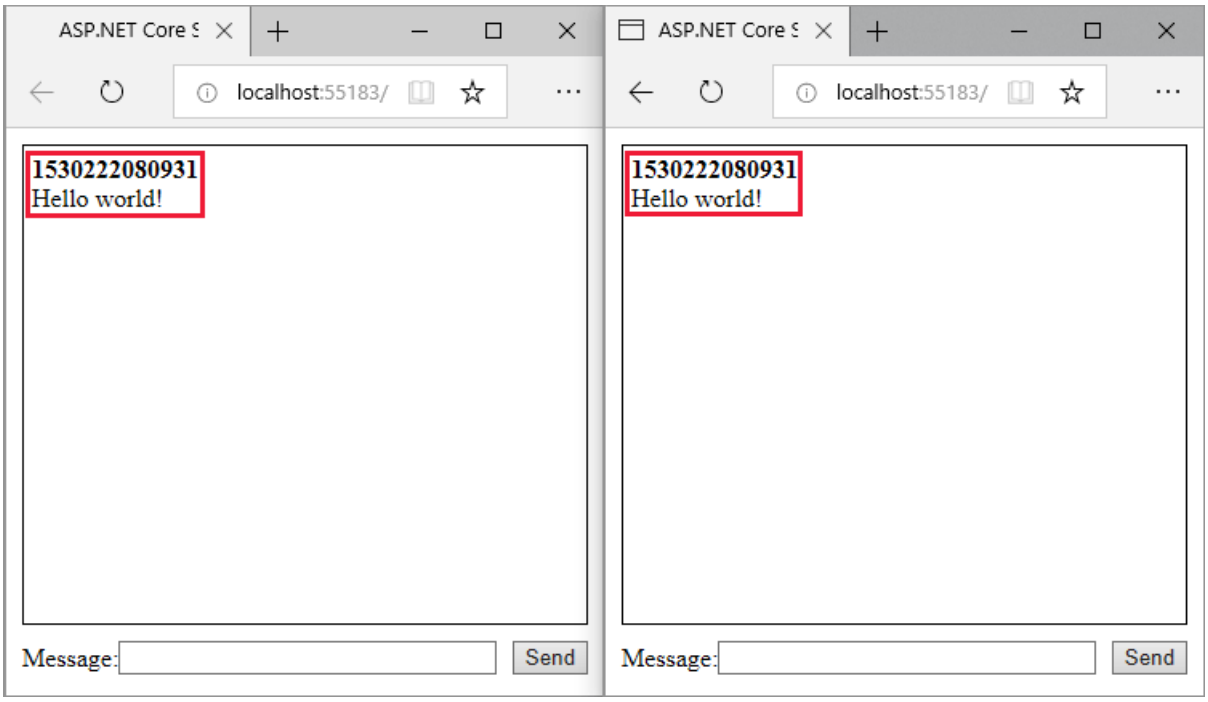
1. Run Webpack in *release* mode. Using the **Package Manager Console** window, run the following command in the project root. If you are not in the project root, enter `cd SignalRWebPack` before entering the command.

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the *wwwroot* folder.

   Webpack completed the following tasks:

   - Purged the contents of the *wwwroot* directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from *src* to the *wwwroot* directory.
   - Injected the following elements into the *wwwroot/index.html* file:
     - A `<link>` tag, referencing the *wwwroot/main.<hash>.css* file. This tag is placed immediately before the closing `</head>` tag.
     - A `<script>` tag, referencing the minified *wwwroot/main.<hash>.js* file. This tag is placed immediately before the closing `</body>` tag.
2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The *wwwroot/index.html* file is served at `http://localhost:<port_number>`.
3. Open another browser instance (any browser). Paste the URL in the address bar.
4. Choose either browser, type something in the **Message** text box, and click the **Send** button. The unique user name and message are displayed on both pages instantly.



Additional resources

- ASP.NET Core SignalR JavaScript client
- Use hubs in ASP.NET Core SignalR

Use ASP.NET Core SignalR with Blazor WebAssembly

9/22/2020 • 7 minutes to read • Edit Online

By Daniel Roth and Luke Latham

This tutorial teaches the basics of building a real-time app using SignalR with Blazor WebAssembly. You learn how to:

- Create a Blazor WebAssembly Hosted app project
- Add the SignalR client library
- Add a SignalR hub
- Add SignalR services and an endpoint for the SignalR hub
- Add Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

View or download sample code (how to download)

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

- Visual Studio 2019 16.6 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a hosted Blazor WebAssembly app project

Follow the guidance for your choice of tooling:

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

> **NOTE**
> Visual Studio 16.6 or later and .NET Core SDK 3.1.300 or later are required.

1. Create a new project.
2. Select **Blazor App** and select **Next**.
3. Type `BlazorSignalRApp` in the **Project name** field. Confirm the **Location** entry is correct or provide a location for the project. Select **Create**.
4. Choose the **Blazor WebAssembly App** template.
5. Under **Advanced**, select the **ASP.NET Core hosted** check box.

6. Select **Create**.

Add the SignalR client library

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

1. In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.
2. In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.
3. With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.
4. In the search results, select the `Microsoft.AspNetCore.SignalR.Client` package and select **Install**.
5. If the **Preview Changes** dialog appears, select **OK**.
6. If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the `BlazorSignalRApp.Server` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Server.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

Add services and an endpoint for the SignalR hub

1. In the `BlazorSignalRApp.Server` project, open the `Startup.cs` file.
2. Add the namespace for the `ChatHub` class to the top of the file:

   ```
   using BlazorSignalRApp.Server.Hubs;
   ```

3. Add SignalR and Response Compression Middleware services to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR();
    services.AddControllersWithViews();
    services.AddResponseCompression(opts =>
    {
        opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "application/octet-stream" });
    });
}
```

4. In `Startup.Configure`:

   - Use Response Compression Middleware at the top of the processing pipeline's configuration.
   - Between the endpoints for controllers and the client-side fallback, add an endpoint for the hub.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseResponseCompression();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseWebAssemblyDebugging();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseBlazorFrameworkFiles();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapHub<ChatHub>("/chathub");
        endpoints.MapFallbackToFile("index.html");
    });
}
```

Add Razor component code for chat

1. In the `BlazorSignalRApp.Client` project, open the `Pages/Index.razor` file.
2. Replace the markup with the following code:

```
@page "/"
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager NavigationManager
@implements IDisposable

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection hubConnection;
    private List<string> messages = new List<string>();
    private string userInput;
    private string messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(NavigationManager.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message) =>
        {
            var encodedMsg = $"{user}: {message}";
            messages.Add(encodedMsg);
            StateHasChanged();
        });

        await hubConnection.StartAsync();
    }

    Task Send() =>
        hubConnection.SendAsync("SendMessage", userInput, messageInput);

    public bool IsConnected =>
        hubConnection.State == HubConnectionState.Connected;

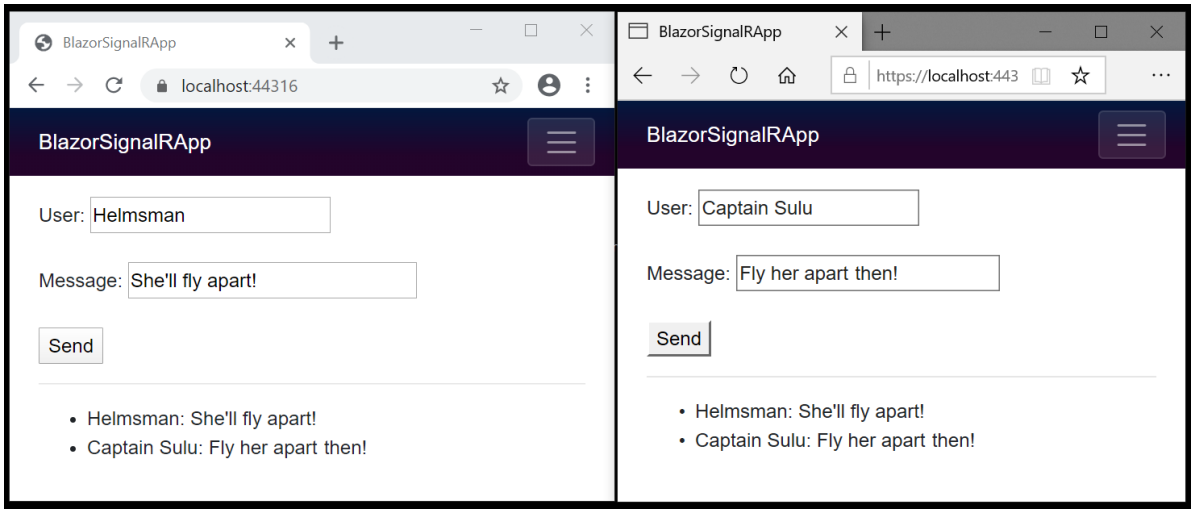
    public void Dispose()
    {
        _ = hubConnection.DisposeAsync();
    }
}
```

Run the app

1. Follow the guidance for your tooling:

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET Core CLI

1. In **Solution Explorer**, select the `BlazorSignalRApp.Server` project. Press `F5` to run the app with debugging or `Ctrl+F5` to run the app without debugging.
2. Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.
3. Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* ©1991 Paramount

Next steps

In this tutorial, you learned how to:

- Create a Blazor WebAssembly Hosted app project
- Add the SignalR client library
- Add a SignalR hub
- Add SignalR services and an endpoint for the SignalR hub
- Add Razor component code for chat

To learn more about building Blazor apps, see the Blazor documentation:

Introduction to ASP.NET Core Blazor

Additional resources

- Introduction to ASP.NET Core SignalR
- SignalR cross-origin negotiation for authentication

Use hubs in SignalR for ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

By Rachel Appel and Kevin Griffin

View or download sample code (how to download)

What is a SignalR hub

The SignalR Hubs API enables you to call methods on connected clients from the server. In the server code, you define methods that are called by client. In the client code, you define methods that are called from the server. SignalR takes care of everything behind the scenes that makes real-time client-to-server and server-to-client communications possible.

Configure SignalR hubs

The SignalR middleware requires some services, which are configured by calling `services.AddSignalR`.

```
services.AddSignalR();
```

When adding SignalR functionality to an ASP.NET Core app, setup SignalR routes by calling `endpoint.MapHub` in the `Startup.Configure` method's `app.UseEndpoints` callback.

```
app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>("/chathub");
});
```

When adding SignalR functionality to an ASP.NET Core app, setup SignalR routes by calling `app.UseSignalR` in the `Startup.Configure` method.

```
app.UseSignalR(route =>
{
    route.MapHub<ChatHub>("/chathub");
});
```

Create and use hubs

Create a hub by declaring a class that inherits from `Hub`, and add public methods to it. Clients can call methods that are defined as `public`.

```
public class ChatHub : Hub
{
    public Task SendMessage(string user, string message)
    {
        return Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

You can specify a return type and parameters, including complex types and arrays, as you would in any C# method. SignalR handles the serialization and deserialization of complex objects and arrays in your parameters and return values.

> **NOTE**
>
> Hubs are transient:
>
> - Don't store state in a property on the hub class. Every hub method call is executed on a new hub instance.
> - Use `await` when calling asynchronous methods that depend on the hub staying alive. For example, a method such as `Clients.All.SendAsync(...)` can fail if it's called without `await` and the hub method completes before `SendAsync` finishes.

The Context object

The `Hub` class has a `Context` property that contains the following properties with information about the connection:

PROPERTY	DESCRIPTION
`ConnectionId`	Gets the unique ID for the connection, assigned by SignalR. There is one connection ID for each connection.
`UserIdentifier`	Gets the user identifier. By default, SignalR uses the `ClaimTypes.NameIdentifier` from the `ClaimsPrincipal` associated with the connection as the user identifier.
`User`	Gets the `ClaimsPrincipal` associated with the current user.
`Items`	Gets a key/value collection that can be used to share data within the scope of this connection. Data can be stored in this collection and it will persist for the connection across different hub method invocations.
`Features`	Gets the collection of features available on the connection. For now, this collection isn't needed in most scenarios, so it isn't documented in detail yet.
`ConnectionAborted`	Gets a `CancellationToken` that notifies when the connection is aborted.

`Hub.Context` also contains the following methods:

METHOD	DESCRIPTION
`GetHttpContext`	Returns the `HttpContext` for the connection, or `null` if the connection is not associated with an HTTP request. For HTTP connections, you can use this method to get information such as HTTP headers and query strings.
`Abort`	Aborts the connection.

The Clients object

The `Hub` class has a `Clients` property that contains the following properties for communication between server and client:

PROPERTY	DESCRIPTION
`All`	Calls a method on all connected clients
`Caller`	Calls a method on the client that invoked the hub method
`Others`	Calls a method on all connected clients except the client that invoked the method

`Hub.Clients` also contains the following methods:

METHOD	DESCRIPTION
`AllExcept`	Calls a method on all connected clients except for the specified connections
`Client`	Calls a method on a specific connected client
`Clients`	Calls a method on specific connected clients
`Group`	Calls a method on all connections in the specified group
`GroupExcept`	Calls a method on all connections in the specified group, except the specified connections
`Groups`	Calls a method on multiple groups of connections
`OthersInGroup`	Calls a method on a group of connections, excluding the client that invoked the hub method
`User`	Calls a method on all connections associated with a specific user
`Users`	Calls a method on all connections associated with the specified users

Each property or method in the preceding tables returns an object with a `SendAsync` method. The `SendAsync` method allows you to supply the name and parameters of the client method to call.

Send messages to clients

To make calls to specific clients, use the properties of the `Clients` object. In the following example, there are three Hub methods:

- `SendMessage` sends a message to all connected clients, using `Clients.All`.
- `SendMessageToCaller` sends a message back to the caller, using `Clients.Caller`.
- `SendMessageToGroups` sends a message to all clients in the `SignalR Users` group.

```
public Task SendMessage(string user, string message)
{
    return Clients.All.SendAsync("ReceiveMessage", user, message);
}

public Task SendMessageToCaller(string message)
{
    return Clients.Caller.SendAsync("ReceiveMessage", message);
}

public Task SendMessageToGroup(string message)
{
    return Clients.Group("SignalR Users").SendAsync("ReceiveMessage", message);
}
```

Strongly typed hubs

A drawback of using `SendAsync` is that it relies on a magic string to specify the client method to be called. This leaves code open to runtime errors if the method name is misspelled or missing from the client.

An alternative to using `SendAsync` is to strongly type the `Hub` with Hub<T>. In the following example, the `ChatHub` client methods have been extracted out into an interface called `IChatClient`.

```
public interface IChatClient
{
    Task ReceiveMessage(string user, string message);
    Task ReceiveMessage(string message);
}
```

This interface can be used to refactor the preceding `ChatHub` example.

```
public class StronglyTypedChatHub : Hub<IChatClient>
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.ReceiveMessage(user, message);
    }

    public Task SendMessageToCaller(string message)
    {
        return Clients.Caller.ReceiveMessage(message);
    }
}
```

Using `Hub<IChatClient>` enables compile-time checking of the client methods. This prevents issues caused by using magic strings, since `Hub<T>` can only provide access to the methods defined in the interface.

Using a strongly typed `Hub<T>` disables the ability to use `SendAsync`. Any methods defined on the interface can still be defined as asynchronous. In fact, each of these methods should return a `Task`. Since it's an interface, don't use the `async` keyword. For example:

```
public interface IClient
{
    Task ClientMethod();
}
```

> **NOTE**
>
> The `Async` suffix isn't stripped from the method name. Unless your client method is defined with `.on('MyMethodAsync')`, you shouldn't use `MyMethodAsync` as a name.

Change the name of a hub method

By default, a server hub method name is the name of the .NET method. However, you can use the HubMethodName attribute to change this default and manually specify a name for the method. The client should use this name, instead of the .NET method name, when invoking the method.

```
[HubMethodName("SendMessageToUser")]
public Task DirectMessage(string user, string message)
{
    return Clients.User(user).SendAsync("ReceiveMessage", message);
}
```

Handle events for a connection

The SignalR Hubs API provides the `OnConnectedAsync` and `OnDisconnectedAsync` virtual methods to manage and track connections. Override the `OnConnectedAsync` virtual method to perform actions when a client connects to the Hub, such as adding it to a group.

```
public override async Task OnConnectedAsync()
{
    await Groups.AddToGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnConnectedAsync();
}
```

Override the `OnDisconnectedAsync` virtual method to perform actions when a client disconnects. If the client disconnects intentionally (by calling `connection.stop()`, for example), the `exception` parameter will be `null`. However, if the client is disconnected due to an error (such as a network failure), the `exception` parameter will contain an exception describing the failure.

```
public override async Task OnDisconnectedAsync(Exception exception)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnDisconnectedAsync(exception);
}
```

> **WARNING**
>
> Security warning: Exposing `ConnectionId` can lead to malicious impersonation if the SignalR server or client version is ASP.NET Core 2.2 or earlier.

Handle errors

Exceptions thrown in your hub methods are sent to the client that invoked the method. On the JavaScript client, the `invoke` method returns a JavaScript Promise. When the client receives an error with a handler attached to the promise using `catch`, it's invoked and passed as a JavaScript `Error` object.

```
connection.invoke("SendMessage", user, message).catch(err => console.error(err));
```

If your Hub throws an exception, connections aren't closed. By default, SignalR returns a generic error message to the client. For example:

```
Microsoft.AspNetCore.SignalR.HubException: An unexpected error occurred invoking 'MethodName' on the
server.
```

Unexpected exceptions often contain sensitive information, such as the name of a database server in an exception triggered when the database connection fails. SignalR doesn't expose these detailed error messages by default as a security measure. See the Security considerations article for more information on why exception details are suppressed.

If you have an exceptional condition you *do* want to propagate to the client, you can use the `HubException` class. If you throw a `HubException` from your hub method, SignalR **will** send the entire message to the client, unmodified.

```
public Task ThrowException()
{
    throw new HubException("This error will be sent to the client!");
}
```

> **NOTE**
>
> SignalR only sends the `Message` property of the exception to the client. The stack trace and other properties on the exception aren't available to the client.

Related resources

- Intro to ASP.NET Core SignalR
- JavaScript client
- Publish to Azure

Send messages from outside a hub

9/22/2020 • 2 minutes to read • Edit Online

By Mikael Mengistu

The SignalR hub is the core abstraction for sending messages to clients connected to the SignalR server. It's also possible to send messages from other places in your app using the `IHubContext` service. This article explains how to access a SignalR `IHubContext` to send notifications to clients from outside a hub.

View or download sample code (how to download)

Get an instance of IHubContext

In ASP.NET Core SignalR, you can access an instance of `IHubContext` via dependency injection. You can inject an instance of `IHubContext` into a controller, middleware, or other DI service. Use the instance to send messages to clients.

> **NOTE**
> This differs from ASP.NET 4.x SignalR which used GlobalHost to provide access to the `IHubContext`. ASP.NET Core has a dependency injection framework that removes the need for this global singleton.

Inject an instance of IHubContext in a controller

You can inject an instance of `IHubContext` into a controller by adding it to your constructor:

```
public class HomeController : Controller
{
    private readonly IHubContext<NotificationHub> _hubContext;

    public HomeController(IHubContext<NotificationHub> hubContext)
    {
        _hubContext = hubContext;
    }
}
```

Now, with access to an instance of `IHubContext`, you can call hub methods as if you were in the hub itself.

```
public async Task<IActionResult> Index()
{
    await _hubContext.Clients.All.SendAsync("Notify", $"Home page loaded at: {DateTime.Now}");
    return View();
}
```

Get an instance of IHubContext in middleware

Access the `IHubContext` within the middleware pipeline like so:

```
app.Use(async (context, next) =>
{
    var hubContext = context.RequestServices
                            .GetRequiredService<IHubContext<ChatHub>>();
    //...

    if (next != null)
    {
        await next.Invoke();
    }
});
```

> **NOTE**
>
> When hub methods are called from outside of the `Hub` class, there's no caller associated with the invocation. Therefore, there's no access to the `ConnectionId`, `Caller`, and `Others` properties.

Get an instance of IHubContext from IHost

Accessing an `IHubContext` from the web host is useful for integrating with areas outside of ASP.NET Core, for example, using third-party dependency injection frameworks:

```
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();
            var hubContext = host.Services.GetService(typeof(IHubContext<ChatHub>));
            host.Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder => {
                    webBuilder.UseStartup<Startup>();
                });
    }
```

Inject a strongly-typed HubContext

To inject a strongly-typed HubContext, ensure your Hub inherits from `Hub<T>`. Inject it using the `IHubContext<THub, T>` interface rather than `IHubContext<THub>`.

```
public class ChatController : Controller
{
    public IHubContext<ChatHub, IChatClient> _strongChatHubContext { get; }

    public ChatController(IHubContext<ChatHub, IChatClient> chatHubContext)
    {
        _strongChatHubContext = chatHubContext;
    }

    public async Task SendMessage(string message)
    {
        await _strongChatHubContext.Clients.All.ReceiveMessage(message);
    }
}
```

Related resources

- Get started
- Hubs
- Publish to Azure

Manage users and groups in SignalR

9/22/2020 • 2 minutes to read • Edit Online

By Brennan Conroy

SignalR allows messages to be sent to all connections associated with a specific user, as well as to named groups of connections.

View or download sample code (how to download)

Users in SignalR

A single user in SignalR can have multiple connections to an app. For example, a user could be connected on their desktop as well as their phone. Each device has a separate SignalR connection, but they're all associated with the same user. If a message is sent to the user, all of the connections associated with that user receive the message. The user identifier for a connection can be accessed by the `Context.UserIdentifier` property in the hub.

By default, SignalR uses the `ClaimTypes.NameIdentifier` from the `ClaimsPrincipal` associated with the connection as the user identifier. To customize this behavior, see Use claims to customize identity handling.

Send a message to a specific user by passing the user identifier to the `User` function in a hub method, as shown in the following example:

> **NOTE**
> The user identifier is case-sensitive.

```
public Task SendPrivateMessage(string user, string message)
{
    return Clients.User(user).SendAsync("ReceiveMessage", message);
}
```

Groups in SignalR

A group is a collection of connections associated with a name. Messages can be sent to all connections in a group. Groups are the recommended way to send to a connection or multiple connections because the groups are managed by the application. A connection can be a member of multiple groups. Groups are ideal for something like a chat application, where each room can be represented as a group. Connections are added to or removed from groups via the `AddToGroupAsync` and `RemoveFromGroupAsync` methods.

```
public async Task AddToGroup(string groupName)
{
    await Groups.AddToGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"{Context.ConnectionId} has joined the group
{groupName}.");
}

public async Task RemoveFromGroup(string groupName)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"{Context.ConnectionId} has left the group
{groupName}.");
}
```

Group membership isn't preserved when a connection reconnects. The connection needs to rejoin the group when it's re-established. It's not possible to count the members of a group, since this information is not available if the application is scaled to multiple servers.

To protect access to resources while using groups, use authentication and authorization functionality in ASP.NET Core. If a user is added to a group only when the credentials are valid for that group, messages sent to that group will only go to authorized users. However, groups are not a security feature. Authentication claims have features that groups do not, such as expiry and revocation. If a user's permission to access the group is revoked, the app must remove the user from the group explicitly.

> **NOTE**
> Group names are case-sensitive.

Related resources

- Get started
- Hubs
- Publish to Azure

SignalR API design considerations

9/22/2020 • 2 minutes to read • Edit Online

By Andrew Stanton-Nurse

This article provides guidance for building SignalR-based APIs.

Use custom object parameters to ensure backwards-compatibility

Adding parameters to a SignalR hub method (on either the client or the server) is a *breaking change*. This means older clients/servers will get errors when they try to invoke the method without the appropriate number of parameters. However, adding properties to a custom object parameter is **not** a breaking change. This can be used to design compatible APIs that are resilient to changes on the client or the server.

For example, consider a server-side API like the following:

```
public async Task<string> GetTotalLength(string param1)
{
    return param1.Length;
}
```

The JavaScript client calls this method using `invoke` as follows:

```
connection.invoke("GetTotalLength", "value1");
```

If you later add a second parameter to the server method, older clients won't provide this parameter value. For example:

```
public async Task<string> GetTotalLength(string param1, string param2)
{
    return param1.Length + param2.Length;
}
```

When the old client tries to invoke this method, it will get an error like this:

```
Microsoft.AspNetCore.SignalR.HubException: Failed to invoke 'GetTotalLength' due to an error on the server.
```

On the server, you'll see a log message like this:

```
System.IO.InvalidDataException: Invocation provides 1 argument(s) but target expects 2.
```

The old client only sent one parameter, but the newer server API required two parameters. Using custom objects as parameters gives you more flexibility. Let's redesign the original API to use a custom object:

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
}

public async Task GetTotalLength(TotalLengthRequest req)
{
    return req.Param1.Length;
}
```

Now, the client uses an object to call the method:

```
connection.invoke("GetTotalLength", { param1: "value1" });
```

Instead of adding a parameter, add a property to the `TotalLengthRequest` object:

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}

public async Task GetTotalLength(TotalLengthRequest req)
{
    var length = req.Param1.Length;
    if (req.Param2 != null)
    {
        length += req.Param2.Length;
    }
    return length;
}
```

When the old client sends a single parameter, the extra `Param2` property will be left `null`. You can detect a message sent by an older client by checking the `Param2` for `null` and apply a default value. A new client can send both parameters.

```
connection.invoke("GetTotalLength", { param1: "value1", param2: "value2" });
```

The same technique works for methods defined on the client. You can send a custom object from the server side:

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Message = message
    });
}
```

On the client side, you access the `Message` property rather than using a parameter:

```
connection.on("ReceiveMessage", (req) => {
    appendMessageToChatWindow(req.message);
});
```

If you later decide to add the sender of the message to the payload, add a property to the object:

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Sender = Context.User.Identity.Name,
        Message = message
    });
}
```

The older clients won't be expecting the `Sender` value, so they'll ignore it. A new client can accept it by updating to read the new property:

```
connection.on("ReceiveMessage", (req) => {
    let message = req.message;
    if (req.sender) {
        message = req.sender + ": " + message;
    }
    appendMessageToChatWindow(message);
});
```

In this case, the new client is also tolerant of an old server that doesn't provide the `Sender` value. Since the old server won't provide the `Sender` value, the client checks to see if it exists before accessing it.

Use hub filters in ASP.NET Core SignalR

9/22/2020 • 3 minutes to read • Edit Online

Hub filters:

- Are available in ASP.NET Core 5.0 or later.
- Allow logic to run before and after hub methods are invoked by clients.

This article provides guidance for writing and using hub filters.

Configure hub filters

Hub filters can be applied globally or per hub type. The order in which filters are added is the order in which the filters run. Global hub filters run before local hub filters.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(options =>
    {
        // Global filters will run first
        options.AddFilter<CustomFilter>();
    }).AddHubOptions<ChatHub>(options =>
    {
        // Local filters will run second
        options.AddFilter<CustomFilter2>();
    });
}
```

A hub filter can be added in one of the following ways:

- Add a filter by concrete type:

  ```
  hubOptions.AddFilter<TFilter>();
  ```

  This will be resolved from dependency injection (DI) or type activated.

- Add a filter by runtime type:

  ```
  hubOptions.AddFilter(typeof(TFilter));
  ```

  This will be resolved from DI or type activated.

- Add a filter by instance:

  ```
  hubOptions.AddFilter(new MyFilter());
  ```

  This instance will be used like a singleton. All hub method invocations will use the same instance.

Hub filters are created and disposed per hub invocation. If you want to store global state in the filter, or no state, add the hub filter type to DI as a singleton for better performance. Alternatively, add the filter as an instance if you can.

Create hub filters

Create a filter by declaring a class that inherits from `IHubFilter`, and add the `InvokeMethodAsync` method. There is also `OnConnectedAsync` and `OnDisconnectedAsync` that can optionally be implemented to wrap the `OnConnectedAsync` and `OnDisconnectedAsync` hub methods respectively.

```
public class CustomFilter : IHubFilter
{
    public async ValueTask<object> InvokeMethodAsync(
        HubInvocationContext invocationContext, Func<HubInvocationContext, ValueTask<object>> next)
    {
        Console.WriteLine($"Calling hub method '{invocationContext.HubMethodName}'");
        try
        {
            return await next(invocationContext);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception calling '{invocationContext.HubMethodName}'");
            throw ex;
        }
    }

    // Optional method
    public Task OnConnectedAsync(HubLifetimeContext context, Func<HubLifetimeContext, Task> next)
    {
        return next(context);
    }

    // Optional method
    public Task OnDisconnectedAsync(
        HubLifetimeContext context, Exception exception, Func<HubLifetimeContext, Exception, Task> next)
    {
        return next(context, exception);
    }
}
```

Filters are very similar to middleware. The `next` method invokes the next filter. The final filter will invoke the hub method. Filters can also store the result from awaiting `next` and run logic after the hub method has been called before returning the result from `next`.

To skip a hub method invocation in a filter, throw an exception of type `HubException` instead of calling `next`. The client will receive an error if it was expecting a result.

Use hub filters

When writing the filter logic, try to make it generic by using attributes on hub methods instead of checking for hub method names.

Consider a filter that will check a hub method argument for banned phrases and replace any phrases it finds with `***`. For this example, assume a `LanguageFilterAttribute` class is defined. The class has a property named `FilterArgument` that can be set when using the attribute.

1. Place the attribute on the hub method that has a string argument to be cleaned:

```
public class ChatHub
{
    [LanguageFilter(filterArgument: 0)]
    public async Task SendMessage(string message, string username)
    {
        await Clients.All.SendAsync("SendMessage", $"{username} says: {message}");
    }
}
```

2. Define a hub filter to check for the attribute and replace banned phrases in a hub method argument with `***`:

```
public class LanguageFilter : IHubFilter
{
    // populated from a file or inline
    private List<string> bannedPhrases = new List<string> { "async void", ".Result" };

    public async ValueTask<object> InvokeMethodAsync(HubInvocationContext invocationContext,
        Func<HubInvocationContext, ValueTask<object>> next)
    {
        var languageFilter = (LanguageFilterAttribute)Attribute.GetCustomAttribute(
            invocationContext.HubMethod, typeof(LanguageFilterAttribute));
        if (languageFilter != null &&
            invocationContext.HubMethodArguments.Count > languageFilter.FilterArgument &&
            invocationContext.HubMethodArguments[languageFilter.FilterArgument] is string str)
        {
            foreach (var bannedPhrase in bannedPhrases)
            {
                str = str.Replace(bannedPhrase, "***");
            }

            arguments = invocationContext.HubMethodArguments.ToArray();
            arguments[languageFilter.FilterArgument] = str;
            invocationContext = new HubInvocationContext(invocationContext.Context,
                invocationContext.ServiceProvider,
                invocationContext.Hub,
                invocationContext.HubMethod,
                arguments);
        }

        return await next(invocationContext);
    }
}
```

3. Register the hub filter in the `Startup.ConfigureServices` method. To avoid reinitializing the banned phrases list for every invocation, the hub filter is registered as a singleton:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.AddFilter<LanguageFilter>();
    });

    services.AddSingleton<LanguageFilter>();
}
```

The HubInvocationContext object

The `HubInvocationContext` contains information for the current hub method invocation.

PROPERTY	DESCRIPTION	TYPE
`Context`	The `HubCallerContext` contains information about the connection.	`HubCallerContext`
`Hub`	The instance of the Hub being used for this hub method invocation.	`Hub`
`HubMethodName`	The name of the hub method being invoked.	`string`
`HubMethodArguments`	The list of arguments being passed to the hub method.	`IReadOnlyList<string>`
`ServiceProvider`	The scoped service provider for this hub method invocation.	`IServiceProvider`
`HubMethod`	The hub method information.	`MethodInfo`

The HubLifetimeContext object

The `HubLifetimeContext` contains information for the `OnConnectedAsync` and `OnDisconnectedAsync` hub methods.

PROPERTY	DESCRIPTION	TYPE
`Context`	The `HubCallerContext` contains information about the connection.	`HubCallerContext`
`Hub`	The instance of the Hub being used for this hub method invocation.	`Hub`
`ServiceProvider`	The scoped service provider for this hub method invocation.	`IServiceProvider`

Authorization and filters

Authorize attributes on hub methods run before hub filters.

ASP.NET Core SignalR clients

9/22/2020 • 2 minutes to read • Edit Online

Versioning, support, and compatibility

The SignalR clients ship alongside the server components and are versioned to match. Any supported client can safely connect to any supported server, and any compatibility issues would be considered bugs to be fixed. SignalR clients are supported in the same support lifecycle as the rest of .NET Core. See the .NET Core Support Policy for details.

Many features require a compatible client **and** server. See below for a table showing the minimum versions for various features.

The 1.x versions of SignalR map to the 2.1 and 2.2 .NET Core releases and have the same lifetime. For version 3.x and above, the SignalR version exactly matches the rest of .NET and has the same support lifecycle.

SIGNALR VERSION	.NET CORE VERSION	SUPPORT LEVEL	END OF SUPPORT
1.0.x	2.1.x	Long Term Support	August 21, 2021
1.1.x	2.2.x	End Of Life	December 23, 2019
3.x or higher	*same as SignalR version*	See the the .NET Core Support Policy	

NOTE: In ASP.NET Core 3.0, the JavaScript client *moved* to the `@microsoft/signalr` npm package.

Feature distribution

The table below shows the features and support for the clients that offer real-time support. For each feature, the *minimum* version supporting this feature is listed. If no version is listed, the feature isn't supported.

FEATURE	SERVER	.NET CLIENT	JAVASCRIPT CLIENT	JAVA CLIENT
Azure SignalR Service Support	2.1.0	1.0.0	1.0.0	1.0.0
Server-to-client Streaming	2.1.0	1.0.0	1.0.0	1.0.0
Client-to-server Streaming	3.0.0	3.0.0	3.0.0	3.0.0
Automatic Reconnection (.NET, JavaScript)	3.0.0	3.0.0	3.0.0	✕
WebSockets Transport	2.1.0	1.0.0	1.0.0	1.0.0
Server-Sent Events Transport	2.1.0	1.0.0	1.0.0	✕

FEATURE	SERVER	.NET CLIENT	JAVASCRIPT CLIENT	JAVA CLIENT
Long Polling Transport	2.1.0	1.0.0	1.0.0	3.0.0
JSON Hub Protocol	2.1.0	1.0.0	1.0.0	1.0.0
MessagePack Hub Protocol	2.1.0	1.0.0	1.0.0	✕

Support for enabling additional client features is tracked in our issue tracker.

Additional resources

- Get started with SignalR for ASP.NET Core
- Supported platforms
- Hubs
- JavaScript client

ASP.NET Core SignalR .NET Client

9/22/2020 • 7 minutes to read • Edit Online

The ASP.NET Core SignalR .NET client library lets you communicate with SignalR hubs from .NET apps.

View or download sample code (how to download)

The code sample in this article is a WPF app that uses the ASP.NET Core SignalR .NET client.

Install the SignalR .NET client package

The Microsoft.AspNetCore.SignalR.Client package is required for .NET clients to connect to SignalR hubs.

- Visual Studio
- .NET Core CLI

To install the client library, run the following command in the **Package Manager Console** window:

```
Install-Package Microsoft.AspNetCore.SignalR.Client
```

Connect to a hub

To establish a connection, create a `HubConnectionBuilder` and call `Build`. The hub URL, protocol, transport type, log level, headers, and other options can be configured while building a connection. Configure any required options by inserting any of the `HubConnectionBuilder` methods into `Build`. Start the connection with `StartAsync`.

```
using System;
using System.Threading.Tasks;
using System.Windows;
using Microsoft.AspNetCore.SignalR.Client;

namespace SignalRChatClient
{
    public partial class MainWindow : Window
    {
        HubConnection connection;
        public MainWindow()
        {
            InitializeComponent();

            connection = new HubConnectionBuilder()
                .WithUrl("http://localhost:53353/ChatHub")
                .Build();

            connection.Closed += async (error) =>
            {
                await Task.Delay(new Random().Next(0,5) * 1000);
                await connection.StartAsync();
            };
        }

        private async void connectButton_Click(object sender, RoutedEventArgs e)
        {
            connection.On<string, string>("ReceiveMessage", (user, message) =>
            {
                this.Dispatcher.Invoke(() =>
                {
                   var newMessage = $"{user}: {message}";
                   messagesList.Items.Add(newMessage);
                });
            });

            try
            {
                await connection.StartAsync();
                messagesList.Items.Add("Connection started");
                connectButton.IsEnabled = false;
                sendButton.IsEnabled = true;
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
        }

        private async void sendButton_Click(object sender, RoutedEventArgs e)
        {
            try
            {
                await connection.InvokeAsync("SendMessage",
                    userTextBox.Text, messageTextBox.Text);
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
        }
    }
}
```

Handle lost connection

Automatically reconnect

The HubConnection can be configured to automatically reconnect using the `WithAutomaticReconnect` method on the HubConnectionBuilder. It won't automatically reconnect by default.

```
HubConnection connection= new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect()
    .Build();
```

Without any parameters, `WithAutomaticReconnect()` configures the client to wait 0, 2, 10, and 30 seconds respectively before trying each reconnect attempt, stopping after four failed attempts.

Before starting any reconnect attempts, the `HubConnection` will transition to the `HubConnectionState.Reconnecting` state and fire the `Reconnecting` event. This provides an opportunity to warn users that the connection has been lost and to disable UI elements. Non-interactive apps can start queuing or dropping messages.

```
connection.Reconnecting += error =>
{
    Debug.Assert(connection.State == HubConnectionState.Reconnecting);

    // Notify users the connection was lost and the client is reconnecting.
    // Start queuing or dropping messages.

    return Task.CompletedTask;
};
```

If the client successfully reconnects within its first four attempts, the `HubConnection` will transition back to the `Connected` state and fire the `Reconnected` event. This provides an opportunity to inform users the connection has been reestablished and dequeue any queued messages.

Since the connection looks entirely new to the server, a new `ConnectionId` will be provided to the `Reconnected` event handlers.

WARNING

The `Reconnected` event handler's `connectionId` parameter will be null if the `HubConnection` was configured to skip negotiation.

```
connection.Reconnected += connectionId =>
{
    Debug.Assert(connection.State == HubConnectionState.Connected);

    // Notify users the connection was reestablished.
    // Start dequeuing messages queued while reconnecting if any.

    return Task.CompletedTask;
};
```

`WithAutomaticReconnect()` won't configure the `HubConnection` to retry initial start failures, so start failures need to be handled manually:

```
public static async Task<bool> ConnectWithRetryAsync(HubConnection connection, CancellationToken token)
{
    // Keep trying to until we can start or the token is canceled.
    while (true)
    {
        try
        {
            await connection.StartAsync(token);
            Debug.Assert(connection.State == HubConnectionState.Connected);
            return true;
        }
        catch when (token.IsCancellationRequested)
        {
            return false;
        }
        catch
        {
            // Failed to connect, trying again in 5000 ms.
            Debug.Assert(connection.State == HubConnectionState.Disconnected);
            await Task.Delay(5000);
        }
    }
}
```

If the client doesn't successfully reconnect within its first four attempts, the `HubConnection` will transition to the `Disconnected` state and fire the Closed event. This provides an opportunity to attempt to restart the connection manually or inform users the connection has been permanently lost.

```
connection.Closed += error =>
{
    Debug.Assert(connection.State == HubConnectionState.Disconnected);

    // Notify users the connection has been closed or manually try to restart the connection.

    return Task.CompletedTask;
};
```

In order to configure a custom number of reconnect attempts before disconnecting or change the reconnect timing, `WithAutomaticReconnect` accepts an array of numbers representing the delay in milliseconds to wait before starting each reconnect attempt.

```
HubConnection connection= new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect(new[] { TimeSpan.Zero, TimeSpan.Zero, TimeSpan.FromSeconds(10) })
    .Build();

    // .WithAutomaticReconnect(new[] { TimeSpan.Zero, TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(10),
TimeSpan.FromSeconds(30) }) yields the default behavior.
```

The preceding example configures the `HubConnection` to start attempting reconnects immediately after the connection is lost. This is also true for the default configuration.

If the first reconnect attempt fails, the second reconnect attempt will also start immediately instead of waiting 2 seconds like it would in the default configuration.

If the second reconnect attempt fails, the third reconnect attempt will start in 10 seconds which is again like the default configuration.

The custom behavior then diverges again from the default behavior by stopping after the third reconnect attempt failure. In the default configuration there would be one more reconnect attempt in another 30 seconds.

If you want even more control over the timing and number of automatic reconnect attempts, `WithAutomaticReconnect` accepts an object implementing the `IRetryPolicy` interface, which has a single method named `NextRetryDelay`.

`NextRetryDelay` takes a single argument with the type `RetryContext`. The `RetryContext` has three properties: `PreviousRetryCount`, `ElapsedTime` and `RetryReason`, which are a `long`, a `TimeSpan` and an `Exception` respectively. Before the first reconnect attempt, both `PreviousRetryCount` and `ElapsedTime` will be zero, and the `RetryReason` will be the Exception that caused the connection to be lost. After each failed retry attempt, `PreviousRetryCount` will be incremented by one, `ElapsedTime` will be updated to reflect the amount of time spent reconnecting so far, and the `RetryReason` will be the Exception that caused the last reconnect attempt to fail.

`NextRetryDelay` must return either a TimeSpan representing the time to wait before the next reconnect attempt or `null` if the `HubConnection` should stop reconnecting.

```
public class RandomRetryPolicy : IRetryPolicy
{
    private readonly Random _random = new Random();

    public TimeSpan? NextRetryDelay(RetryContext retryContext)
    {
        // If we've been reconnecting for less than 60 seconds so far,
        // wait between 0 and 10 seconds before the next reconnect attempt.
        if (retryContext.ElapsedTime < TimeSpan.FromSeconds(60))
        {
            return TimeSpan.FromSeconds(_random.NextDouble() * 10);
        }
        else
        {
            // If we've been reconnecting for more than 60 seconds so far, stop reconnecting.
            return null;
        }
    }
}
```

```
HubConnection connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect(new RandomRetryPolicy())
    .Build();
```

Alternatively, you can write code that will reconnect your client manually as demonstrated in Manually reconnect.

Manually reconnect

> **WARNING**
> Prior to 3.0, the .NET client for SignalR doesn't automatically reconnect. You must write code that will reconnect your client manually.

Use the Closed event to respond to a lost connection. For example, you might want to automate reconnection.

The `Closed` event requires a delegate that returns a `Task`, which allows async code to run without using `async void`. To satisfy the delegate signature in a `Closed` event handler that runs synchronously, return `Task.CompletedTask`:

```
connection.Closed += (error) => {
    // Do your close logic.
    return Task.CompletedTask;
};
```

The main reason for the async support is so you can restart the connection. Starting a connection is an async action.

In a `Closed` handler that restarts the connection, consider waiting for some random delay to prevent overloading the server, as shown in the following example:

```
connection.Closed += async (error) =>
{
    await Task.Delay(new Random().Next(0,5) * 1000);
    await connection.StartAsync();
};
```

Call hub methods from client

`InvokeAsync` calls methods on the hub. Pass the hub method name and any arguments defined in the hub method to `InvokeAsync`. SignalR is asynchronous, so use `async` and `await` when making the calls.

```
await connection.InvokeAsync("SendMessage",
    userTextBox.Text, messageTextBox.Text);
```

The `InvokeAsync` method returns a `Task` which completes when the server method returns. The return value, if any, is provided as the result of the `Task`. Any exceptions thrown by the method on the server produce a faulted `Task`. Use `await` syntax to wait for the server method to complete and `try...catch` syntax to handle errors.

The `SendAsync` method returns a `Task` which completes when the message has been sent to the server. No return value is provided since this `Task` doesn't wait until the server method completes. Any exceptions thrown on the client while sending the message produce a faulted `Task`. Use `await` and `try...catch` syntax to handle send errors.

> **NOTE**
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

Call client methods from hub

Define methods the hub calls using `connection.On` after building, but before starting the connection.

```
connection.On<string, string>("ReceiveMessage", (user, message) =>
{
    this.Dispatcher.Invoke(() =>
    {
       var newMessage = $"{user}: {message}";
       messagesList.Items.Add(newMessage);
    });
});
```

The preceding code in `connection.On` runs when server-side code calls it using the `SendAsync` method.

```
public async Task SendMessage(string user, string message)
{
    await Clients.All.SendAsync("ReceiveMessage", user,message);
}
```

Error handling and logging

Handle errors with a try-catch statement. Inspect the `Exception` object to determine the proper action to take after an error occurs.

```
try
{
    await connection.InvokeAsync("SendMessage",
        userTextBox.Text, messageTextBox.Text);
}
catch (Exception ex)
{
    messagesList.Items.Add(ex.Message);
}
```

Additional resources

- Hubs
- JavaScript client
- Publish to Azure
- Azure SignalR Service serverless documentation

ASP.NET Core SignalR Java client

9/22/2020 • 3 minutes to read • Edit Online

By Mikael Mengistu

The Java client enables connecting to an ASP.NET Core SignalR server from Java code, including Android apps. Like the JavaScript client and the .NET client, the Java client enables you to receive and send messages to a hub in real time. The Java client is available in ASP.NET Core 2.2 and later.

The sample Java console app referenced in this article uses the SignalR Java client.

View or download sample code (how to download)

Install the SignalR Java client package

The *signalr-1.0.0* JAR file allows clients to connect to SignalR hubs. To find the latest JAR file version number, see the Maven search results.

If using Gradle, add the following line to the `dependencies` section of your *build.gradle* file:

```
implementation 'com.microsoft.signalr:signalr:1.0.0'
```

If using Maven, add the following lines inside the `<dependencies>` element of your *pom.xml* file:

```
<dependency>
    <groupId>com.microsoft.signalr</groupId>
    <artifactId>signalr</artifactId>
    <version>1.0.0</version>
</dependency>
```

Connect to a hub

To establish a `HubConnection`, the `HubConnectionBuilder` should be used. The hub URL and log level can be configured while building a connection. Configure any required options by calling any of the `HubConnectionBuilder` methods before `build`. Start the connection with `start`.

```
HubConnection hubConnection = HubConnectionBuilder.create(input)
        .build();
```

Call hub methods from client

A call to `send` invokes a hub method. Pass the hub method name and any arguments defined in the hub method to `send`.

```
hubConnection.send("Send", input);
```

> **NOTE**
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

Call client methods from hub

Use `hubConnection.on` to define methods on the client that the hub can call. Define the methods after building but before starting the connection.

```
hubConnection.on("Send", (message) -> {
    System.out.println("New Message: " + message);
}, String.class);
```

Add logging

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to chose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

This can safely be ignored.

Android development notes

With regards to Android SDK compatibility for the SignalR client features, consider the following items when specifying your target Android SDK version:

- The SignalR Java Client will run on Android API Level 16 and later.
- Connecting through the Azure SignalR Service will require Android API Level 20 and later because the Azure SignalR Service requires TLS 1.2 and doesn't support SHA-1-based cipher suites. Android added support for SHA-256 (and above) cipher suites in API Level 20.

Configure bearer token authentication

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an "access token factory" to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

```
HubConnection hubConnection = HubConnectionBuilder.create("YOUR HUB URL HERE")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Known limitations

- Only the JSON protocol is supported.
- Transport fallback and the Server Sent Events transport aren't supported.

- Only the JSON protocol is supported.
- Only the WebSockets transport is supported.
- Streaming isn't supported yet.

Additional resources

- Java API reference
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- Publish an ASP.NET Core SignalR app to Azure App Service
- Azure SignalR Service serverless documentation

ASP.NET Core SignalR JavaScript client

9/22/2020 • 14 minutes to read • Edit Online

By Rachel Appel

The ASP.NET Core SignalR JavaScript client library enables developers to call server-side hub code.

View or download sample code (how to download)

Install the SignalR client package

The SignalR JavaScript client library is delivered as an npm package. The following sections outline different ways to install the client library.

Install with npm

For Visual Studio, run the following commands from **Package Manager Console** while in the root folder. For Visual Studio Code, run the following commands from the **Integrated Terminal**.

```
npm init -y
npm install @microsoft/signalr
```

npm installs the package contents in the *node_modules\@microsoft\signalr\dist\browser* folder. Create a new folder named *signalr* under the *wwwroot\lib* folder. Copy the *signalr.js* file to the *wwwroot\lib\signalr* folder.

Reference the SignalR JavaScript client in the `<script>` element. For example:

```
<script src="~/lib/signalr/signalr.js"></script>
```

Use a Content Delivery Network (CDN)

To use the client library without the npm prerequisite, reference a CDN-hosted copy of the client library. For example:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/3.1.7/signalr.min.js"></script>
```

The client library is available on the following CDNs:

- cdnjs
- jsDelivr
- unpkg

Install with LibMan

LibMan can be used to install specific client library files from the CDN-hosted client library. For example, only add the minified JavaScript file to the project. For details on that approach, see Add the SignalR client library.

Connect to a hub

The following code creates and starts a connection. The hub's name is case insensitive:

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();

async function start() {
    try {
        await connection.start();
        console.log("SignalR Connected.");
    } catch (err) {
        console.log(err);
        setTimeout(start, 5000);
    }
};

connection.onclose(start);

// Start the connection.
start();
```

Cross-origin connections

Typically, browsers load connections from the same domain as the requested page. However, there are occasions when a connection to another domain is required.

To prevent a malicious site from reading sensitive data from another site, cross-origin connections are disabled by default. To allow a cross-origin request, enable it in the `Startup` class:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
            services.AddSignalR();

            services.AddCors(options =>
            {
                options.AddDefaultPolicy(builder =>
                {
                    builder.WithOrigins("https://example.com")
                        .AllowCredentials();
                });
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
            }

            app.UseStaticFiles();
            app.UseRouting();

            app.UseCors();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
                endpoints.MapHub<ChatHub>("/chathub");
            });
        }
    }
}
```

Call hub methods from the client

JavaScript clients call public methods on hubs via the invoke method of the HubConnection. The `invoke` method accepts:

- The name of the hub method.
- Any arguments defined in the hub method.

In the following example, the method name on the hub is `SendMessage`. The second and third arguments passed to `invoke` map to the hub method's `user` and `message` arguments:

```
try {
    await connection.invoke("SendMessage", user, message);
} catch (err) {
    console.error(err);
}
```

> **NOTE**
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

The `invoke` method returns a JavaScript Promise. The `Promise` is resolved with the return value (if any) when the method on the server returns. If the method on the server throws an error, the `Promise` is rejected with the error message. Use `async` and `await` or the `Promise`'s `then` and `catch` methods to handle these cases.

JavaScript clients can also call public methods on hubs via the the send method of the `HubConnection`. Unlike the `invoke` method, the `send` method doesn't wait for a response from the server. The `send` method returns a JavaScript `Promise`. The `Promise` is resolved when the message has been sent to the server. If there is an error sending the message, the `Promise` is rejected with the error message. Use `async` and `await` or the `Promise`'s `then` and `catch` methods to handle these cases.

> **NOTE**
> Using `send` doesn't wait until the server has received the message. Consequently, it's not possible to return data or errors from the server.

Call client methods from the hub

To receive messages from the hub, define a method using the on method of the `HubConnection`.

- The name of the JavaScript client method.
- Arguments the hub passes to the method.

In the following example, the method name is `ReceiveMessage`. The argument names are `user` and `message`:

```
connection.on("ReceiveMessage", (user, message) => {
    const li = document.createElement("li");
    li.textContent = `${user}: ${message}`;
    document.getElementById("messageList").appendChild(li);
});
```

The preceding code in `connection.on` runs when server-side code calls it using the SendAsync method:

```
public async Task SendMessage(string user, string message)
{
    await Clients.All.SendAsync("ReceiveMessage", user, message);
}
```

SignalR determines which client method to call by matching the method name and arguments defined in `SendAsync` and `connection.on`.

> **NOTE**
> As a best practice, call the start method on the `HubConnection` after `on`. Doing so ensures your handlers are registered before any messages are received.

Error handling and logging

Use `try` and `catch` with `async` and `await` or the `Promise`'s `catch` method to handle client-side errors. Use `console.error` to output errors to the browser's console:

```
try {
    await connection.invoke("SendMessage", user, message);
} catch (err) {
    console.error(err);
}
```

Set up client-side log tracing by passing a logger and type of event to log when the connection is made. Messages are logged with the specified log level and higher. Available log levels are as follows:

- `signalR.LogLevel.Error`: Error messages. Logs `Error` messages only.
- `signalR.LogLevel.Warning`: Warning messages about potential errors. Logs `Warning`, and `Error` messages.
- `signalR.LogLevel.Information`: Status messages without errors. Logs `Information`, `Warning`, and `Error` messages.
- `signalR.LogLevel.Trace`: Trace messages. Logs everything, including data transported between hub and client.

Use the configureLogging method on HubConnectionBuilder to configure the log level. Messages are logged to the browser console:

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Reconnect clients

Automatically reconnect

The JavaScript client for SignalR can be configured to automatically reconnect using the `withAutomaticReconnect` method on HubConnectionBuilder. It won't automatically reconnect by default.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

Without any parameters, `withAutomaticReconnect()` configures the client to wait 0, 2, 10, and 30 seconds respectively before trying each reconnect attempt, stopping after four failed attempts.

Before starting any reconnect attempts, the `HubConnection` will transition to the `HubConnectionState.Reconnecting` state and fire its `onreconnecting` callbacks instead of transitioning to the `Disconnected` state and triggering its `onclose` callbacks like a `HubConnection` without automatic reconnect configured. This provides an opportunity to warn users that the connection has been lost and to disable UI

elements.

```
connection.onreconnecting(error => {
    console.assert(connection.state === signalR.HubConnectionState.Reconnecting);

    document.getElementById("messageInput").disabled = true;

    const li = document.createElement("li");
    li.textContent = `Connection lost due to error "${error}". Reconnecting.`;
    document.getElementById("messagesList").appendChild(li);
});
```

If the client successfully reconnects within its first four attempts, the `HubConnection` will transition back to the `Connected` state and fire its `onreconnected` callbacks. This provides an opportunity to inform users the connection has been reestablished.

Since the connection looks entirely new to the server, a new `connectionId` will be provided to the `onreconnected` callback.

WARNING

The `onreconnected` callback's `connectionId` parameter will be undefined if the `HubConnection` was configured to skip negotiation.

```
connection.onreconnected(connectionId => {
    console.assert(connection.state === signalR.HubConnectionState.Connected);

    document.getElementById("messageInput").disabled = false;

    const li = document.createElement("li");
    li.textContent = `Connection reestablished. Connected with connectionId "${connectionId}".`;
    document.getElementById("messagesList").appendChild(li);
});
```

`withAutomaticReconnect()` won't configure the `HubConnection` to retry initial start failures, so start failures need to be handled manually:

```
async function start() {
    try {
        await connection.start();
        console.assert(connection.state === signalR.HubConnectionState.Connected);
        console.log("SignalR Connected.");
    } catch (err) {
        console.assert(connection.state === signalR.HubConnectionState.Disconnected);
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};
```

If the client doesn't successfully reconnect within its first four attempts, the `HubConnection` will transition to the `Disconnected` state and fire its onclose callbacks. This provides an opportunity to inform users the connection has been permanently lost and recommend refreshing the page:

```
connection.onclose(error => {
    console.assert(connection.state === signalR.HubConnectionState.Disconnected);

    document.getElementById("messageInput").disabled = true;

    const li = document.createElement("li");
    li.textContent = `Connection closed due to error "${error}". Try refreshing this page to restart the
connection.`;
    document.getElementById("messagesList").appendChild(li);
});
```

In order to configure a custom number of reconnect attempts before disconnecting or change the reconnect timing, `withAutomaticReconnect` accepts an array of numbers representing the delay in milliseconds to wait before starting each reconnect attempt.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect([0, 0, 10000])
    .build();

    // .withAutomaticReconnect([0, 2000, 10000, 30000]) yields the default behavior
```

The preceding example configures the `HubConnection` to start attempting reconnects immediately after the connection is lost. This is also true for the default configuration.

If the first reconnect attempt fails, the second reconnect attempt will also start immediately instead of waiting 2 seconds like it would in the default configuration.

If the second reconnect attempt fails, the third reconnect attempt will start in 10 seconds which is again like the default configuration.

The custom behavior then diverges again from the default behavior by stopping after the third reconnect attempt failure instead of trying one more reconnect attempt in another 30 seconds like it would in the default configuration.

If you want even more control over the timing and number of automatic reconnect attempts, `withAutomaticReconnect` accepts an object implementing the `IRetryPolicy` interface, which has a single method named `nextRetryDelayInMilliseconds`.

`nextRetryDelayInMilliseconds` takes a single argument with the type `RetryContext`. The `RetryContext` has three properties: `previousRetryCount`, `elapsedMilliseconds` and `retryReason` which are a `number`, a `number` and an `Error` respectively. Before the first reconnect attempt, both `previousRetryCount` and `elapsedMilliseconds` will be zero, and the `retryReason` will be the Error that caused the connection to be lost. After each failed retry attempt, `previousRetryCount` will be incremented by one, `elapsedMilliseconds` will be updated to reflect the amount of time spent reconnecting so far in milliseconds, and the `retryReason` will be the Error that caused the last reconnect attempt to fail.

`nextRetryDelayInMilliseconds` must return either a number representing the number of milliseconds to wait before the next reconnect attempt or `null` if the `HubConnection` should stop reconnecting.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect({
        nextRetryDelayInMilliseconds: retryContext => {
            if (retryContext.elapsedMilliseconds < 60000) {
                // If we've been reconnecting for less than 60 seconds so far,
                // wait between 0 and 10 seconds before the next reconnect attempt.
                return Math.random() * 10000;
            } else {
                // If we've been reconnecting for more than 60 seconds so far, stop reconnecting.
                return null;
            }
        }
    })
    .build();
```

Alternatively, you can write code that will reconnect your client manually as demonstrated in Manually reconnect.

Manually reconnect

The following code demonstrates a typical manual reconnection approach:

1. A function (in this case, the `start` function) is created to start the connection.
2. Call the `start` function in the connection's `onclose` event handler.

```
async function start() {
    try {
        await connection.start();
        console.log("SignalR Connected.");
    } catch (err) {
        console.log(err);
        setTimeout(start, 5000);
    }
};

connection.onclose(start);
```

A real-world implementation would use an exponential back-off or retry a specified number of times before giving up.

Additional resources

- JavaScript API reference
- JavaScript tutorial
- WebPack and TypeScript tutorial
- Hubs
- .NET client
- Publish to Azure
- Cross-Origin Requests (CORS)
- Azure SignalR Service serverless documentation

By Rachel Appel

The ASP.NET Core SignalR JavaScript client library enables developers to call server-side hub code.

View or download sample code (how to download)

Install the SignalR client package

The SignalR JavaScript client library is delivered as an npm package. The following sections outline different ways to install the client library.

Install with npm

If using Visual Studio, run the following commands from **Package Manager Console** while in the root folder. For Visual Studio Code, run the following commands from the **Integrated Terminal**.

```
npm init -y
npm install @aspnet/signalr
```

npm installs the package contents in the *node_modules\@aspnet\signalr\dist\browser* folder. Create a new folder named *signalr* under the *wwwroot\lib* folder. Copy the *signalr.js* file to the *wwwroot\lib\signalr* folder.

Reference the SignalR JavaScript client in the `<script>` element. For example:

```
<script src="~/lib/signalr/signalr.js"></script>
```

Use a Content Delivery Network (CDN)

To use the client library without the npm prerequisite, reference a CDN-hosted copy of the client library. For example:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/3.1.3/signalr.min.js"></script>
```

The client library is available on the following CDNs:

- cdnjs
- jsDelivr
- unpkg

Install with LibMan

LibMan can be used to install specific client library files from the CDN-hosted client library. For example, only add the minified JavaScript file to the project. For details on that approach, see Add the SignalR client library.

Connect to a hub

The following code creates and starts a connection. The hub's name is case insensitive.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .configureLogging(signalR.LogLevel.Information)
    .build();

async function start() {
    try {
        await connection.start();
        console.log("connected");
    } catch (err) {
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};

connection.onclose(async () => {
    await start();
});

// Start the connection.
start();

/* this is here to show an alternative to start, with a then
connection.start().then(() => console.log("connected"));
*/

/* this is here to show another alternative to start, with a catch
connection.start().catch(err => console.error(err));
*/
```

Cross-origin connections

Typically, browsers load connections from the same domain as the requested page. However, there are occasions when a connection to another domain is required.

To prevent a malicious site from reading sensitive data from another site, cross-origin connections are disabled by default. To allow a cross-origin request, enable it in the `Startup` class.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using SignalRChat.Hubs;

namespace SignalRChat
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });

            services.AddMvc();

            services.AddCors(options => options.AddPolicy("CorsPolicy",
            builder =>
            {
                builder.AllowAnyMethod().AllowAnyHeader()
                       .WithOrigins("http://localhost:55830")
                       .AllowCredentials();
            }));

            services.AddSignalR();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseBrowserLink();
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            app.UseCors("CorsPolicy");
            app.UseSignalR(routes =>
            {
                routes.MapHub<ChatHub>("/chathub");
            });
            app.UseMvc();
        }
    }
}
```

Call hub methods from client

JavaScript clients call public methods on hubs via the invoke method of the HubConnection. The `invoke` method accepts two arguments:

- The name of the hub method. In the following example, the method name on the hub is `SendMessage`.
- Any arguments defined in the hub method. In the following example, the argument name is `message`. The example code uses arrow function syntax that is supported in current versions of all major browsers except Internet Explorer.

```
connection.invoke("SendMessage", user, message).catch(err => console.error(err));
```

> **NOTE**
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

The `invoke` method returns a JavaScript Promise. The `Promise` is resolved with the return value (if any) when the method on the server returns. If the method on the server throws an error, the `Promise` is rejected with the error message. Use the `then` and `catch` methods on the `Promise` itself to handle these cases (or `await` syntax).

The `send` method returns a JavaScript `Promise`. The `Promise` is resolved when the message has been sent to the server. If there is an error sending the message, the `Promise` is rejected with the error message. Use the `then` and `catch` methods on the `Promise` itself to handle these cases (or `await` syntax).

> **NOTE**
> Using `send` doesn't wait until the server has received the message. Consequently, it's not possible to return data or errors from the server.

Call client methods from hub

To receive messages from the hub, define a method using the on method of the `HubConnection`.

- The name of the JavaScript client method. In the following example, the method name is `ReceiveMessage`.
- Arguments the hub passes to the method. In the following example, the argument value is `message`.

```
connection.on("ReceiveMessage", (user, message) => {
    const encodedMsg = `${user} says ${message}`;
    const li = document.createElement("li");
    li.textContent = encodedMsg;
    document.getElementById("messagesList").appendChild(li);
});
```

The preceding code in `connection.on` runs when server-side code calls it using the SendAsync method.

```
public async Task SendMessage(string user, string message)
{
    await Clients.All.SendAsync("ReceiveMessage", user, message);
}
```

SignalR determines which client method to call by matching the method name and arguments defined in `SendAsync` and `connection.on`.

> **NOTE**
> As a best practice, call the start method on the `HubConnection` after `on`. Doing so ensures your handlers are registered before any messages are received.

Error handling and logging

Chain a `catch` method to the end of the `start` method to handle client-side errors. Use `console.error` to output errors to the browser's console.

```
connection.start().catch(err => console.error(err));
```

Set up client-side log tracing by passing a logger and type of event to log when the connection is made. Messages are logged with the specified log level and higher. Available log levels are as follows:

- `signalR.LogLevel.Error`: Error messages. Logs `Error` messages only.
- `signalR.LogLevel.Warning`: Warning messages about potential errors. Logs `Warning`, and `Error` messages.
- `signalR.LogLevel.Information`: Status messages without errors. Logs `Information`, `Warning`, and `Error` messages.
- `signalR.LogLevel.Trace`: Trace messages. Logs everything, including data transported between hub and client.

Use the configureLogging method on HubConnectionBuilder to configure the log level. Messages are logged to the browser console.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Reconnect clients

Manually reconnect

> **WARNING**
> Prior to 3.0, the JavaScript client for SignalR doesn't automatically reconnect. You must write code that will reconnect your client manually.

The following code demonstrates a typical manual reconnection approach:

1. A function (in this case, the `start` function) is created to start the connection.
2. Call the `start` function in the connection's `onclose` event handler.

```
async function start() {
    try {
        await connection.start();
        console.log("connected");
    } catch (err) {
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};

connection.onclose(async () => {
    await start();
});
```

A real-world implementation would use an exponential back-off or retry a specified number of times before giving up.

Additional resources

- JavaScript API reference
- JavaScript tutorial
- WebPack and TypeScript tutorial
- Hubs
- .NET client
- Publish to Azure
- Cross-Origin Requests (CORS)
- Azure SignalR Service serverless documentation

ASP.NET Core SignalR hosting and scaling

9/22/2020 • 5 minutes to read • Edit Online

By Andrew Stanton-Nurse, Brady Gaster, and Tom Dykstra

This article explains hosting and scaling considerations for high-traffic apps that use ASP.NET Core SignalR.

Sticky Sessions

SignalR requires that all HTTP requests for a specific connection be handled by the same server process. When SignalR is running on a server farm (multiple servers), "sticky sessions" must be used. "Sticky sessions" are also called session affinity by some load balancers. Azure App Service uses Application Request Routing (ARR) to route requests. Enabling the "ARR Affinity" setting in your Azure App Service will enable "sticky sessions". The only circumstances in which sticky sessions are not required are:

1. When hosting on a single server, in a single process.
2. When using the Azure SignalR Service.
3. When all clients are configured to **only** use WebSockets, **and** the SkipNegotiation setting is enabled in the client configuration.

In all other circumstances (including when the Redis backplane is used), the server environment must be configured for sticky sessions.

For guidance on configuring Azure App Service for SignalR, see Publish an ASP.NET Core SignalR app to Azure App Service.

TCP connection resources

The number of concurrent TCP connections that a web server can support is limited. Standard HTTP clients use *ephemeral* connections. These connections can be closed when the client goes idle and reopened later. On the other hand, a SignalR connection is *persistent*. SignalR connections stay open even when the client goes idle. In a high-traffic app that serves many clients, these persistent connections can cause servers to hit their maximum number of connections.

Persistent connections also consume some additional memory, to track each connection.

The heavy use of connection-related resources by SignalR can affect other web apps that are hosted on the same server. When SignalR opens and holds the last available TCP connections, other web apps on the same server also have no more connections available to them.

If a server runs out of connections, you'll see random socket errors and connection reset errors. For example:

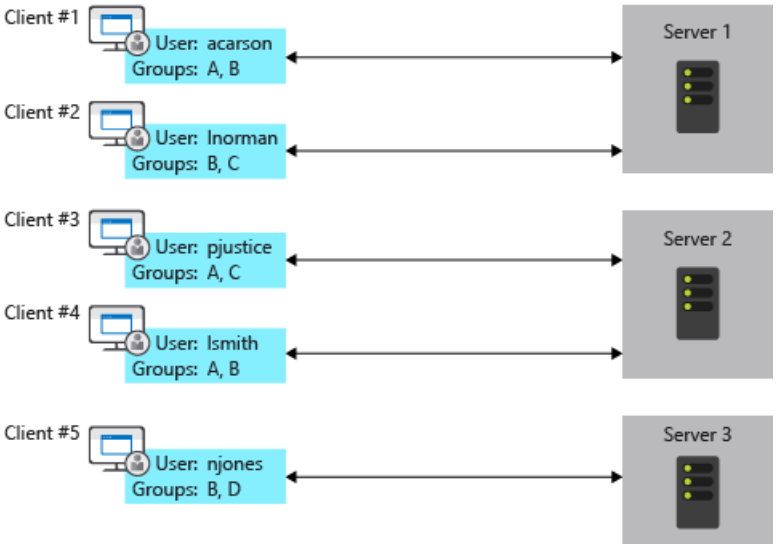
```
An attempt was made to access a socket in a way forbidden by its access permissions...
```

To keep SignalR resource usage from causing errors in other web apps, run SignalR on different servers than your other web apps.

To keep SignalR resource usage from causing errors in a SignalR app, scale out to limit the number of connections a server has to handle.

Scale out

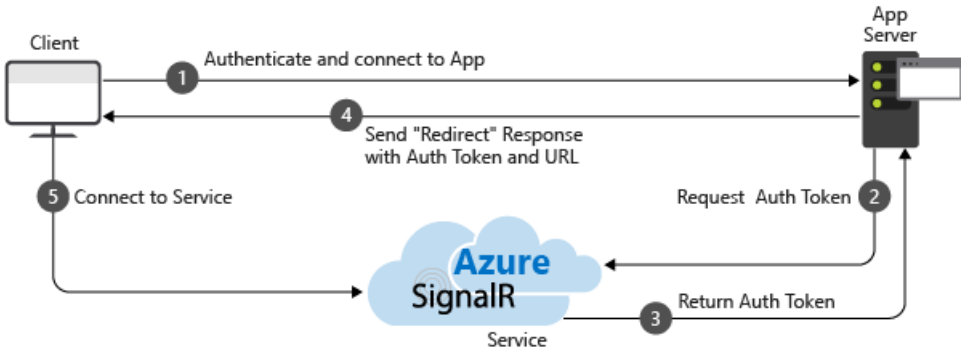
An app that uses SignalR needs to keep track of all its connections, which creates problems for a server farm. Add a server, and it gets new connections that the other servers don't know about. For example, SignalR on each server in the following diagram is unaware of the connections on the other servers. When SignalR on one of the servers wants to send a message to all clients, the message only goes to the clients connected to that server.



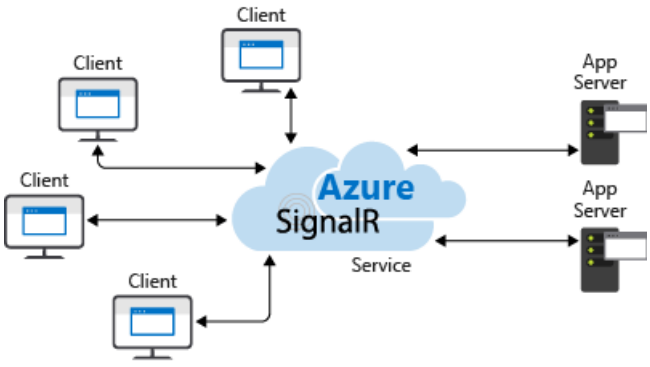
The options for solving this problem are the Azure SignalR Service and Redis backplane.

Azure SignalR Service

The Azure SignalR Service is a proxy rather than a backplane. Each time a client initiates a connection to the server, the client is redirected to connect to the service. That process is illustrated in the following diagram:



The result is that the service manages all of the client connections, while each server needs only a small constant number of connections to the service, as shown in the following diagram:



This approach to scale-out has several advantages over the Redis backplane alternative:

- Sticky sessions, also known as client affinity, is not required, because clients are immediately redirected to the Azure SignalR Service when they connect.
- A SignalR app can scale out based on the number of messages sent, while the Azure SignalR Service scales to

handle any number of connections. For example, there could be thousands of clients, but if only a few messages per second are sent, the SignalR app won't need to scale out to multiple servers just to handle the connections themselves.

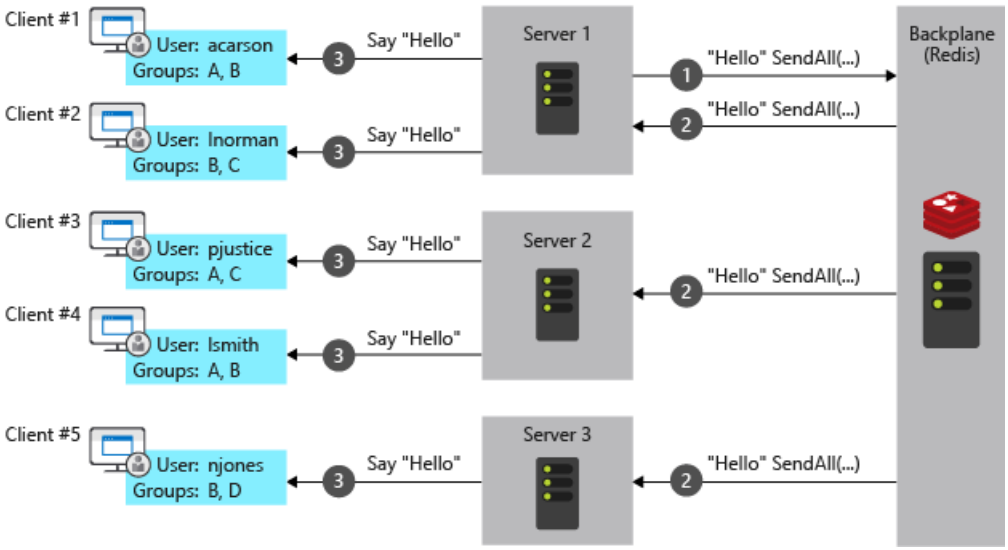
- A SignalR app won't use significantly more connection resources than a web app without SignalR.

For these reasons, we recommend the Azure SignalR Service for all ASP.NET Core SignalR apps hosted on Azure, including App Service, VMs, and containers.

For more information see the Azure SignalR Service documentation.

Redis backplane

Redis is an in-memory key-value store that supports a messaging system with a publish/subscribe model. The SignalR Redis backplane uses the pub/sub feature to forward messages to other servers. When a client makes a connection, the connection information is passed to the backplane. When a server wants to send a message to all clients, it sends to the backplane. The backplane knows all connected clients and which servers they're on. It sends the message to all clients via their respective servers. This process is illustrated in the following diagram:



The Redis backplane is the recommended scale-out approach for apps hosted on your own infrastructure. If there is significant connection latency between your data center and an Azure data center, Azure SignalR Service may not be a practical option for on-premises apps with low latency or high throughput requirements.

The Azure SignalR Service advantages noted earlier are disadvantages for the Redis backplane:

- Sticky sessions, also known as client affinity, is required, except when **both** of the following are true:
  - All clients are configured to **only** use WebSockets.
  - The SkipNegotiation setting is enabled in the client configuration. Once a connection is initiated on a server, the connection has to stay on that server.
- A SignalR app must scale out based on number of clients even if few messages are being sent.
- A SignalR app uses significantly more connection resources than a web app without SignalR.

IIS limitations on Windows client OS

Windows 10 and Windows 8.x are client operating systems. IIS on client operating systems has a limit of 10 concurrent connections. SignalR's connections are:

- Transient and frequently re-established.
- **Not** disposed immediately when no longer used.

The preceding conditions make it likely to hit the 10 connection limit on a client OS. When a client OS is used for

development, we recommend:

- Avoid IIS.
- Use Kestrel or IIS Express as deployment targets.

Linux with Nginx

Set the proxy's `Connection` and `Upgrade` headers to the following for SignalR WebSockets:

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection $connection_upgrade;
```

For more information, see NGINX as a WebSocket Proxy.

Third-party SignalR backplane providers

- NCache
- Orleans

Next steps

For more information, see the following resources:

- Azure SignalR Service documentation
- Set up a Redis backplane

Publish an ASP.NET Core SignalR app to Azure App Service

9/22/2020 • 2 minutes to read • Edit Online

By Brady Gaster

Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core.

> **NOTE**
> This article refers to publishing an ASP.NET Core SignalR app from Visual Studio. For more information, see SignalR service for Azure.

Publish the app

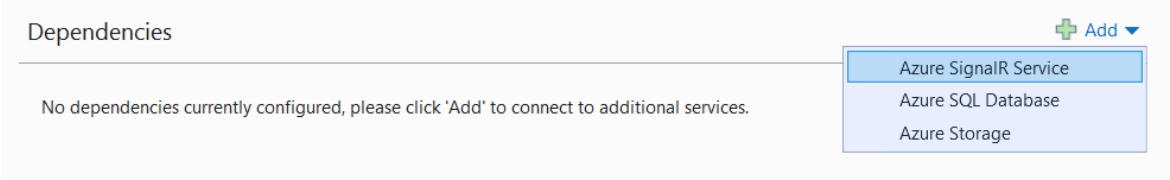
This article covers publishing using the tools in Visual Studio. Visual Studio Code users can use Azure CLI commands to publish apps to Azure. For more information, see Publish an ASP.NET Core app to Azure with command line tools.

1. Right-click on the project in **Solution Explorer** and select **Publish**.
2. Confirm that **App Service** and **Create new** are selected in the **Pick a publish target** dialog.
3. Select **Create Profile** from the **Publish** button drop down.

   Enter the information described in the following table in the **Create App Service** dialog and select **Create**.

ITEM	DESCRIPTION
Name	Unique name of the app.
Subscription	Azure subscription that the app uses.
Resource Group	Group of related resources to which the app belongs.
Hosting Plan	Pricing plan for the web app.

4. Select the **Azure SignalR Service** in the **Dependencies** > **Add** drop-down list:



5. In the **Azure SignalR Service** dialog, select **Create a new Azure SignalR Service instance**.
6. Provide a **Name**, **Resource Group**, and **Location**. Return to the **Azure SignalR Service** dialog and select **Add**.

Visual Studio completes the following tasks:

- Creates a Publish Profile containing publish settings.
- Creates an *Azure Web App* with the provided details.
- Publishes the app.
- Launches a browser, which loads the web app.

The format of the app's URL is `{APP SERVICE NAME}.azurewebsites.net`. For example, an app named `SignalRChatApp` has a URL of `https://signalrchatapp.azurewebsites.net`.

If an HTTP *502.2 - Bad Gateway* error occurs when deploying an app that targets a preview .NET Core release, see Deploy ASP.NET Core preview release to Azure App Service to resolve it.

Configure the app in Azure App Service

> **NOTE**
>
> *This section only applies to apps not using the Azure SignalR Service.*
>
> If the app uses the Azure SignalR Service, the App Service doesn't require the configuration of Application Request Routing (ARR) Affinity and Web Sockets described in this section. Clients connect their Web Sockets to the Azure SignalR Service, not directly to the app.

For apps hosted without the Azure SignalR Service, enable:

- ARR Affinity to route requests from a user back to the same App Service instance. The default setting is **On**.
- Web Sockets to allow the Web Sockets transport to function. The default setting is **Off**.

1. In the Azure portal, navigate to the web app in **App Services**.
2. Open **Configuration** > **General settings**.
3. Set **Web sockets** to **On**.
4. Verify that **ARR affinity** is set to **On**.

App Service Plan limits

Web Sockets and other transports are limited based on the App Service Plan selected. For more information, see the *Azure Cloud Services limits* and *App Service limits* sections of the Azure subscription and service limits, quotas, and constraints article.

Additional resources

- What is Azure SignalR Service?
- Introduction to ASP.NET Core SignalR
- Host and deploy ASP.NET Core
- Publish an ASP.NET Core app to Azure with Visual Studio
- Publish an ASP.NET Core app to Azure with command line tools
- Host and deploy ASP.NET Core Preview apps on Azure

Set up a Redis backplane for ASP.NET Core SignalR scale-out

9/22/2020 • 4 minutes to read • Edit Online

By Andrew Stanton-Nurse, Brady Gaster, and Tom Dykstra,

This article explains SignalR-specific aspects of setting up a Redis server to use for scaling out an ASP.NET Core SignalR app.

Set up a Redis backplane

- Deploy a Redis server.

  > **IMPORTANT**
  > For production use, a Redis backplane is recommended only when it runs in the same data center as the SignalR app. Otherwise, network latency degrades performance. If your SignalR app is running in the Azure cloud, we recommend Azure SignalR Service instead of a Redis backplane. You can use the Azure Redis Cache Service for development and test environments.

  For more information, see the following resources:

  - ASP.NET Core SignalR production hosting and scaling
  - Redis documentation
  - Azure Redis Cache documentation

- In the SignalR app, install the `Microsoft.AspNetCore.SignalR.Redis` NuGet package.

- In the `Startup.ConfigureServices` method, call `AddRedis` after `AddSignalR`:

  ```
  services.AddSignalR().AddRedis("<your_Redis_connection_string>");
  ```

- Configure options as needed:

  Most options can be set in the connection string or in the ConfigurationOptions object. Options specified in `ConfigurationOptions` override the ones set in the connection string.

  The following example shows how to set options in the `ConfigurationOptions` object. This example adds a channel prefix so that multiple apps can share the same Redis instance, as explained in the following step.

  ```
  services.AddSignalR()
    .AddRedis(connectionString, options => {
        options.Configuration.ChannelPrefix = "MyApp";
    });
  ```

  In the preceding code, `options.Configuration` is initialized with whatever was specified in the connection string.

- In the SignalR app, install one of the following NuGet packages:

  - `Microsoft.AspNetCore.SignalR.StackExchangeRedis` - Depends on StackExchange.Redis 2.X.X. This is the

recommended package for ASP.NET Core 2.2 and later.

  - `Microsoft.AspNetCore.SignalR.Redis` - Depends on StackExchange.Redis 1.X.X. This package isn't included in ASP.NET Core 3.0 and later.

- In the `Startup.ConfigureServices` method, call AddStackExchangeRedis:

```
services.AddSignalR().AddStackExchangeRedis("<your_Redis_connection_string>");
```

When using `Microsoft.AspNetCore.SignalR.Redis`, call AddRedis.

- Configure options as needed:

  Most options can be set in the connection string or in the ConfigurationOptions object. Options specified in `ConfigurationOptions` override the ones set in the connection string.

  The following example shows how to set options in the `ConfigurationOptions` object. This example adds a channel prefix so that multiple apps can share the same Redis instance, as explained in the following step.

```
services.AddSignalR()
  .AddStackExchangeRedis(connectionString, options => {
      options.Configuration.ChannelPrefix = "MyApp";
  });
```

When using `Microsoft.AspNetCore.SignalR.Redis`, call AddRedis.

In the preceding code, `options.Configuration` is initialized with whatever was specified in the connection string.

For information about Redis options, see the StackExchange Redis documentation.

- In the SignalR app, install the following NuGet package:

  - `Microsoft.AspNetCore.SignalR.StackExchangeRedis`

- In the `Startup.ConfigureServices` method, call AddStackExchangeRedis:

```
services.AddSignalR().AddStackExchangeRedis("<your_Redis_connection_string>");
```

- Configure options as needed:

  Most options can be set in the connection string or in the ConfigurationOptions object. Options specified in `ConfigurationOptions` override the ones set in the connection string.

  The following example shows how to set options in the `ConfigurationOptions` object. This example adds a channel prefix so that multiple apps can share the same Redis instance, as explained in the following step.

```
services.AddSignalR()
  .AddStackExchangeRedis(connectionString, options => {
      options.Configuration.ChannelPrefix = "MyApp";
  });
```

  In the preceding code, `options.Configuration` is initialized with whatever was specified in the connection string.

  For information about Redis options, see the StackExchange Redis documentation.

- If you're using one Redis server for multiple SignalR apps, use a different channel prefix for each SignalR app.

Setting a channel prefix isolates one SignalR app from others that use different channel prefixes. If you don't assign different prefixes, a message sent from one app to all of its own clients will go to all clients of all apps that use the Redis server as a backplane.

- Configure your server farm load balancing software for sticky sessions. Here are some examples of documentation on how to do that:
  - IIS
  - HAProxy
  - Nginx
  - pfSense

Redis server errors

When a Redis server goes down, SignalR throws exceptions that indicate messages won't be delivered. Some typical exception messages:

- *Failed writing message*
- *Failed to invoke hub method 'MethodName'*
- *Connection to Redis failed*

SignalR doesn't buffer messages to send them when the server comes back up. Any messages sent while the Redis server is down are lost.

SignalR automatically reconnects when the Redis server is available again.

Custom behavior for connection failures

Here's an example that shows how to handle Redis connection failure events.

```
services.AddSignalR()
        .AddRedis(o =>
        {
            o.ConnectionFactory = async writer =>
            {
                var config = new ConfigurationOptions
                {
                    AbortOnConnectFail = false
                };
                config.EndPoints.Add(IPAddress.Loopback, 0);
                config.SetDefaultPorts();
                var connection = await ConnectionMultiplexer.ConnectAsync(config, writer);
                connection.ConnectionFailed += (_, e) =>
                {
                    Console.WriteLine("Connection to Redis failed.");
                };

                if (!connection.IsConnected)
                {
                    Console.WriteLine("Did not connect to Redis.");
                }

                return connection;
            };
        });
```

```
services.AddSignalR()
        .AddMessagePackProtocol()
        .AddStackExchangeRedis(o =>
        {
            o.ConnectionFactory = async writer =>
            {
                var config = new ConfigurationOptions
                {
                    AbortOnConnectFail = false
                };
                config.EndPoints.Add(IPAddress.Loopback, 0);
                config.SetDefaultPorts();
                var connection = await ConnectionMultiplexer.ConnectAsync(config, writer);
                connection.ConnectionFailed += (_, e) =>
                {
                    Console.WriteLine("Connection to Redis failed.");
                };

                if (!connection.IsConnected)
                {
                    Console.WriteLine("Did not connect to Redis.");
                }

                return connection;
            };
        });
```

Redis Clustering

Redis Clustering is a method for achieving high availability by using multiple Redis servers. Clustering isn't officially supported, but it might work.

Next steps

For more information, see the following resources:

- ASP.NET Core SignalR production hosting and scaling
- Redis documentation
- StackExchange Redis documentation
- Azure Redis Cache documentation

Host ASP.NET Core SignalR in background services

9/22/2020 • 4 minutes to read • Edit Online

By Brady Gaster

This article provides guidance for:

- Hosting SignalR Hubs using a background worker process hosted with ASP.NET Core.
- Sending messages to connected clients from within a .NET Core BackgroundService.

View or download sample code (how to download)

View or download sample code (how to download)

Enable SignalR in startup

Hosting ASP.NET Core SignalR Hubs in the context of a background worker process is identical to hosting a Hub in an ASP.NET Core web app. In the `Startup.ConfigureServices` method, calling `services.AddSignalR` adds the required services to the ASP.NET Core Dependency Injection (DI) layer to support SignalR. In `Startup.Configure`, the `MapHub` method is called in the `UseEndpoints` callback to connect the Hub endpoints in the ASP.NET Core request pipeline.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSignalR();
        services.AddHostedService<Worker>();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapHub<ClockHub>("/hubs/clock");
        });
    }
}
```

Hosting ASP.NET Core SignalR Hubs in the context of a background worker process is identical to hosting a Hub in an ASP.NET Core web app. In the `Startup.ConfigureServices` method, calling `services.AddSignalR` adds the required services to the ASP.NET Core Dependency Injection (DI) layer to support SignalR. In `Startup.Configure`, the `UseSignalR` method is called to connect the Hub endpoint(s) in the ASP.NET Core request pipeline.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSignalR();
        services.AddHostedService<Worker>();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseSignalR((routes) =>
        {
            routes.MapHub<ClockHub>("/hubs/clock");
        });
    }
}
```

In the preceding example, the `ClockHub` class implements the `Hub<T>` class to create a strongly typed Hub. The `ClockHub` has been configured in the `Startup` class to respond to requests at the endpoint `/hubs/clock`.

For more information on strongly typed Hubs, see Use hubs in SignalR for ASP.NET Core.

> **NOTE**
> This functionality isn't limited to the Hub<T> class. Any class that inherits from Hub, such as DynamicHub, works.

```
public class ClockHub : Hub<IClock>
{
    public async Task SendTimeToClients(DateTime dateTime)
    {
        await Clients.All.ShowTime(dateTime);
    }
}
```

```
public class ClockHub : Hub<IClock>
{
    public async Task SendTimeToClients(DateTime dateTime)
    {
        await Clients.All.ShowTime(dateTime);
    }
}
```

The interface used by the strongly typed `ClockHub` is the `IClock` interface.

```
public interface IClock
{
    Task ShowTime(DateTime currentTime);
}
```

```
public interface IClock
{
    Task ShowTime(DateTime currentTime);
}
```

Call a SignalR Hub from a background service

During startup, the `Worker` class, a `BackgroundService`, is enabled using `AddHostedService`.

```
services.AddHostedService<Worker>();
```

Since SignalR is also enabled up during the `Startup` phase, in which each Hub is attached to an individual endpoint in ASP.NET Core's HTTP request pipeline, each Hub is represented by an `IHubContext<T>` on the server. Using ASP.NET Core's DI features, other classes instantiated by the hosting layer, like `BackgroundService` classes, MVC Controller classes, or Razor page models, can get references to server-side Hubs by accepting instances of `IHubContext<ClockHub, IClock>` during construction.

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    private readonly IHubContext<ClockHub, IClock> _clockHub;

    public Worker(ILogger<Worker> logger, IHubContext<ClockHub, IClock> clockHub)
    {
        _logger = logger;
        _clockHub = clockHub;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {Time}", DateTime.Now);
            await _clockHub.Clients.All.ShowTime(DateTime.Now);
            await Task.Delay(1000);
        }
    }
}
```

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    private readonly IHubContext<ClockHub, IClock> _clockHub;

    public Worker(ILogger<Worker> logger, IHubContext<ClockHub, IClock> clockHub)
    {
        _logger = logger;
        _clockHub = clockHub;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {Time}", DateTime.Now);
            await _clockHub.Clients.All.ShowTime(DateTime.Now);
            await Task.Delay(1000);
        }
    }
}
```

As the `ExecuteAsync` method is called iteratively in the background service, the server's current date and time are sent to the connected clients using the `ClockHub`.

React to SignalR events with background services

Like a Single Page App using the JavaScript client for SignalR or a .NET desktop app can do using the using the ASP.NET Core SignalR .NET Client, a `BackgroundService` or `IHostedService` implementation can also be used to connect to SignalR Hubs and respond to events.

The `ClockHubClient` class implements both the `IClock` interface and the `IHostedService` interface. This way it can be enabled during `Startup` to run continuously and respond to Hub events from the server.

```
public partial class ClockHubClient : IClock, IHostedService
{
}
```

During initialization, the `ClockHubClient` creates an instance of a `HubConnection` and enables the `IClock.ShowTime` method as the handler for the Hub's `ShowTime` event.

```
private readonly ILogger<ClockHubClient> _logger;
private HubConnection _connection;

public ClockHubClient(ILogger<ClockHubClient> logger)
{
    _logger = logger;

    _connection = new HubConnectionBuilder()
        .WithUrl(Strings.HubUrl)
        .Build();

    _connection.On<DateTime>(Strings.Events.TimeSent, ShowTime);
}

public Task ShowTime(DateTime currentTime)
{
    _logger.LogInformation("{CurrentTime}", currentTime.ToShortTimeString());

    return Task.CompletedTask;
}
```

In the `IHostedService.StartAsync` implementation, the `HubConnection` is started asynchronously.

```
public async Task StartAsync(CancellationToken cancellationToken)
{
    // Loop is here to wait until the server is running
    while (true)
    {
        try
        {
            await _connection.StartAsync(cancellationToken);

            break;
        }
        catch
        {
            await Task.Delay(1000);
        }
    }
}
```

During the `IHostedService.StopAsync` method, the `HubConnection` is disposed of asynchronously.

```
public Task StopAsync(CancellationToken cancellationToken)
{
    return _connection.DisposeAsync();
}
```

```
private readonly ILogger<ClockHubClient> _logger;
private HubConnection _connection;

public ClockHubClient(ILogger<ClockHubClient> logger)
{
    _logger = logger;

    _connection = new HubConnectionBuilder()
        .WithUrl(Strings.HubUrl)
        .Build();

    _connection.On<DateTime>(Strings.Events.TimeSent,
        dateTime => _ = ShowTime(dateTime));
}

public Task ShowTime(DateTime currentTime)
{
    _logger.LogInformation("{CurrentTime}", currentTime.ToShortTimeString());

    return Task.CompletedTask;
}
```

In the `IHostedService.StartAsync` implementation, the `HubConnection` is started asynchronously.

```
public async Task StartAsync(CancellationToken cancellationToken)
{
    // Loop is here to wait until the server is running
    while (true)
    {
        try
        {
            await _connection.StartAsync(cancellationToken);

            break;
        }
        catch
        {
            await Task.Delay(1000);
        }
    }
}
```

During the `IHostedService.StopAsync` method, the `HubConnection` is disposed of asynchronously.

```
public Task StopAsync(CancellationToken cancellationToken)
{
    return _connection.DisposeAsync();
}
```

Additional resources

- Get started
- Hubs
- Publish to Azure
- Strongly typed Hubs

ASP.NET Core SignalR configuration

9/22/2020 • 69 minutes to read • Edit Online

JSON/MessagePack serialization options

ASP.NET Core SignalR supports two protocols for encoding messages: JSON and MessagePack. Each protocol has serialization configuration options.

JSON serialization can be configured on the server using the AddJsonProtocol extension method. `AddJsonProtocol` can be added after AddSignalR in `Startup.ConfigureServices`. The `AddJsonProtocol` method takes a delegate that receives an `options` object. The PayloadSerializerOptions property on that object is a `System.Text.Json` JsonSerializerOptions object that can be used to configure serialization of arguments and return values. For more information, see the System.Text.Json documentation.

As an example, to configure the serializer to not change the casing of property names, instead of the default "camelCase" names, use the following code in `Startup.ConfigureServices`:

```
services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null
    });
```

In the .NET client, the same `AddJsonProtocol` extension method exists on HubConnectionBuilder. The `Microsoft.Extensions.DependencyInjection` namespace must be imported to resolve the extension method:

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
    })
    .Build();
```

> **NOTE**
> It's not possible to configure JSON serialization in the JavaScript client at this time.

Switch to Newtonsoft.Json

If you need features of `Newtonsoft.Json` that aren't supported in `System.Text.Json`, See Switch to NewtonsoftJson.

MessagePack serialization options

MessagePack serialization can be configured by providing a delegate to the AddMessagePackProtocol call. See MessagePack in SignalR for more details.

> **NOTE**
> It's not possible to configure MessagePack serialization in the JavaScript client at this time.

Configure server options

The following table describes options for configuring SignalR hubs:

OPTION	DEFAULT VALUE	DESCRIPTION
`ClientTimeoutInterval`	30 seconds	The server will consider the client disconnected if it hasn't received a message (including keep-alive) in this interval. It could take longer than this timeout interval for the client to actually be marked disconnected, due to how this is implemented. The recommended value is double the `KeepAliveInterval` value.
`HandshakeTimeout`	15 seconds	If the client doesn't send an initial handshake message within this time interval, the connection is closed. This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	If the server hasn't sent a message within this interval, a ping message is sent automatically to keep the connection open. When changing `KeepAliveInterval`, change the `ServerTimeout` / `serverTimeoutInMilliseconds` setting on the client. The recommended `ServerTimeout` / `serverTimeoutInMilliseconds` value is double the `KeepAliveInterval` value.
`SupportedProtocols`	All installed protocols	Protocols supported by this hub. By default, all protocols registered on the server are allowed, but protocols can be removed from this list to disable specific protocols for individual hubs.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a Hub method. The default is `false`, as these exception messages can contain sensitive information.
`StreamBufferCapacity`	`10`	The maximum number of items that can be buffered for client upload streams. If this limit is reached, the processing of invocations is blocked until the server processes stream items.

OPTION	DEFAULT VALUE	DESCRIPTION
`MaximumReceiveMessageSize`	32 KB	Maximum size of a single incoming hub message.

Options can be configured for all hubs by providing an options delegate to the `AddSignalR` call in `Startup.ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.EnableDetailedErrors = true;
        hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
    });
}
```

Options for a single hub override the global options provided in `AddSignalR` and can be configured using AddHubOptions:

```
services.AddSignalR().AddHubOptions<ChatHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

Advanced HTTP configuration options

Use `HttpConnectionDispatcherOptions` to configure advanced settings related to transports and memory buffer management. These options are configured by passing a delegate to MapHub<T> in `Startup.Configure`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chathub", options =>
        {
            options.Transports =
                HttpTransportType.WebSockets |
                HttpTransportType.LongPolling;
        });
    });
}
```

The following table describes options for configuring ASP.NET Core SignalR's advanced HTTP options:

OPTION	DEFAULT VALUE	DESCRIPTION
`ApplicationMaxBufferSize`	32 KB	The maximum number of bytes received from the client that the server buffers before applying backpressure. Increasing this value allows the server to receive larger messages more quickly without applying backpressure, but can increase memory consumption.

OPTION	DEFAULT VALUE	DESCRIPTION
`AuthorizationData`	Data automatically gathered from the `Authorize` attributes applied to the Hub class.	A list of IAuthorizeData objects used to determine if a client is authorized to connect to the hub.
`TransportMaxBufferSize`	32 KB	The maximum number of bytes sent by the app that the server buffers before observing backpressure. Increasing this value allows the server to buffer larger messages more quickly without awaiting backpressure, but can increase memory consumption.
`Transports`	All Transports are enabled.	A bit flags enum of `HttpTransportType` values that can restrict the transports a client can use to connect.
`LongPolling`	See below.	Additional options specific to the Long Polling transport.
`WebSockets`	See below.	Additional options specific to the WebSockets transport.
`MinimumProtocolVersion`	0	Specify the minimum version of the negotiate protocol. This is used to limit clients to newer versions.

The Long Polling transport has additional options that can be configured using the `LongPolling` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`PollTimeout`	90 seconds	The maximum amount of time the server waits for a message to send to the client before terminating a single poll request. Decreasing this value causes the client to issue new poll requests more frequently.

The WebSocket transport has additional options that can be configured using the `WebSockets` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`CloseTimeout`	5 seconds	After the server closes, if the client fails to close within this time interval, the connection is terminated.
`SubProtocolSelector`	`null`	A delegate that can be used to set the `Sec-WebSocket-Protocol` header to a custom value. The delegate receives the values requested by the client as input and is expected to return the desired value.

Configure client options

Client options can be configured on the `HubConnectionBuilder` type (available in the .NET and JavaScript clients). It's also available in the Java client, but the `HttpHubConnectionBuilder` subclass is what contains the builder configuration options, as well as on the `HubConnection` itself.

Configure logging

Logging is configured in the .NET Client using the `ConfigureLogging` method. Logging providers and filters can be registered in the same way as they are on the server. See the Logging in ASP.NET Core documentation for more information.

> **NOTE**
> In order to register Logging providers, you must install the necessary packages. See the Built-in logging providers section of the docs for a full list.

For example, to enable Console logging, install the `Microsoft.Extensions.Logging.Console` NuGet package. Call the `AddConsole` extension method:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

In the JavaScript client, a similar `configureLogging` method exists. Provide a `LogLevel` value indicating the minimum level of log messages to produce. Logs are written to the browser console window.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Instead of a `LogLevel` value, you can also provide a `string` value representing a log level name. This is useful when configuring SignalR logging in environments where you don't have access to the `LogLevel` constants.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging("warn")
    .build();
```

The following table lists the available log levels. The value you provide to `configureLogging` sets the **minimum** log level that will be logged. Messages logged at this level, **or the levels listed after it in the table**, will be logged.

STRING	LOGLEVEL
`trace`	`LogLevel.Trace`
`debug`	`LogLevel.Debug`
`info` **or** `information`	`LogLevel.Information`
`warn` **or** `warning`	`LogLevel.Warning`

STRING	LOGLEVEL
`error`	`LogLevel.Error`
`critical`	`LogLevel.Critical`
`none`	`LogLevel.None`

> **NOTE**
> To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

For more information on logging, see the SignalR Diagnostics documentation.

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to chose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

This can safely be ignored.

Configure allowed transports

The transports used by SignalR can be configured in the `WithUrl` call (`withUrl` in JavaScript). A bitwise-OR of the values of `HttpTransportType` can be used to restrict the client to only use the specified transports. All transports are enabled by default.

For example, to disable the Server-Sent Events transport, but allow WebSockets and Long Polling connections:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", HttpTransportType.WebSockets | HttpTransportType.LongPolling)
    .Build();
```

In the JavaScript client, transports are configured by setting the `transport` field on the options object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

In this version of the Java client websockets is the only available transport.

In the Java client, the transport is selected with the `withTransport` method on the `HttpHubConnectionBuilder`. The Java client defaults to using the WebSockets transport.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withTransport(TransportEnum.WEBSOCKETS)
    .build();
```

> **NOTE**
> The SignalR Java client doesn't support transport fallback yet.

Configure bearer authentication

To provide authentication data along with SignalR requests, use the `AccessTokenProvider` option (`accessTokenFactory` in JavaScript) to specify a function that returns the desired access token. In the .NET Client, this access token is passed in as an HTTP "Bearer Authentication" token (Using the `Authorization` header with a type of `Bearer`). In the JavaScript client, the access token is used as a Bearer token, **except** in a few cases where browser APIs restrict the ability to apply headers (specifically, in Server-Sent Events and WebSockets requests). In these cases, the access token is provided as a query string value `access_token`.

In the .NET client, the `AccessTokenProvider` option can be specified using the options delegate in `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

In the JavaScript client, the access token is configured by setting the `accessTokenFactory` field on the options object in `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an access token factory to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Configure timeout and keep-alive options

Additional options for configuring timeout and keep-alive behavior are available on the `HubConnection` object itself:

- .NET
- JavaScript

- Java

OPTION	DEFAULT VALUE	DESCRIPTION
`ServerTimeout`	30 seconds (30,000 milliseconds)	Timeout for server activity. If the server hasn't sent a message in this interval, the client considers the server disconnected and triggers the `Closed` event (`onclose` in JavaScript). This value must be large enough for a ping message to be sent from the server **and** received by the client within the timeout interval. The recommended value is a number at least double the server's `KeepAliveInterval` value to allow time for pings to arrive.
`HandshakeTimeout`	15 seconds	Timeout for initial server handshake. If the server doesn't send a handshake response in this interval, the client cancels the handshake and triggers the `Closed` event (`onclose` in JavaScript). This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	Determines the interval at which the client sends ping messages. Sending any message from the client resets the timer to the start of the interval. If the client hasn't sent a message in the `ClientTimeoutInterval` set on the server, the server considers the client disconnected.

In the .NET Client, timeout values are specified as `TimeSpan` values.

Configure additional options

Additional options can be configured in the `WithUrl` (`withUrl` in JavaScript) method on `HubConnectionBuilder` or on the various configuration APIs on the `HttpHubConnectionBuilder` in the Java client:

- .NET
- JavaScript
- Java

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`AccessTokenProvider`	`null`	A function returning a string that is provided as a Bearer authentication token in HTTP requests.

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`SkipNegotiation`	`false`	Set this to `true` to skip the negotiation step. **Only supported when the WebSockets transport is the only enabled transport**. This setting can't be enabled when using the Azure SignalR Service.
`ClientCertificates`	Empty	A collection of TLS certificates to send to authenticate requests.
`Cookies`	Empty	A collection of HTTP cookies to send with every HTTP request.
`Credentials`	Empty	Credentials to send with every HTTP request.
`CloseTimeout`	5 seconds	WebSockets only. The maximum amount of time the client waits after closing for the server to acknowledge the close request. If the server doesn't acknowledge the close within this time, the client disconnects.
`Headers`	Empty	A Map of additional HTTP headers to send with every HTTP request.
`HttpMessageHandlerFactory`	`null`	A delegate that can be used to configure or replace the `HttpMessageHandler` used to send HTTP requests. Not used for WebSocket connections. This delegate must return a non-null value, and it receives the default value as a parameter. Either modify settings on that default value and return it, or return a new `HttpMessageHandler` instance. **When replacing the handler make sure to copy the settings you want to keep from the provided handler, otherwise, the configured options (such as Cookies and Headers) won't apply to the new handler.**
`Proxy`	`null`	An HTTP proxy to use when sending HTTP requests.
`UseDefaultCredentials`	`false`	Set this boolean to send the default credentials for HTTP and WebSockets requests. This enables the use of Windows authentication.
`WebSocketConfiguration`	`null`	A delegate that can be used to configure additional WebSocket options. Receives an instance of ClientWebSocketOptions that can be used to configure the options.

In the .NET Client, these options can be modified by the options delegate provided to `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.Headers["Foo"] = "Bar";
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

In the JavaScript Client, these options can be provided in a JavaScript object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        skipNegotiation: true,
        transport: signalR.HttpTransportType.WebSockets
    })
    .build();
```

In the Java client, these options can be configured with the methods on the `HttpHubConnectionBuilder` returned from the `HubConnectionBuilder.create("HUB URL")`

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
        .withHeader("Foo", "Bar")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .build();
```

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client
- Use MessagePack Hub Protocol in SignalR for ASP.NET Core
- ASP.NET Core SignalR supported platforms

JSON/MessagePack serialization options

ASP.NET Core SignalR supports two protocols for encoding messages: JSON and MessagePack. Each protocol has serialization configuration options.

JSON serialization can be configured on the server using the AddJsonProtocol extension method. `AddJsonProtocol` can be added after AddSignalR in `Startup.ConfigureServices`. The `AddJsonProtocol` method takes a delegate that receives an `options` object. The PayloadSerializerOptions property on that object is a `System.Text.Json` JsonSerializerOptions object that can be used to configure serialization of arguments and return values. For more information, see the System.Text.Json documentation.

As an example, to configure the serializer to not change the casing of property names, instead of the default "camelCase" names, use the following code in `Startup.ConfigureServices`:

```
services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null
    });
```

In the .NET client, the same `AddJsonProtocol` extension method exists on HubConnectionBuilder. The `Microsoft.Extensions.DependencyInjection` namespace must be imported to resolve the extension method:

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
    })
    .Build();
```

> **NOTE**
> It's not possible to configure JSON serialization in the JavaScript client at this time.

Switch to Newtonsoft.Json

If you need features of `Newtonsoft.Json` that aren't supported in `System.Text.Json`, See Switch to NewtonsoftJson.

MessagePack serialization options

MessagePack serialization can be configured by providing a delegate to the AddMessagePackProtocol call. See MessagePack in SignalR for more details.

> **NOTE**
> It's not possible to configure MessagePack serialization in the JavaScript client at this time.

Configure server options

The following table describes options for configuring SignalR hubs:

OPTION	DEFAULT VALUE	DESCRIPTION
`ClientTimeoutInterval`	30 seconds	The server will consider the client disconnected if it hasn't received a message (including keep-alive) in this interval. It could take longer than this timeout interval for the client to actually be marked disconnected, due to how this is implemented. The recommended value is double the `KeepAliveInterval` value.

OPTION	DEFAULT VALUE	DESCRIPTION
`HandshakeTimeout`	15 seconds	If the client doesn't send an initial handshake message within this time interval, the connection is closed. This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	If the server hasn't sent a message within this interval, a ping message is sent automatically to keep the connection open. When changing `KeepAliveInterval`, change the `ServerTimeout` / `serverTimeoutInMilliseconds` setting on the client. The recommended `ServerTimeout` / `serverTimeoutInMilliseconds` value is double the `KeepAliveInterval` value.
`SupportedProtocols`	All installed protocols	Protocols supported by this hub. By default, all protocols registered on the server are allowed, but protocols can be removed from this list to disable specific protocols for individual hubs.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a Hub method. The default is `false`, as these exception messages can contain sensitive information.
`StreamBufferCapacity`	`10`	The maximum number of items that can be buffered for client upload streams. If this limit is reached, the processing of invocations is blocked until the server processes stream items.
`MaximumReceiveMessageSize`	32 KB	Maximum size of a single incoming hub message.

Options can be configured for all hubs by providing an options delegate to the `AddSignalR` call in `Startup.ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.EnableDetailedErrors = true;
        hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
    });
}
```

Options for a single hub override the global options provided in `AddSignalR` and can be configured using AddHubOptions:

```
services.AddSignalR().AddHubOptions<ChatHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

Advanced HTTP configuration options

Use `HttpConnectionDispatcherOptions` to configure advanced settings related to transports and memory buffer management. These options are configured by passing a delegate to MapHub<T> in `Startup.Configure`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chathub", options =>
        {
            options.Transports =
                HttpTransportType.WebSockets |
                HttpTransportType.LongPolling;
        });
    });
}
```

The following table describes options for configuring ASP.NET Core SignalR's advanced HTTP options:

OPTION	DEFAULT VALUE	DESCRIPTION
`ApplicationMaxBufferSize`	32 KB	The maximum number of bytes received from the client that the server buffers before applying backpressure. Increasing this value allows the server to receive larger messages more quickly without applying backpressure, but can increase memory consumption.
`AuthorizationData`	Data automatically gathered from the `Authorize` attributes applied to the Hub class.	A list of IAuthorizeData objects used to determine if a client is authorized to connect to the hub.
`TransportMaxBufferSize`	32 KB	The maximum number of bytes sent by the app that the server buffers before observing backpressure. Increasing this value allows the server to buffer larger messages more quickly without awaiting backpressure, but can increase memory consumption.
`Transports`	All Transports are enabled.	A bit flags enum of `HttpTransportType` values that can restrict the transports a client can use to connect.
`LongPolling`	See below.	Additional options specific to the Long Polling transport.

OPTION	DEFAULT VALUE	DESCRIPTION
`WebSockets`	See below.	Additional options specific to the WebSockets transport.
`MinimumProtocolVersion`	0	Specify the minimum version of the negotiate protocol. This is used to limit clients to newer versions.

The Long Polling transport has additional options that can be configured using the `LongPolling` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`PollTimeout`	90 seconds	The maximum amount of time the server waits for a message to send to the client before terminating a single poll request. Decreasing this value causes the client to issue new poll requests more frequently.

The WebSocket transport has additional options that can be configured using the `WebSockets` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`CloseTimeout`	5 seconds	After the server closes, if the client fails to close within this time interval, the connection is terminated.
`SubProtocolSelector`	`null`	A delegate that can be used to set the `Sec-WebSocket-Protocol` header to a custom value. The delegate receives the values requested by the client as input and is expected to return the desired value.

Configure client options

Client options can be configured on the `HubConnectionBuilder` type (available in the .NET and JavaScript clients). It's also available in the Java client, but the `HttpHubConnectionBuilder` subclass is what contains the builder configuration options, as well as on the `HubConnection` itself.

Configure logging

Logging is configured in the .NET Client using the `ConfigureLogging` method. Logging providers and filters can be registered in the same way as they are on the server. See the Logging in ASP.NET Core documentation for more information.

> **NOTE**
> In order to register Logging providers, you must install the necessary packages. See the Built-in logging providers section of the docs for a full list.

For example, to enable Console logging, install the `Microsoft.Extensions.Logging.Console` NuGet package. Call the `AddConsole` extension method:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

In the JavaScript client, a similar `configureLogging` method exists. Provide a `LogLevel` value indicating the minimum level of log messages to produce. Logs are written to the browser console window.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Instead of a `LogLevel` value, you can also provide a `string` value representing a log level name. This is useful when configuring SignalR logging in environments where you don't have access to the `LogLevel` constants.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging("warn")
    .build();
```

The following table lists the available log levels. The value you provide to `configureLogging` sets the **minimum** log level that will be logged. Messages logged at this level, **or the levels listed after it in the table**, will be logged.

STRING	LOGLEVEL
`trace`	`LogLevel.Trace`
`debug`	`LogLevel.Debug`
`info` **or** `information`	`LogLevel.Information`
`warn` **or** `warning`	`LogLevel.Warning`
`error`	`LogLevel.Error`
`critical`	`LogLevel.Critical`
`none`	`LogLevel.None`

> **NOTE**
> To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

For more information on logging, see the SignalR Diagnostics documentation.

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to chose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

This can safely be ignored.

Configure allowed transports

The transports used by SignalR can be configured in the `WithUrl` call (`withUrl` in JavaScript). A bitwise-OR of the values of `HttpTransportType` can be used to restrict the client to only use the specified transports. All transports are enabled by default.

For example, to disable the Server-Sent Events transport, but allow WebSockets and Long Polling connections:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", HttpTransportType.WebSockets | HttpTransportType.LongPolling)
    .Build();
```

In the JavaScript client, transports are configured by setting the `transport` field on the options object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

In this version of the Java client websockets is the only available transport.

In the Java client, the transport is selected with the `withTransport` method on the `HttpHubConnectionBuilder`. The Java client defaults to using the WebSockets transport.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withTransport(TransportEnum.WEBSOCKETS)
    .build();
```

> **NOTE**
> The SignalR Java client doesn't support transport fallback yet.

Configure bearer authentication

To provide authentication data along with SignalR requests, use the `AccessTokenProvider` option (`accessTokenFactory` in JavaScript) to specify a function that returns the desired access token. In the .NET Client, this access token is passed in as an HTTP "Bearer Authentication" token (Using the `Authorization` header with a type of `Bearer`). In the JavaScript client, the access token is used as a Bearer token, **except** in a few cases where browser APIs restrict the ability to apply headers (specifically, in Server-Sent Events and WebSockets requests). In these cases, the access token is provided as a query string value `access_token`.

In the .NET client, the `AccessTokenProvider` option can be specified using the options delegate in `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

In the JavaScript client, the access token is configured by setting the `accessTokenFactory` field on the options object in `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an access token factory to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Configure timeout and keep-alive options

Additional options for configuring timeout and keep-alive behavior are available on the `HubConnection` object itself:

- .NET
- JavaScript
- Java

OPTION	DEFAULT VALUE	DESCRIPTION
`ServerTimeout`	30 seconds (30,000 milliseconds)	Timeout for server activity. If the server hasn't sent a message in this interval, the client considers the server disconnected and triggers the `Closed` event (`onclose` in JavaScript). This value must be large enough for a ping message to be sent from the server **and** received by the client within the timeout interval. The recommended value is a number at least double the server's `KeepAliveInterval` value to allow time for pings to arrive.

OPTION	DEFAULT VALUE	DESCRIPTION
`HandshakeTimeout`	15 seconds	Timeout for initial server handshake. If the server doesn't send a handshake response in this interval, the client cancels the handshake and triggers the `Closed` event (`onclose` in JavaScript). This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	Determines the interval at which the client sends ping messages. Sending any message from the client resets the timer to the start of the interval. If the client hasn't sent a message in the `ClientTimeoutInterval` set on the server, the server considers the client disconnected.

In the .NET Client, timeout values are specified as `TimeSpan` values.

Configure additional options

Additional options can be configured in the `WithUrl` (`withUrl` in JavaScript) method on `HubConnectionBuilder` or on the various configuration APIs on the `HttpHubConnectionBuilder` in the Java client:

- .NET
- JavaScript
- Java

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`AccessTokenProvider`	`null`	A function returning a string that is provided as a Bearer authentication token in HTTP requests.
`SkipNegotiation`	`false`	Set this to `true` to skip the negotiation step. **Only supported when the WebSockets transport is the only enabled transport**. This setting can't be enabled when using the Azure SignalR Service.
`ClientCertificates`	Empty	A collection of TLS certificates to send to authenticate requests.
`Cookies`	Empty	A collection of HTTP cookies to send with every HTTP request.
`Credentials`	Empty	Credentials to send with every HTTP request.

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`CloseTimeout`	5 seconds	WebSockets only. The maximum amount of time the client waits after closing for the server to acknowledge the close request. If the server doesn't acknowledge the close within this time, the client disconnects.
`Headers`	Empty	A Map of additional HTTP headers to send with every HTTP request.
`HttpMessageHandlerFactory`	`null`	A delegate that can be used to configure or replace the `HttpMessageHandler` used to send HTTP requests. Not used for WebSocket connections. This delegate must return a non-null value, and it receives the default value as a parameter. Either modify settings on that default value and return it, or return a new `HttpMessageHandler` instance. **When replacing the handler make sure to copy the settings you want to keep from the provided handler, otherwise, the configured options (such as Cookies and Headers) won't apply to the new handler.**
`Proxy`	`null`	An HTTP proxy to use when sending HTTP requests.
`UseDefaultCredentials`	`false`	Set this boolean to send the default credentials for HTTP and WebSockets requests. This enables the use of Windows authentication.
`WebSocketConfiguration`	`null`	A delegate that can be used to configure additional WebSocket options. Receives an instance of ClientWebSocketOptions that can be used to configure the options.

In the .NET Client, these options can be modified by the options delegate provided to `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.Headers["Foo"] = "Bar";
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

In the JavaScript Client, these options can be provided in a JavaScript object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        skipNegotiation: true,
        transport: signalR.HttpTransportType.WebSockets
    })
    .build();
```

In the Java client, these options can be configured with the methods on the `HttpHubConnectionBuilder` returned from the `HubConnectionBuilder.create("HUB URL")`

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
        .withHeader("Foo", "Bar")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .build();
```

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client
- Use MessagePack Hub Protocol in SignalR for ASP.NET Core
- ASP.NET Core SignalR supported platforms

JSON/MessagePack serialization options

ASP.NET Core SignalR supports two protocols for encoding messages: JSON and MessagePack. Each protocol has serialization configuration options.

JSON serialization can be configured on the server using the AddJsonProtocol extension method. `AddJsonProtocol` can be added after AddSignalR in `Startup.ConfigureServices`. The `AddJsonProtocol` method takes a delegate that receives an `options` object. The PayloadSerializerOptions property on that object is a `System.Text.Json` JsonSerializerOptions object that can be used to configure serialization of arguments and return values. For more information, see the System.Text.Json documentation.

As an example, to configure the serializer to not change the casing of property names, instead of the default "camelCase" names, use the following code in `Startup.ConfigureServices`:

```
services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
    });
```

In the .NET client, the same `AddJsonProtocol` extension method exists on HubConnectionBuilder. The `Microsoft.Extensions.DependencyInjection` namespace must be imported to resolve the extension method:

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
    })
    .Build();
```

> **NOTE**
> It's not possible to configure JSON serialization in the JavaScript client at this time.

Switch to Newtonsoft.Json

If you need features of `Newtonsoft.Json` that aren't supported in `System.Text.Json`, See Switch to NewtonsoftJson.

MessagePack serialization options

MessagePack serialization can be configured by providing a delegate to the AddMessagePackProtocol call. See MessagePack in SignalR for more details.

> **NOTE**
> It's not possible to configure MessagePack serialization in the JavaScript client at this time.

Configure server options

The following table describes options for configuring SignalR hubs:

OPTION	DEFAULT VALUE	DESCRIPTION
`ClientTimeoutInterval`	30 seconds	The server will consider the client disconnected if it hasn't received a message (including keep-alive) in this interval. It could take longer than this timeout interval for the client to actually be marked disconnected, due to how this is implemented. The recommended value is double the `KeepAliveInterval` value.
`HandshakeTimeout`	15 seconds	If the client doesn't send an initial handshake message within this time interval, the connection is closed. This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.

OPTION	DEFAULT VALUE	DESCRIPTION
`KeepAliveInterval`	15 seconds	If the server hasn't sent a message within this interval, a ping message is sent automatically to keep the connection open. When changing `KeepAliveInterval`, change the `ServerTimeout` / `serverTimeoutInMilliseconds` setting on the client. The recommended `ServerTimeout` / `serverTimeoutInMilliseconds` value is double the `KeepAliveInterval` value.
`SupportedProtocols`	All installed protocols	Protocols supported by this hub. By default, all protocols registered on the server are allowed, but protocols can be removed from this list to disable specific protocols for individual hubs.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a Hub method. The default is `false`, as these exception messages can contain sensitive information.
`StreamBufferCapacity`	`10`	The maximum number of items that can be buffered for client upload streams. If this limit is reached, the processing of invocations is blocked until the server processes stream items.
`MaximumReceiveMessageSize`	32 KB	Maximum size of a single incoming hub message.

Options can be configured for all hubs by providing an options delegate to the `AddSignalR` call in `Startup.ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.EnableDetailedErrors = true;
        hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
    });
}
```

Options for a single hub override the global options provided in `AddSignalR` and can be configured using AddHubOptions:

```
services.AddSignalR().AddHubOptions<ChatHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

Advanced HTTP configuration options

Use `HttpConnectionDispatcherOptions` to configure advanced settings related to transports and memory buffer management. These options are configured by passing a delegate to MapHub<T> in `Startup.Configure`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chathub", options =>
        {
            options.Transports =
                HttpTransportType.WebSockets |
                HttpTransportType.LongPolling;
        });
    });
}
```

The following table describes options for configuring ASP.NET Core SignalR's advanced HTTP options:

OPTION	DEFAULT VALUE	DESCRIPTION
`ApplicationMaxBufferSize`	32 KB	The maximum number of bytes received from the client that the server buffers before applying backpressure. Increasing this value allows the server to receive larger messages more quickly without applying backpressure, but can increase memory consumption.
`AuthorizationData`	Data automatically gathered from the `Authorize` attributes applied to the Hub class.	A list of IAuthorizeData objects used to determine if a client is authorized to connect to the hub.
`TransportMaxBufferSize`	32 KB	The maximum number of bytes sent by the app that the server buffers before observing backpressure. Increasing this value allows the server to buffer larger messages more quickly without awaiting backpressure, but can increase memory consumption.
`Transports`	All Transports are enabled.	A bit flags enum of `HttpTransportType` values that can restrict the transports a client can use to connect.
`LongPolling`	See below.	Additional options specific to the Long Polling transport.
`WebSockets`	See below.	Additional options specific to the WebSockets transport.

The Long Polling transport has additional options that can be configured using the `LongPolling` property:

OPTION	DEFAULT VALUE	DESCRIPTION

OPTION	DEFAULT VALUE	DESCRIPTION
`PollTimeout`	90 seconds	The maximum amount of time the server waits for a message to send to the client before terminating a single poll request. Decreasing this value causes the client to issue new poll requests more frequently.

The WebSocket transport has additional options that can be configured using the `WebSockets` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`CloseTimeout`	5 seconds	After the server closes, if the client fails to close within this time interval, the connection is terminated.
`SubProtocolSelector`	`null`	A delegate that can be used to set the `Sec-WebSocket-Protocol` header to a custom value. The delegate receives the values requested by the client as input and is expected to return the desired value.

Configure client options

Client options can be configured on the `HubConnectionBuilder` type (available in the .NET and JavaScript clients). It's also available in the Java client, but the `HttpHubConnectionBuilder` subclass is what contains the builder configuration options, as well as on the `HubConnection` itself.

Configure logging

Logging is configured in the .NET Client using the `ConfigureLogging` method. Logging providers and filters can be registered in the same way as they are on the server. See the Logging in ASP.NET Core documentation for more information.

> **NOTE**
> In order to register Logging providers, you must install the necessary packages. See the Built-in logging providers section of the docs for a full list.

For example, to enable Console logging, install the `Microsoft.Extensions.Logging.Console` NuGet package. Call the `AddConsole` extension method:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

In the JavaScript client, a similar `configureLogging` method exists. Provide a `LogLevel` value indicating the minimum level of log messages to produce. Logs are written to the browser console window.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Instead of a `LogLevel` value, you can also provide a `string` value representing a log level name. This is useful when configuring SignalR logging in environments where you don't have access to the `LogLevel` constants.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging("warn")
    .build();
```

The following table lists the available log levels. The value you provide to `configureLogging` sets the **minimum** log level that will be logged. Messages logged at this level, **or the levels listed after it in the table**, will be logged.

STRING	LOGLEVEL
`trace`	`LogLevel.Trace`
`debug`	`LogLevel.Debug`
`info` **or** `information`	`LogLevel.Information`
`warn` **or** `warning`	`LogLevel.Warning`
`error`	`LogLevel.Error`
`critical`	`LogLevel.Critical`
`none`	`LogLevel.None`

> **NOTE**
> To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

For more information on logging, see the SignalR Diagnostics documentation.

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to chose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

This can safely be ignored.

Configure allowed transports

The transports used by SignalR can be configured in the `WithUrl` call (`withUrl` in JavaScript). A bitwise-OR of the values of `HttpTransportType` can be used to restrict the client to only use the specified transports. All transports are enabled by default.

For example, to disable the Server-Sent Events transport, but allow WebSockets and Long Polling connections:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", HttpTransportType.WebSockets | HttpTransportType.LongPolling)
    .Build();
```

In the JavaScript client, transports are configured by setting the `transport` field on the options object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

In this version of the Java client websockets is the only available transport.

In the Java client, the transport is selected with the `withTransport` method on the `HttpHubConnectionBuilder`. The Java client defaults to using the WebSockets transport.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withTransport(TransportEnum.WEBSOCKETS)
    .build();
```

> **NOTE**
> The SignalR Java client doesn't support transport fallback yet.

Configure bearer authentication

To provide authentication data along with SignalR requests, use the `AccessTokenProvider` option (`accessTokenFactory` in JavaScript) to specify a function that returns the desired access token. In the .NET Client, this access token is passed in as an HTTP "Bearer Authentication" token (Using the `Authorization` header with a type of `Bearer`). In the JavaScript client, the access token is used as a Bearer token, **except** in a few cases where browser APIs restrict the ability to apply headers (specifically, in Server-Sent Events and WebSockets requests). In these cases, the access token is provided as a query string value `access_token`.

In the .NET client, the `AccessTokenProvider` option can be specified using the options delegate in `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

In the JavaScript client, the access token is configured by setting the `accessTokenFactory` field on the options object in `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an access token factory to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Configure timeout and keep-alive options

Additional options for configuring timeout and keep-alive behavior are available on the `HubConnection` object itself:

- .NET
- JavaScript
- Java

OPTION	DEFAULT VALUE	DESCRIPTION
`ServerTimeout`	30 seconds (30,000 milliseconds)	Timeout for server activity. If the server hasn't sent a message in this interval, the client considers the server disconnected and triggers the `Closed` event (`onclose` in JavaScript). This value must be large enough for a ping message to be sent from the server **and** received by the client within the timeout interval. The recommended value is a number at least double the server's `KeepAliveInterval` value to allow time for pings to arrive.
`HandshakeTimeout`	15 seconds	Timeout for initial server handshake. If the server doesn't send a handshake response in this interval, the client cancels the handshake and triggers the `Closed` event (`onclose` in JavaScript). This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.

OPTION	DEFAULT VALUE	DESCRIPTION
`KeepAliveInterval`	15 seconds	Determines the interval at which the client sends ping messages. Sending any message from the client resets the timer to the start of the interval. If the client hasn't sent a message in the `ClientTimeoutInterval` set on the server, the server considers the client disconnected.

In the .NET Client, timeout values are specified as `TimeSpan` values.

Configure additional options

Additional options can be configured in the `WithUrl` (`withUrl` in JavaScript) method on `HubConnectionBuilder` or on the various configuration APIs on the `HttpHubConnectionBuilder` in the Java client:

- .NET
- JavaScript
- Java

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`AccessTokenProvider`	`null`	A function returning a string that is provided as a Bearer authentication token in HTTP requests.
`SkipNegotiation`	`false`	Set this to `true` to skip the negotiation step. **Only supported when the WebSockets transport is the only enabled transport**. This setting can't be enabled when using the Azure SignalR Service.
`ClientCertificates`	Empty	A collection of TLS certificates to send to authenticate requests.
`Cookies`	Empty	A collection of HTTP cookies to send with every HTTP request.
`Credentials`	Empty	Credentials to send with every HTTP request.
`CloseTimeout`	5 seconds	WebSockets only. The maximum amount of time the client waits after closing for the server to acknowledge the close request. If the server doesn't acknowledge the close within this time, the client disconnects.
`Headers`	Empty	A Map of additional HTTP headers to send with every HTTP request.

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`HttpMessageHandlerFactory`	`null`	A delegate that can be used to configure or replace the `HttpMessageHandler` used to send HTTP requests. Not used for WebSocket connections. This delegate must return a non-null value, and it receives the default value as a parameter. Either modify settings on that default value and return it, or return a new `HttpMessageHandler` instance. **When replacing the handler make sure to copy the settings you want to keep from the provided handler, otherwise, the configured options (such as Cookies and Headers) won't apply to the new handler.**
`Proxy`	`null`	An HTTP proxy to use when sending HTTP requests.
`UseDefaultCredentials`	`false`	Set this boolean to send the default credentials for HTTP and WebSockets requests. This enables the use of Windows authentication.
`WebSocketConfiguration`	`null`	A delegate that can be used to configure additional WebSocket options. Receives an instance of ClientWebSocketOptions that can be used to configure the options.

In the .NET Client, these options can be modified by the options delegate provided to `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.Headers["Foo"] = "Bar";
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

In the JavaScript Client, these options can be provided in a JavaScript object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        skipNegotiation: true,
        transport: signalR.HttpTransportType.WebSockets
    })
    .build();
```

In the Java client, these options can be configured with the methods on the `HttpHubConnectionBuilder` returned from the `HubConnectionBuilder.create("HUB URL")`

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
        .withHeader("Foo", "Bar")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .build();
```

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client
- Use MessagePack Hub Protocol in SignalR for ASP.NET Core
- ASP.NET Core SignalR supported platforms

JSON/MessagePack serialization options

ASP.NET Core SignalR supports two protocols for encoding messages: JSON and MessagePack. Each protocol has serialization configuration options.

JSON serialization can be configured on the server using the AddJsonProtocol extension method, which can be added after AddSignalR in your `Startup.ConfigureServices` method. The `AddJsonProtocol` method takes a delegate that receives an `options` object. The PayloadSerializerSettings property on that object is a JSON.NET `JsonSerializerSettings` object that can be used to configure serialization of arguments and return values. For more information, see the JSON.NET documentation.

As an example, to configure the serializer to use "PascalCase" property names, instead of the default "camelCase" names, use the following code in `Startup.ConfigureServices`:

```
services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerSettings.ContractResolver =
            new DefaultContractResolver();
    });
```

In the .NET client, the same `AddJsonProtocol` extension method exists on HubConnectionBuilder. The `Microsoft.Extensions.DependencyInjection` namespace must be imported to resolve the extension method:

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerSettings.ContractResolver =
            new DefaultContractResolver();
    })
    .Build();
```

> **NOTE**
> It's not possible to configure JSON serialization in the JavaScript client at this time.

MessagePack serialization options

MessagePack serialization can be configured by providing a delegate to the AddMessagePackProtocol call. See MessagePack in SignalR for more details.

> **NOTE**
> It's not possible to configure MessagePack serialization in the JavaScript client at this time.

Configure server options

The following table describes options for configuring SignalR hubs:

OPTION	DEFAULT VALUE	DESCRIPTION
`ClientTimeoutInterval`	30 seconds	The server will consider the client disconnected if it hasn't received a message (including keep-alive) in this interval. It could take longer than this timeout interval for the client to actually be marked disconnected, due to how this is implemented. The recommended value is double the `KeepAliveInterval` value.
`HandshakeTimeout`	15 seconds	If the client doesn't send an initial handshake message within this time interval, the connection is closed. This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	If the server hasn't sent a message within this interval, a ping message is sent automatically to keep the connection open. When changing `KeepAliveInterval`, change the `ServerTimeout` / `serverTimeoutInMilliseconds` setting on the client. The recommended `ServerTimeout` / `serverTimeoutInMilliseconds` value is double the `KeepAliveInterval` value.
`SupportedProtocols`	All installed protocols	Protocols supported by this hub. By default, all protocols registered on the server are allowed, but protocols can be removed from this list to disable specific protocols for individual hubs.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a Hub method. The default is `false`, as these exception messages can contain sensitive information.

Options can be configured for all hubs by providing an options delegate to the `AddSignalR` call in `Startup.ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.EnableDetailedErrors = true;
        hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
    });
}
```

Options for a single hub override the global options provided in `AddSignalR` and can be configured using AddHubOptions:

```
services.AddSignalR().AddHubOptions<ChatHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

Advanced HTTP configuration options

Use `HttpConnectionDispatcherOptions` to configure advanced settings related to transports and memory buffer management. These options are configured by passing a delegate to MapHub<T> in `Startup.Configure`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseSignalR((configure) =>
    {
        var desiredTransports =
            HttpTransportType.WebSockets |
            HttpTransportType.LongPolling;

        configure.MapHub<ChatHub>("/chathub", (options) =>
        {
            options.Transports = desiredTransports;
        });
    });
}
```

The following table describes options for configuring ASP.NET Core SignalR's advanced HTTP options:

OPTION	DEFAULT VALUE	DESCRIPTION
`ApplicationMaxBufferSize`	32 KB	The maximum number of bytes received from the client that the server buffers. Increasing this value allows the server to receive larger messages, but can negatively impact memory consumption.
`AuthorizationData`	Data automatically gathered from the `Authorize` attributes applied to the Hub class.	A list of IAuthorizeData objects used to determine if a client is authorized to connect to the hub.

OPTION	DEFAULT VALUE	DESCRIPTION
`TransportMaxBufferSize`	32 KB	The maximum number of bytes sent by the app that the server buffers. Increasing this value allows the server to send larger messages, but can negatively impact memory consumption.
`Transports`	All Transports are enabled.	A bit flags enum of `HttpTransportType` values that can restrict the transports a client can use to connect.
`LongPolling`	See below.	Additional options specific to the Long Polling transport.
`WebSockets`	See below.	Additional options specific to the WebSockets transport.

The Long Polling transport has additional options that can be configured using the `LongPolling` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`PollTimeout`	90 seconds	The maximum amount of time the server waits for a message to send to the client before terminating a single poll request. Decreasing this value causes the client to issue new poll requests more frequently.

The WebSocket transport has additional options that can be configured using the `WebSockets` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`CloseTimeout`	5 seconds	After the server closes, if the client fails to close within this time interval, the connection is terminated.
`SubProtocolSelector`	`null`	A delegate that can be used to set the `Sec-WebSocket-Protocol` header to a custom value. The delegate receives the values requested by the client as input and is expected to return the desired value.

Configure client options

Client options can be configured on the `HubConnectionBuilder` type (available in the .NET and JavaScript clients). It's also available in the Java client, but the `HttpHubConnectionBuilder` subclass is what contains the builder configuration options, as well as on the `HubConnection` itself.

Configure logging

Logging is configured in the .NET Client using the `ConfigureLogging` method. Logging providers and filters can be registered in the same way as they are on the server. See the Logging in ASP.NET Core documentation for more information.

> **NOTE**
> In order to register Logging providers, you must install the necessary packages. See the Built-in logging providers section of the docs for a full list.

For example, to enable Console logging, install the `Microsoft.Extensions.Logging.Console` NuGet package. Call the `AddConsole` extension method:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

In the JavaScript client, a similar `configureLogging` method exists. Provide a `LogLevel` value indicating the minimum level of log messages to produce. Logs are written to the browser console window.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

> **NOTE**
> To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

For more information on logging, see the SignalR Diagnostics documentation.

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to chose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

This can safely be ignored.

Configure allowed transports

The transports used by SignalR can be configured in the `WithUrl` call (`withUrl` in JavaScript). A bitwise-OR of the values of `HttpTransportType` can be used to restrict the client to only use the specified transports. All transports are enabled by default.

For example, to disable the Server-Sent Events transport, but allow WebSockets and Long Polling connections:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", HttpTransportType.WebSockets | HttpTransportType.LongPolling)
    .Build();
```

In the JavaScript client, transports are configured by setting the `transport` field on the options object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

In this version of the Java client websockets is the only available transport.

Configure bearer authentication

To provide authentication data along with SignalR requests, use the `AccessTokenProvider` option (`accessTokenFactory` in JavaScript) to specify a function that returns the desired access token. In the .NET Client, this access token is passed in as an HTTP "Bearer Authentication" token (Using the `Authorization` header with a type of `Bearer`). In the JavaScript client, the access token is used as a Bearer token, **except** in a few cases where browser APIs restrict the ability to apply headers (specifically, in Server-Sent Events and WebSockets requests). In these cases, the access token is provided as a query string value `access_token`.

In the .NET client, the `AccessTokenProvider` option can be specified using the options delegate in `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

In the JavaScript client, the access token is configured by setting the `accessTokenFactory` field on the options object in `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an access token factory to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Configure timeout and keep-alive options

Additional options for configuring timeout and keep-alive behavior are available on the `HubConnection` object itself:

- .NET
- JavaScript
- Java

OPTION	DEFAULT VALUE	DESCRIPTION
`ServerTimeout`	30 seconds (30,000 milliseconds)	Timeout for server activity. If the server hasn't sent a message in this interval, the client considers the server disconnected and triggers the `Closed` event (`onclose` in JavaScript). This value must be large enough for a ping message to be sent from the server **and** received by the client within the timeout interval. The recommended value is a number at least double the server's `KeepAliveInterval` value to allow time for pings to arrive.
`HandshakeTimeout`	15 seconds	Timeout for initial server handshake. If the server doesn't send a handshake response in this interval, the client cancels the handshake and triggers the `Closed` event (`onclose` in JavaScript). This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	Determines the interval at which the client sends ping messages. Sending any message from the client resets the timer to the start of the interval. If the client hasn't sent a message in the `ClientTimeoutInterval` set on the server, the server considers the client disconnected.

In the .NET Client, timeout values are specified as `TimeSpan` values.

Configure additional options

Additional options can be configured in the `WithUrl` (`withUrl` in JavaScript) method on `HubConnectionBuilder` or on the various configuration APIs on the `HttpHubConnectionBuilder` in the Java client:

- .NET
- JavaScript
- Java

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`AccessTokenProvider`	`null`	A function returning a string that is provided as a Bearer authentication token in HTTP requests.

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`SkipNegotiation`	`false`	Set this to `true` to skip the negotiation step. **Only supported when the WebSockets transport is the only enabled transport**. This setting can't be enabled when using the Azure SignalR Service.
`ClientCertificates`	Empty	A collection of TLS certificates to send to authenticate requests.
`Cookies`	Empty	A collection of HTTP cookies to send with every HTTP request.
`Credentials`	Empty	Credentials to send with every HTTP request.
`CloseTimeout`	5 seconds	WebSockets only. The maximum amount of time the client waits after closing for the server to acknowledge the close request. If the server doesn't acknowledge the close within this time, the client disconnects.
`Headers`	Empty	A Map of additional HTTP headers to send with every HTTP request.
`HttpMessageHandlerFactory`	`null`	A delegate that can be used to configure or replace the `HttpMessageHandler` used to send HTTP requests. Not used for WebSocket connections. This delegate must return a non-null value, and it receives the default value as a parameter. Either modify settings on that default value and return it, or return a new `HttpMessageHandler` instance. **When replacing the handler make sure to copy the settings you want to keep from the provided handler, otherwise, the configured options (such as Cookies and Headers) won't apply to the new handler.**
`Proxy`	`null`	An HTTP proxy to use when sending HTTP requests.
`UseDefaultCredentials`	`false`	Set this boolean to send the default credentials for HTTP and WebSockets requests. This enables the use of Windows authentication.
`WebSocketConfiguration`	`null`	A delegate that can be used to configure additional WebSocket options. Receives an instance of ClientWebSocketOptions that can be used to configure the options.

In the .NET Client, these options can be modified by the options delegate provided to `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.Headers["Foo"] = "Bar";
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

In the JavaScript Client, these options can be provided in a JavaScript object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        skipNegotiation: true,
        transport: signalR.HttpTransportType.WebSockets
    })
    .build();
```

In the Java client, these options can be configured with the methods on the `HttpHubConnectionBuilder` returned from the `HubConnectionBuilder.create("HUB URL")`

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
        .withHeader("Foo", "Bar")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .build();
```

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client
- Use MessagePack Hub Protocol in SignalR for ASP.NET Core
- ASP.NET Core SignalR supported platforms

JSON/MessagePack serialization options

ASP.NET Core SignalR supports two protocols for encoding messages: JSON and MessagePack. Each protocol has serialization configuration options.

JSON serialization can be configured on the server using the AddJsonProtocol extension method, which can be added after AddSignalR in your `Startup.ConfigureServices` method. The `AddJsonProtocol` method takes a delegate that receives an `options` object. The PayloadSerializerSettings property on that object is a JSON.NET `JsonSerializerSettings` object that can be used to configure serialization of arguments and return values. For more information, see the JSON.NET documentation.

As an example, to configure the serializer to use "PascalCase" property names, instead of the default "camelCase" names, use the following code in `Startup.ConfigureServices`:

```
services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerSettings.ContractResolver =
            new DefaultContractResolver();
    });
```

In the .NET client, the same `AddJsonProtocol` extension method exists on HubConnectionBuilder. The `Microsoft.Extensions.DependencyInjection` namespace must be imported to resolve the extension method:

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerSettings.ContractResolver =
            new DefaultContractResolver();
    })
    .Build();
```

> **NOTE**
> It's not possible to configure JSON serialization in the JavaScript client at this time.

MessagePack serialization options

MessagePack serialization can be configured by providing a delegate to the AddMessagePackProtocol call. See MessagePack in SignalR for more details.

> **NOTE**
> It's not possible to configure MessagePack serialization in the JavaScript client at this time.

Configure server options

The following table describes options for configuring SignalR hubs:

OPTION	DEFAULT VALUE	DESCRIPTION
`HandshakeTimeout`	15 seconds	If the client doesn't send an initial handshake message within this time interval, the connection is closed. This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.

OPTION	DEFAULT VALUE	DESCRIPTION
`KeepAliveInterval`	15 seconds	If the server hasn't sent a message within this interval, a ping message is sent automatically to keep the connection open. When changing `KeepAliveInterval`, change the `ServerTimeout` / `serverTimeoutInMilliseconds` setting on the client. The recommended `ServerTimeout` / `serverTimeoutInMilliseconds` value is double the `KeepAliveInterval` value.
`SupportedProtocols`	All installed protocols	Protocols supported by this hub. By default, all protocols registered on the server are allowed, but protocols can be removed from this list to disable specific protocols for individual hubs.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a Hub method. The default is `false`, as these exception messages can contain sensitive information.

Options can be configured for all hubs by providing an options delegate to the `AddSignalR` call in `Startup.ConfigureServices`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.EnableDetailedErrors = true;
        hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
    });
}
```

Options for a single hub override the global options provided in `AddSignalR` and can be configured using AddHubOptions:

```
services.AddSignalR().AddHubOptions<ChatHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

Advanced HTTP configuration options

Use `HttpConnectionDispatcherOptions` to configure advanced settings related to transports and memory buffer management. These options are configured by passing a delegate to MapHub<T> in `Startup.Configure`.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseSignalR((configure) =>
    {
        var desiredTransports =
            HttpTransportType.WebSockets |
            HttpTransportType.LongPolling;

        configure.MapHub<ChatHub>("/chathub", (options) =>
        {
            options.Transports = desiredTransports;
        });
    });
}
```

The following table describes options for configuring ASP.NET Core SignalR's advanced HTTP options:

OPTION	DEFAULT VALUE	DESCRIPTION
`ApplicationMaxBufferSize`	32 KB	The maximum number of bytes received from the client that the server buffers. Increasing this value allows the server to receive larger messages, but can negatively impact memory consumption.
`AuthorizationData`	Data automatically gathered from the `Authorize` attributes applied to the Hub class.	A list of IAuthorizeData objects used to determine if a client is authorized to connect to the hub.
`TransportMaxBufferSize`	32 KB	The maximum number of bytes sent by the app that the server buffers. Increasing this value allows the server to send larger messages, but can negatively impact memory consumption.
`Transports`	All Transports are enabled.	A bit flags enum of `HttpTransportType` values that can restrict the transports a client can use to connect.
`LongPolling`	See below.	Additional options specific to the Long Polling transport.
`WebSockets`	See below.	Additional options specific to the WebSockets transport.

The Long Polling transport has additional options that can be configured using the `LongPolling` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`PollTimeout`	90 seconds	The maximum amount of time the server waits for a message to send to the client before terminating a single poll request. Decreasing this value causes the client to issue new poll requests more frequently.

The WebSocket transport has additional options that can be configured using the `WebSockets` property:

OPTION	DEFAULT VALUE	DESCRIPTION
`CloseTimeout`	5 seconds	After the server closes, if the client fails to close within this time interval, the connection is terminated.
`SubProtocolSelector`	`null`	A delegate that can be used to set the `Sec-WebSocket-Protocol` header to a custom value. The delegate receives the values requested by the client as input and is expected to return the desired value.

Configure client options

Client options can be configured on the `HubConnectionBuilder` type (available in the .NET and JavaScript clients). It's also available in the Java client, but the `HttpHubConnectionBuilder` subclass is what contains the builder configuration options, as well as on the `HubConnection` itself.

Configure logging

Logging is configured in the .NET Client using the `ConfigureLogging` method. Logging providers and filters can be registered in the same way as they are on the server. See the Logging in ASP.NET Core documentation for more information.

> **NOTE**
> In order to register Logging providers, you must install the necessary packages. See the Built-in logging providers section of the docs for a full list.

For example, to enable Console logging, install the `Microsoft.Extensions.Logging.Console` NuGet package. Call the `AddConsole` extension method:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

In the JavaScript client, a similar `configureLogging` method exists. Provide a `LogLevel` value indicating the minimum level of log messages to produce. Logs are written to the browser console window.

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

> **NOTE**
> To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

For more information on logging, see the SignalR Diagnostics documentation.

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to chose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

This can safely be ignored.

Configure allowed transports

The transports used by SignalR can be configured in the `WithUrl` call (`withUrl` in JavaScript). A bitwise-OR of the values of `HttpTransportType` can be used to restrict the client to only use the specified transports. All transports are enabled by default.

For example, to disable the Server-Sent Events transport, but allow WebSockets and Long Polling connections:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", HttpTransportType.WebSockets | HttpTransportType.LongPolling)
    .Build();
```

In the JavaScript client, transports are configured by setting the `transport` field on the options object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

Configure bearer authentication

To provide authentication data along with SignalR requests, use the `AccessTokenProvider` option (`accessTokenFactory` in JavaScript) to specify a function that returns the desired access token. In the .NET Client, this access token is passed in as an HTTP "Bearer Authentication" token (Using the `Authorization` header with a type of `Bearer`). In the JavaScript client, the access token is used as a Bearer token, **except** in a few cases where browser APIs restrict the ability to apply headers (specifically, in Server-Sent Events and WebSockets requests). In these cases, the access token is provided as a query string value `access_token`.

In the .NET client, the `AccessTokenProvider` option can be specified using the options delegate in `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

In the JavaScript client, the access token is configured by setting the `accessTokenFactory` field on the options object in `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an access token factory to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Configure timeout and keep-alive options

Additional options for configuring timeout and keep-alive behavior are available on the `HubConnection` object itself:

- .NET
- JavaScript
- Java

OPTION	DEFAULT VALUE	DESCRIPTION
`ServerTimeout`	30 seconds (30,000 milliseconds)	Timeout for server activity. If the server hasn't sent a message in this interval, the client considers the server disconnected and triggers the `Closed` event (`onclose` in JavaScript). This value must be large enough for a ping message to be sent from the server **and** received by the client within the timeout interval. The recommended value is a number at least double the server's `KeepAliveInterval` value to allow time for pings to arrive.
`HandshakeTimeout`	15 seconds	Timeout for initial server handshake. If the server doesn't send a handshake response in this interval, the client cancels the handshake and triggers the `Closed` event (`onclose` in JavaScript). This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.

In the .NET Client, timeout values are specified as `TimeSpan` values.

Configure additional options

Additional options can be configured in the `WithUrl` (`withUrl` in JavaScript) method on `HubConnectionBuilder` or on the various configuration APIs on the `HttpHubConnectionBuilder` in the Java client:

- .NET
- JavaScript
- Java

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`AccessTokenProvider`	`null`	A function returning a string that is provided as a Bearer authentication token in HTTP requests.
`SkipNegotiation`	`false`	Set this to `true` to skip the negotiation step. **Only supported when the WebSockets transport is the only enabled transport**. This setting can't be enabled when using the Azure SignalR Service.
`ClientCertificates`	Empty	A collection of TLS certificates to send to authenticate requests.
`Cookies`	Empty	A collection of HTTP cookies to send with every HTTP request.
`Credentials`	Empty	Credentials to send with every HTTP request.
`CloseTimeout`	5 seconds	WebSockets only. The maximum amount of time the client waits after closing for the server to acknowledge the close request. If the server doesn't acknowledge the close within this time, the client disconnects.
`Headers`	Empty	A Map of additional HTTP headers to send with every HTTP request.
`HttpMessageHandlerFactory`	`null`	A delegate that can be used to configure or replace the `HttpMessageHandler` used to send HTTP requests. Not used for WebSocket connections. This delegate must return a non-null value, and it receives the default value as a parameter. Either modify settings on that default value and return it, or return a new `HttpMessageHandler` instance. **When replacing the handler make sure to copy the settings you want to keep from the provided handler, otherwise, the configured options (such as Cookies and Headers) won't apply to the new handler.**

.NET OPTION	DEFAULT VALUE	DESCRIPTION
`Proxy`	`null`	An HTTP proxy to use when sending HTTP requests.
`UseDefaultCredentials`	`false`	Set this boolean to send the default credentials for HTTP and WebSockets requests. This enables the use of Windows authentication.
`WebSocketConfiguration`	`null`	A delegate that can be used to configure additional WebSocket options. Receives an instance of ClientWebSocketOptions that can be used to configure the options.

In the .NET Client, these options can be modified by the options delegate provided to `WithUrl`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.Headers["Foo"] = "Bar";
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

In the JavaScript Client, these options can be provided in a JavaScript object provided to `withUrl`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        skipNegotiation: true,
        transport: signalR.HttpTransportType.WebSockets
    })
    .build();
```

In the Java client, these options can be configured with the methods on the `HttpHubConnectionBuilder` returned from the `HubConnectionBuilder.create("HUB URL")`

```
HubConnection hubConnection = HubConnectionBuilder.create("https://example.com/chathub")
        .withHeader("Foo", "Bar")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .build();
```

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client
- Use MessagePack Hub Protocol in SignalR for ASP.NET Core
- ASP.NET Core SignalR supported platforms

Authentication and authorization in ASP.NET Core SignalR

9/22/2020 • 9 minutes to read • Edit Online

By Andrew Stanton-Nurse

View or download sample code (how to download)

Authenticate users connecting to a SignalR hub

SignalR can be used with ASP.NET Core authentication to associate a user with each connection. In a hub, authentication data can be accessed from the HubConnectionContext.User property. Authentication allows the hub to call methods on all connections associated with a user. For more information, see Manage users and groups in SignalR. Multiple connections may be associated with a single user.

The following is an example of `Startup.Configure` which uses SignalR and ASP.NET Core authentication:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chat");
        endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
    });
}
```

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    app.UseAuthentication();

    app.UseSignalR(hubs =>
    {
        hubs.MapHub<ChatHub>("/chat");
    });

    app.UseMvc(routes =>
    {
        routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
    });
}
```

> **NOTE**
>
> The order in which you register the SignalR and ASP.NET Core authentication middleware matters. Always call `UseAuthentication` before `UseSignalR` so that SignalR has a user on the `HttpContext`.

Cookie authentication

In a browser-based app, cookie authentication allows your existing user credentials to automatically flow to SignalR connections. When using the browser client, no additional configuration is needed. If the user is logged in to your app, the SignalR connection automatically inherits this authentication.

Cookies are a browser-specific way to send access tokens, but non-browser clients can send them. When using the .NET Client, the `Cookies` property can be configured in the `.WithUrl` call to provide a cookie. However, using cookie authentication from the .NET client requires the app to provide an API to exchange authentication data for a cookie.

Bearer token authentication

The client can provide an access token instead of using a cookie. The server validates the token and uses it to identify the user. This validation is done only when the connection is established. During the life of the connection, the server doesn't automatically revalidate to check for token revocation.

On the server, bearer token authentication is configured using the JWT Bearer middleware.

In the JavaScript client, the token can be provided using the accessTokenFactory option.

```
// Connect, using the token we got.
this.connection = new signalR.HubConnectionBuilder()
    .withUrl("/hubs/chat", { accessTokenFactory: () => this.loginToken })
    .build();
```

In the .NET client, there's a similar AccessTokenProvider property that can be used to configure the token:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options =>
    {
        options.AccessTokenProvider = () => Task.FromResult(_myAccessToken);
    })
    .Build();
```

> **NOTE**
>
> The access token function you provide is called before **every** HTTP request made by SignalR. If you need to renew the token in order to keep the connection active (because it may expire during the connection), do so from within this function and return the updated token.

In standard web APIs, bearer tokens are sent in an HTTP header. However, SignalR is unable to set these headers in browsers when using some transports. When using WebSockets and Server-Sent Events, the token is transmitted as a query string parameter. To support this on the server, additional configuration is required:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
```

```
            .AddDefaultTokenProviders();

        services.AddAuthentication(options =>
            {
                // Identity made Cookie authentication the default.
                // However, we want JWT Bearer Auth to be the default.
                options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
                options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
            })
            .AddJwtBearer(options =>
            {
                // Configure the Authority to the expected value for your authentication provider
                // This ensures the token is appropriately validated
                options.Authority = /* TODO: Insert Authority URL here */;

                // We have to hook the OnMessageReceived event in order to
                // allow the JWT authentication handler to read the access
                // token from the query string when a WebSocket or
                // Server-Sent Events request comes in.

                // Sending the access token in the query string is required due to
                // a limitation in Browser APIs. We restrict it to only calls to the
                // SignalR hub in this code.
                // See https://docs.microsoft.com/aspnet/core/signalr/security#access-token-logging
                // for more information about security considerations when using
                // the query string to transmit the access token.
                options.Events = new JwtBearerEvents
                {
                    OnMessageReceived = context =>
                    {
                        var accessToken = context.Request.Query["access_token"];

                        // If the request is for our hub...
                        var path = context.HttpContext.Request.Path;
                        if (!string.IsNullOrEmpty(accessToken) &&
                            (path.StartsWithSegments("/hubs/chat")))
                        {
                            // Read the token out of the query string
                            context.Token = accessToken;
                        }
                        return Task.CompletedTask;
                    }
                };
            });

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

        services.AddSignalR();

        // Change to use Name as the user identifier for SignalR
        // WARNING: This requires that the source of your JWT token
        // ensures that the Name claim is unique!
        // If the Name claim isn't unique, users could receive messages
        // intended for a different user!
        services.AddSingleton<IUserIdProvider, NameUserIdProvider>();

        // Change to use email as the user identifier for SignalR
        // services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>();

        // WARNING: use *either* the NameUserIdProvider *or* the
        // EmailBasedUserIdProvider, but do not use both.
    }
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

> **NOTE**
>
> The query string is used on browsers when connecting with WebSockets and Server-Sent Events due to browser API limitations. When using HTTPS, query string values are secured by the TLS connection. However, many servers log query string values. For more information, see Security considerations in ASP.NET Core SignalR. SignalR uses headers to transmit tokens in environments which support them (such as the .NET and Java clients).

Cookies vs. bearer tokens

Cookies are specific to browsers. Sending them from other kinds of clients adds complexity compared to sending bearer tokens. Consequently, cookie authentication isn't recommended unless the app only needs to authenticate users from the browser client. Bearer token authentication is the recommended approach when using clients other than the browser client.

Windows authentication

If Windows authentication is configured in your app, SignalR can use that identity to secure hubs. However, to send messages to individual users, you need to add a custom User ID provider. The Windows authentication system doesn't provide the "Name Identifier" claim. SignalR uses the claim to determine the user name.

Add a new class that implements `IUserIdProvider` and retrieve one of the claims from the user to use as the identifier. For example, to use the "Name" claim (which is the Windows username in the form `[Domain]\[Username]`), create the following class:

```
public class NameUserIdProvider : IUserIdProvider
{
    public string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.Identity?.Name;
    }
}
```

Rather than `ClaimTypes.Name`, you can use any value from the `User` (such as the Windows SID identifier, and so on).

> **NOTE**
>
> The value you choose must be unique among all the users in your system. Otherwise, a message intended for one user could end up going to a different user.

Register this component in your `Startup.ConfigureServices` method.

```
public void ConfigureServices(IServiceCollection services)
{
    // ... other services ...

    services.AddSignalR();
    services.AddSingleton<IUserIdProvider, NameUserIdProvider>();
}
```

In the .NET Client, Windows Authentication must be enabled by setting the UseDefaultCredentials property:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options =>
    {
        options.UseDefaultCredentials = true;
    })
    .Build();
```

Windows authentication is supported in Internet Explorer and Microsoft Edge, but not in all browsers. For example, in Chrome and Safari, attempting to use Windows authentication and WebSockets fails. When Windows authentication fails, the client attempts to fall back to other transports which might work.

Use claims to customize identity handling

An app that authenticates users can derive SignalR user IDs from user claims. To specify how SignalR creates user IDs, implement `IUserIdProvider` and register the implementation.

The sample code demonstrates how you would use claims to select the user's email address as the identifying property.

> **NOTE**
> The value you choose must be unique among all the users in your system. Otherwise, a message intended for one user could end up going to a different user.

```
public class EmailBasedUserIdProvider : IUserIdProvider
{
    public virtual string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.FindFirst(ClaimTypes.Email)?.Value;
    }
}
```

The account registration adds a claim with type `ClaimsTypes.Email` to the ASP.NET identity database.

```
// create a new user
var user = new ApplicationUser { UserName = Input.Email, Email = Input.Email };
var result = await _userManager.CreateAsync(user, Input.Password);

// add the email claim and value for this user
await _userManager.AddClaimAsync(user, new Claim(ClaimTypes.Email, Input.Email));
```

Register this component in your `Startup.ConfigureServices`.

```
services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>();
```

Authorize users to access hubs and hub methods

By default, all methods in a hub can be called by an unauthenticated user. To require authentication, apply the Authorize attribute to the hub:

```
[Authorize]
public class ChatHub: Hub
{
}
```

You can use the constructor arguments and properties of the `[Authorize]` attribute to restrict access to only users matching specific authorization policies. For example, if you have a custom authorization policy called `MyAuthorizationPolicy` you can ensure that only users matching that policy can access the hub using the following code:

```
[Authorize("MyAuthorizationPolicy")]
public class ChatHub : Hub
{
}
```

Individual hub methods can have the `[Authorize]` attribute applied as well. If the current user doesn't match the policy applied to the method, an error is returned to the caller:

```
[Authorize]
public class ChatHub : Hub
{
    public async Task Send(string message)
    {
        // ... send a message to all users ...
    }

    [Authorize("Administrators")]
    public void BanUser(string userName)
    {
        // ... ban a user from the chat room (something only Administrators can do) ...
    }
}
```

Use authorization handlers to customize hub method authorization

SignalR provides a custom resource to authorization handlers when a hub method requires authorization. The resource is an instance of `HubInvocationContext`. The `HubInvocationContext` includes the `HubCallerContext`, the name of the hub method being invoked, and the arguments to the hub method.

Consider the example of a chat room allowing multiple organization sign-in via Azure Active Directory. Anyone with a Microsoft account can sign in to chat, but only members of the owning organization should be able to ban users or view users' chat histories. Furthermore, we might want to restrict certain functionality from certain users. Using the updated features in ASP.NET Core 3.0, this is entirely possible. Note how the `DomainRestrictedRequirement` serves as a custom `IAuthorizationRequirement`. Now that the `HubInvocationContext` resource parameter is being passed in, the internal logic can inspect the context in which the Hub is being called and make decisions on allowing the user to execute individual Hub methods.

```
[Authorize]
public class ChatHub : Hub
{
    public void SendMessage(string message)
    {
    }

    [Authorize("DomainRestricted")]
    public void BanUser(string username)
    {
    }

    [Authorize("DomainRestricted")]
    public void ViewUserHistory(string username)
    {
    }
}

public class DomainRestrictedRequirement :
    AuthorizationHandler<DomainRestrictedRequirement, HubInvocationContext>,
    IAuthorizationRequirement
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        DomainRestrictedRequirement requirement,
        HubInvocationContext resource)
    {
        if (IsUserAllowedToDoThis(resource.HubMethodName, context.User.Identity.Name) &&
            context.User.Identity.Name.EndsWith("@microsoft.com"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }

    private bool IsUserAllowedToDoThis(string hubMethodName,
        string currentUsername)
    {
        return !(currentUsername.Equals("asdf42@microsoft.com") &&
            hubMethodName.Equals("banUser", StringComparison.OrdinalIgnoreCase));
    }
}
```

In `Startup.ConfigureServices`, add the new policy, providing the custom `DomainRestrictedRequirement` requirement as a parameter to create the `DomainRestricted` policy.

```
public void ConfigureServices(IServiceCollection services)
{
    // ... other services ...

    services
        .AddAuthorization(options =>
        {
            options.AddPolicy("DomainRestricted", policy =>
            {
                policy.Requirements.Add(new DomainRestrictedRequirement());
            });
        });
}
```

In the preceding example, the `DomainRestrictedRequirement` class is both an `IAuthorizationRequirement` and its own `AuthorizationHandler` for that requirement. It's acceptable to split these two components into separate classes to separate concerns. A benefit of the example's approach is there's no need to inject the `AuthorizationHandler` during startup, as the requirement and the handler are the same thing.

Additional resources

- Bearer Token Authentication in ASP.NET Core
- Resource-based Authorization

Security considerations in ASP.NET Core SignalR

9/22/2020 • 5 minutes to read • Edit Online

By Andrew Stanton-Nurse

This article provides information on securing SignalR.

Cross-origin resource sharing

Cross-origin resource sharing (CORS) can be used to allow cross-origin SignalR connections in the browser. If JavaScript code is hosted on a different domain from the SignalR app, CORS middleware must be enabled to allow the JavaScript to connect to the SignalR app. Allow cross-origin requests only from domains you trust or control. For example:

- Your site is hosted on `http://www.example.com`
- Your SignalR app is hosted on `http://signalr.example.com`

CORS should be configured in the SignalR app to only allow the origin `www.example.com`.

For more information on configuring CORS, see Enable Cross-Origin Requests (CORS). SignalR **requires** the following CORS policies:

- Allow the specific expected origins. Allowing any origin is possible but is **not** secure or recommended.
- HTTP methods `GET` and `POST` must be allowed.
- Credentials must be allowed in order for cookie-based sticky sessions to work correctly. They must be enabled even when authentication isn't used.

However, in 5.0 we have provided an option in the TypeScript client to not use credentials. The option to not use credentials should only be used when you know 100% that credentials like Cookies are not needed in your app (cookies are used by azure app service when using multiple servers for sticky sessions).

For example, the following CORS policy allows a SignalR browser client hosted on `https://example.com` to access the SignalR app hosted on `https://signalr.example.com`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // ... other middleware ...

    // Make sure the CORS middleware is ahead of SignalR.
    app.UseCors(builder =>
    {
        builder.WithOrigins("https://example.com")
            .AllowAnyHeader()
            .WithMethods("GET", "POST")
            .AllowCredentials();
    });

    // ... other middleware ...
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chathub");
    });

    // ... other middleware ...
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // ... other middleware ...

    // Make sure the CORS middleware is ahead of SignalR.
    app.UseCors(builder =>
    {
        builder.WithOrigins("https://example.com")
            .AllowAnyHeader()
            .WithMethods("GET", "POST")
            .AllowCredentials();
    });

    // ... other middleware ...

    app.UseSignalR(routes =>
    {
        routes.MapHub<ChatHub>("/chathub");
    });

    // ... other middleware ...
}
```

WebSocket Origin Restriction

The protections provided by CORS don't apply to WebSockets. For origin restriction on WebSockets, read WebSockets origin restriction.

The protections provided by CORS don't apply to WebSockets. Browsers do **not**:

- Perform CORS pre-flight requests.
- Respect the restrictions specified in `Access-Control` headers when making WebSocket requests.

However, browsers do send the `Origin` header when issuing WebSocket requests. Applications should be configured to validate these headers to ensure that only WebSockets coming from the expected origins are allowed.

In ASP.NET Core 2.1 and later, header validation can be achieved using a custom middleware placed **before** `UseSignalR`, **and authentication middleware** in `Configure`:

```
// In Startup, add a static field listing the allowed Origin values:
private static readonly HashSet<string> _allowedOrigins = new HashSet<string>()
{
    // Add allowed origins here. For example:
    "https://www.mysite.com",
    "https://mysite.com",
};

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // ... other middleware ...

    // Validate Origin header on WebSocket requests to prevent unexpected cross-site
    // WebSocket requests.
    app.Use((context, next) =>
    {
        // Check for a WebSocket request.
        if (string.Equals(context.Request.Headers["Upgrade"], "websocket"))
        {
            var origin = context.Request.Headers["Origin"];

            // If there is an origin header, and the origin header doesn't match
            // an allowed value:
            if (!string.IsNullOrEmpty(origin) && !_allowedOrigins.Contains(origin))
            {
                // The origin is not allowed, reject the request
                context.Response.StatusCode = StatusCodes.Status403Forbidden;
                return Task.CompletedTask;
            }
        }

        // The request is a valid Origin or not a WebSocket request, so continue.
        return next();
    });

    // ... other middleware ...

    app.UseSignalR(routes =>
    {
        routes.MapHub<ChatHub>("/chathub");
    });

    // ... other middleware ...
}
```

> **NOTE**
>
> The `Origin` header is controlled by the client and, like the `Referer` header, can be faked. These headers should **not** be used as an authentication mechanism.

ConnectionId

Exposing `ConnectionId` can lead to malicious impersonation if the SignalR server or client version is ASP.NET Core 2.2 or earlier. If the SignalR server and client version are ASP.NET Core 3.0 or later, the `ConnectionToken` rather than the `ConnectionId` must be kept secret. The `ConnectionToken` is purposely not exposed in any API. It can be difficult to ensure that older SignalR clients aren't connecting to the server, so even if your SignalR server version is

ASP.NET Core 3.0 or later, the `ConnectionId` shouldn't be exposed.

Access token logging

When using WebSockets or Server-Sent Events, the browser client sends the access token in the query string. Receiving the access token via query string is generally secure as using the standard `Authorization` header. Always use HTTPS to ensure a secure end-to-end connection between the client and the server. Many web servers log the URL for each request, including the query string. Logging the URLs may log the access token. ASP.NET Core logs the URL for each request by default, which will include the query string. For example:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/chathub?access_token=1234
```

If you have concerns about logging this data with your server logs, you can disable this logging entirely by configuring the `Microsoft.AspNetCore.Hosting` logger to the `Warning` level or above (these messages are written at `Info` level). For more information, see Log Filtering for more information. If you still want to log certain request information, you can write a middleware to log the data you require and filter out the `access_token` query string value (if present).

Exceptions

Exception messages are generally considered sensitive data that shouldn't be revealed to a client. By default, SignalR doesn't send the details of an exception thrown by a hub method to the client. Instead, the client receives a generic message indicating an error occurred. Exception message delivery to the client can be overridden (for example in development or test) with EnableDetailedErrors. Exception messages should not be exposed to the client in production apps.

Buffer management

SignalR uses per-connection buffers to manage incoming and outgoing messages. By default, SignalR limits these buffers to 32 KB. The largest message a client or server can send is 32 KB. The maximum memory consumed by a connection for messages is 32 KB. If your messages are always smaller than 32 KB, you can reduce the limit, which:

- Prevents a client from being able to send a larger message.
- The server will never need to allocate large buffers to accept messages.

If your messages are larger than 32 KB, you can increase the limit. Increasing this limit means:

- The client can cause the server to allocate large memory buffers.
- Server allocation of large buffers may reduce the number of concurrent connections.

There are limits for incoming and outgoing messages, both can be configured on the HttpConnectionDispatcherOptions object configured in `MapHub`:

- `ApplicationMaxBufferSize` represents the maximum number of bytes from the client that the server buffers. If the client attempts to send a message larger than this limit, the connection may be closed.
- `TransportMaxBufferSize` represents the maximum number of bytes the server can send. If the server attempts to send a message (including return values from hub methods) larger than this limit, an exception will be thrown.

Setting the limit to `0` disables the limit. Removing the limit allows a client to send a message of any size. Malicious clients sending large messages can cause excess memory to be allocated. Excess memory usage can significantly reduce the number of concurrent connections.

Use MessagePack Hub Protocol in SignalR for ASP.NET Core

9/22/2020 • 15 minutes to read • Edit Online

This article assumes the reader is familiar with the topics covered in Get Started.

What is MessagePack?

MessagePack is a fast and compact binary serialization format. It's useful when performance and bandwidth are a concern because it creates smaller messages compared to JSON. The binary messages are unreadable when looking at network traces and logs unless the bytes are passed through a MessagePack parser. SignalR has built-in support for the MessagePack format and provides APIs for the client and server to use.

Configure MessagePack on the server

To enable the MessagePack Hub Protocol on the server, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package in your app. In the `Startup.ConfigureServices` method, add `AddMessagePackProtocol` to the `AddSignalR` call to enable MessagePack support on the server.

> **NOTE**
> JSON is enabled by default. Adding MessagePack enables support for both JSON and MessagePack clients.

```
services.AddSignalR()
    .AddMessagePackProtocol();
```

To customize how MessagePack will format your data, `AddMessagePackProtocol` takes a delegate for configuring options. In that delegate, the `SerializerOptions` property can be used to configure MessagePack serialization options. For more information on how the resolvers work, visit the MessagePack library at MessagePack-CSharp. Attributes can be used on the objects you want to serialize to define how they should be handled.

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.SerializerOptions = MessagePackSerializerOptions.Standard
            .WithResolver(new CustomResolver())
            .WithSecurity(MessagePackSecurity.UntrustedData);
    });
```

> **WARNING**
> We strongly recommend reviewing CVE-2020-5234 and applying the recommended patches. For example, calling `.WithSecurity(MessagePackSecurity.UntrustedData)` when replacing the `SerializerOptions`.

Configure MessagePack on the client

> **NOTE**
> JSON is enabled by default for the supported clients. Clients can only support a single protocol. Adding MessagePack support will replace any previously configured protocols.

.NET client

To enable MessagePack in the .NET Client, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package and call `AddMessagePackProtocol` on `HubConnectionBuilder`.

```
var hubConnection = new HubConnectionBuilder()
                        .WithUrl("/chathub")
                        .AddMessagePackProtocol()
                        .Build();
```

> **NOTE**
> This `AddMessagePackProtocol` call takes a delegate for configuring options just like the server.

JavaScript client

MessagePack support for the JavaScript client is provided by the @microsoft/signalr-protocol-msgpack npm package. Install the package by executing the following command in a command shell:

```
npm install @microsoft/signalr-protocol-msgpack
```

After installing the npm package, the module can be used directly via a JavaScript module loader or imported into the browser by referencing the following file:

node_modules\@microsoft\signalr-protocol-msgpack\dist\browser\signalr-protocol-msgpack.js

In a browser, the `msgpack5` library must also be referenced. Use a `<script>` tag to create a reference. The library can be found at *node_modules\msgpack5\dist\msgpack5.js*.

> **NOTE**
> When using the `<script>` element, the order is important. If *signalr-protocol-msgpack.js* is referenced before *msgpack5.js*, an error occurs when trying to connect with MessagePack. *signalr.js* is also required before *signalr-protocol-msgpack.js*.

```
<script src="~/lib/signalr/signalr.js"></script>
<script src="~/lib/msgpack5/msgpack5.js"></script>
<script src="~/lib/signalr/signalr-protocol-msgpack.js"></script>
```

Adding `.withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())` to the `HubConnectionBuilder` will configure the client to use the MessagePack protocol when connecting to a server.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
    .build();
```

> **NOTE**
> At this time, there are no configuration options for the MessagePack protocol on the JavaScript client.

MessagePack quirks

There are a few issues to be aware of when using the MessagePack Hub Protocol.

MessagePack is case-sensitive

The MessagePack protocol is case-sensitive. For example, consider the following C# class:

```
public class ChatMessage
{
    public string Sender { get; }
    public string Message { get; }
}
```

When sending from the JavaScript client, you must use `PascalCased` property names, since the casing must match the C# class exactly. For example:

```
connection.invoke("SomeMethod", { Sender: "Sally", Message: "Hello!" });
```

Using `camelCased` names won't properly bind to the C# class. You can work around this by using the `Key` attribute to specify a different name for the MessagePack property. For more information, see the MessagePack-CSharp documentation.

DateTime.Kind is not preserved when serializing/deserializing

The MessagePack protocol doesn't provide a way to encode the `Kind` value of a `DateTime`. As a result, when deserializing a date, the MessagePack Hub Protocol will convert to the UTC format if the `DateTime.Kind` is `DateTimeKind.Local` otherwise it will not touch the time and pass it as is. If you're working with `DateTime` values, we recommend converting to UTC before sending them. Convert them from UTC to local time when you receive them.

DateTime.MinValue is not supported by MessagePack in JavaScript

The msgpack5 library used by the SignalR JavaScript client doesn't support the `timestamp96` type in MessagePack. This type is used to encode very large date values (either very early in the past or very far in the future). The value of `DateTime.MinValue` is `January 1, 0001`, which must be encoded in a `timestamp96` value. Because of this, sending `DateTime.MinValue` to a JavaScript client isn't supported. When `DateTime.MinValue` is received by the JavaScript client, the following error is thrown:

```
Uncaught Error: unable to find ext type 255 at decoder.js:427
```

Usually, `DateTime.MinValue` is used to encode a "missing" or `null` value. If you need to encode that value in MessagePack, use a nullable `DateTime` value (`DateTime?`) or encode a separate `bool` value indicating if the date is present.

For more information on this limitation, see GitHub issue aspnet/SignalR#2228.

MessagePack support in "ahead-of-time" compilation environment

The MessagePack-CSharp library used by the .NET client and server uses code generation to optimize serialization. As a result, it isn't supported by default on environments that use "ahead-of-time" compilation (such as Xamarin iOS or Unity). It's possible to use MessagePack in these environments by "pre-generating" the

serializer/deserializer code. For more information, see the MessagePack-CSharp documentation. Once you have pre-generated the serializers, you can register them using the configuration delegate passed to `AddMessagePackProtocol`:

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        StaticCompositeResolver.Instance.Register(
            MessagePack.Resolvers.GeneratedResolver.Instance,
            MessagePack.Resolvers.StandardResolver.Instance
        );
        options.SerializerOptions = MessagePackSerializerOptions.Standard
            .WithResolver(StaticCompositeResolver.Instance)
            .WithSecurity(MessagePackSecurity.UntrustedData);
    });
```

Type checks are more strict in MessagePack

The JSON Hub Protocol will perform type conversions during deserialization. For example, if the incoming object has a property value that is a number (`{ foo: 42 }`) but the property on the .NET class is of type `string`, the value will be converted. However, MessagePack doesn't perform this conversion and will throw an exception that can be seen in server-side logs (and in the console):

```
InvalidDataException: Error binding arguments. Make sure that the types of the provided values match the
types of the hub method being invoked.
```

For more information on this limitation, see GitHub issue aspnet/SignalR#2937.

Related resources

- Get Started
- .NET client
- JavaScript client

This article assumes the reader is familiar with the topics covered in Get Started.

What is MessagePack?

MessagePack is a fast and compact binary serialization format. It's useful when performance and bandwidth are a concern because it creates smaller messages compared to JSON. The binary messages are unreadable when looking at network traces and logs unless the bytes are passed through a MessagePack parser. SignalR has built-in support for the MessagePack format, and provides APIs for the client and server to use.

Configure MessagePack on the server

To enable the MessagePack Hub Protocol on the server, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package in your app. In the `Startup.ConfigureServices` method, add `AddMessagePackProtocol` to the `AddSignalR` call to enable MessagePack support on the server.

> **NOTE**
> JSON is enabled by default. Adding MessagePack enables support for both JSON and MessagePack clients.

```
services.AddSignalR()
    .AddMessagePackProtocol();
```

To customize how MessagePack will format your data, `AddMessagePackProtocol` takes a delegate for configuring options. In that delegate, the `FormatterResolvers` property can be used to configure MessagePack serialization options. For more information on how the resolvers work, visit the MessagePack library at MessagePack-CSharp. Attributes can be used on the objects you want to serialize to define how they should be handled.

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.FormatterResolvers = new List<MessagePack.IFormatterResolver>()
        {
            MessagePack.Resolvers.StandardResolver.Instance
        };
    });
```

WARNING

We strongly recommend reviewing CVE-2020-5234 and applying the recommended patches. For example, setting the `MessagePackSecurity.Active` static property to `MessagePackSecurity.UntrustedData`. Setting the `MessagePackSecurity.Active` requires manually installing a 1.9.x version of MessagePack. Installing `MessagePack` 1.9.x upgrades the version SignalR uses. When `MessagePackSecurity.Active` is not set to `MessagePackSecurity.UntrustedData`, a malicious client could cause a denial of service. Set `MessagePackSecurity.Active` in `Program.Main`, as shown in the following code:

```
public static void Main(string[] args)
{
  MessagePackSecurity.Active = MessagePackSecurity.UntrustedData;

  CreateHostBuilder(args).Build().Run();
}
```

Configure MessagePack on the client

NOTE

JSON is enabled by default for the supported clients. Clients can only support a single protocol. Adding MessagePack support will replace any previously configured protocols.

.NET client

To enable MessagePack in the .NET Client, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package and call `AddMessagePackProtocol` on `HubConnectionBuilder`.

```
var hubConnection = new HubConnectionBuilder()
                        .WithUrl("/chathub")
                        .AddMessagePackProtocol()
                        .Build();
```

> **NOTE**
>
> This `AddMessagePackProtocol` call takes a delegate for configuring options just like the server.

JavaScript client

MessagePack support for the JavaScript client is provided by the @microsoft/signalr-protocol-msgpack npm package. Install the package by executing the following command in a command shell:

```
npm install @microsoft/signalr-protocol-msgpack
```

After installing the npm package, the module can be used directly via a JavaScript module loader or imported into the browser by referencing the following file:

node_modules\@microsoft\signalr-protocol-msgpack\dist\browser\signalr-protocol-msgpack.js

In a browser, the `msgpack5` library must also be referenced. Use a `<script>` tag to create a reference. The library can be found at *node_modules\msgpack5\dist\msgpack5.js*.

> **NOTE**
>
> When using the `<script>` element, the order is important. If *signalr-protocol-msgpack.js* is referenced before *msgpack5.js*, an error occurs when trying to connect with MessagePack. *signalr.js* is also required before *signalr-protocol-msgpack.js*.

```
<script src="~/lib/signalr/signalr.js"></script>
<script src="~/lib/msgpack5/msgpack5.js"></script>
<script src="~/lib/signalr/signalr-protocol-msgpack.js"></script>
```

Adding `.withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())` to the `HubConnectionBuilder` will configure the client to use the MessagePack protocol when connecting to a server.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
    .build();
```

> **NOTE**
>
> At this time, there are no configuration options for the MessagePack protocol on the JavaScript client.

MessagePack quirks

There are a few issues to be aware of when using the MessagePack Hub Protocol.

MessagePack is case-sensitive

The MessagePack protocol is case-sensitive. For example, consider the following C# class:

```
public class ChatMessage
{
    public string Sender { get; }
    public string Message { get; }
}
```

When sending from the JavaScript client, you must use `PascalCased` property names, since the casing must match the C# class exactly. For example:

```
connection.invoke("SomeMethod", { Sender: "Sally", Message: "Hello!" });
```

Using `camelCased` names won't properly bind to the C# class. You can work around this by using the `Key` attribute to specify a different name for the MessagePack property. For more information, see the MessagePack-CSharp documentation.

DateTime.Kind is not preserved when serializing/deserializing

The MessagePack protocol doesn't provide a way to encode the `Kind` value of a `DateTime`. As a result, when deserializing a date, the MessagePack Hub Protocol assumes the incoming date is in UTC format. If you're working with `DateTime` values in local time, we recommend converting to UTC before sending them. Convert them from UTC to local time when you receive them.

For more information on this limitation, see GitHub issue aspnet/SignalR#2632.

DateTime.MinValue is not supported by MessagePack in JavaScript

The msgpack5 library used by the SignalR JavaScript client doesn't support the `timestamp96` type in MessagePack. This type is used to encode very large date values (either very early in the past or very far in the future). The value of `DateTime.MinValue` is `January 1, 0001`, which must be encoded in a `timestamp96` value. Because of this, sending `DateTime.MinValue` to a JavaScript client isn't supported. When `DateTime.MinValue` is received by the JavaScript client, the following error is thrown:

```
Uncaught Error: unable to find ext type 255 at decoder.js:427
```

Usually, `DateTime.MinValue` is used to encode a "missing" or `null` value. If you need to encode that value in MessagePack, use a nullable `DateTime` value (`DateTime?`) or encode a separate `bool` value indicating if the date is present.

For more information on this limitation, see GitHub issue aspnet/SignalR#2228.

MessagePack support in "ahead-of-time" compilation environment

The MessagePack-CSharp library used by the .NET client and server uses code generation to optimize serialization. As a result, it isn't supported by default on environments that use "ahead-of-time" compilation (such as Xamarin iOS or Unity). It's possible to use MessagePack in these environments by "pre-generating" the serializer/deserializer code. For more information, see the MessagePack-CSharp documentation. Once you have pre-generated the serializers, you can register them using the configuration delegate passed to `AddMessagePackProtocol`:

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.FormatterResolvers = new List<MessagePack.IFormatterResolver>()
        {
            MessagePack.Resolvers.GeneratedResolver.Instance,
            MessagePack.Resolvers.StandardResolver.Instance
        };
    });
```

Type checks are more strict in MessagePack

The JSON Hub Protocol will perform type conversions during deserialization. For example, if the incoming object has a property value that is a number (`{ foo: 42 }`) but the property on the .NET class is of type `string`, the value will be converted. However, MessagePack doesn't perform this conversion and will throw an exception that can be seen in server-side logs (and in the console):

```
InvalidDataException: Error binding arguments. Make sure that the types of the provided values match the
types of the hub method being invoked.
```

For more information on this limitation, see GitHub issue aspnet/SignalR#2937.

Related resources

- Get Started
- .NET client
- JavaScript client

This article assumes the reader is familiar with the topics covered in Get Started.

What is MessagePack?

MessagePack is a fast and compact binary serialization format. It's useful when performance and bandwidth are a concern because it creates smaller messages compared to JSON. The binary messages are unreadable when looking at network traces and logs unless the bytes are passed through a MessagePack parser. SignalR has built-in support for the MessagePack format, and provides APIs for the client and server to use.

Configure MessagePack on the server

To enable the MessagePack Hub Protocol on the server, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package in your app. In the `Startup.ConfigureServices` method, add `AddMessagePackProtocol` to the `AddSignalR` call to enable MessagePack support on the server.

> **NOTE**
> JSON is enabled by default. Adding MessagePack enables support for both JSON and MessagePack clients.

```
services.AddSignalR()
    .AddMessagePackProtocol();
```

To customize how MessagePack will format your data, `AddMessagePackProtocol` takes a delegate for configuring options. In that delegate, the `FormatterResolvers` property can be used to configure MessagePack serialization options. For more information on how the resolvers work, visit the MessagePack library at MessagePack-CSharp.

Attributes can be used on the objects you want to serialize to define how they should be handled.

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.FormatterResolvers = new List<MessagePack.IFormatterResolver>()
        {
            MessagePack.Resolvers.StandardResolver.Instance
        };
    });
```

WARNING

We strongly recommend reviewing CVE-2020-5234 and applying the recommended patches. For example, setting the `MessagePackSecurity.Active` static property to `MessagePackSecurity.UntrustedData`. Setting the `MessagePackSecurity.Active` requires manually installing a 1.9.x version of MessagePack. Installing `MessagePack` 1.9.x upgrades the version SignalR uses. When `MessagePackSecurity.Active` is not set to `MessagePackSecurity.UntrustedData`, a malicious client could cause a denial of service. Set `MessagePackSecurity.Active` in `Program.Main`, as shown in the following code:

```
public static void Main(string[] args)
{
  MessagePackSecurity.Active = MessagePackSecurity.UntrustedData;

  CreateHostBuilder(args).Build().Run();
}
```

Configure MessagePack on the client

NOTE

JSON is enabled by default for the supported clients. Clients can only support a single protocol. Adding MessagePack support will replace any previously configured protocols.

.NET client

To enable MessagePack in the .NET Client, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package and call `AddMessagePackProtocol` on `HubConnectionBuilder`.

```
var hubConnection = new HubConnectionBuilder()
                        .WithUrl("/chathub")
                        .AddMessagePackProtocol()
                        .Build();
```

NOTE

This `AddMessagePackProtocol` call takes a delegate for configuring options just like the server.

JavaScript client

MessagePack support for the JavaScript client is provided by the @aspnet/signalr-protocol-msgpack npm package. Install the package by executing the following command in a command shell:

```
npm install @aspnet/signalr-protocol-msgpack
```

After installing the npm package, the module can be used directly via a JavaScript module loader or imported into the browser by referencing the following file:

node_modules\@aspnet\signalr-protocol-msgpack\dist\browser\signalr-protocol-msgpack.js

In a browser, the `msgpack5` library must also be referenced. Use a `<script>` tag to create a reference. The library can be found at *node_modules\msgpack5\dist\msgpack5.js*.

> **NOTE**
>
> When using the `<script>` element, the order is important. If *signalr-protocol-msgpack.js* is referenced before *msgpack5.js*, an error occurs when trying to connect with MessagePack. *signalr.js* is also required before *signalr-protocol-msgpack.js*.

```
<script src="~/lib/signalr/signalr.js"></script>
<script src="~/lib/msgpack5/msgpack5.js"></script>
<script src="~/lib/signalr/signalr-protocol-msgpack.js"></script>
```

Adding `.withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())` to the `HubConnectionBuilder` will configure the client to use the MessagePack protocol when connecting to a server.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
    .build();
```

> **NOTE**
>
> At this time, there are no configuration options for the MessagePack protocol on the JavaScript client.

MessagePack quirks

There are a few issues to be aware of when using the MessagePack Hub Protocol.

MessagePack is case-sensitive

The MessagePack protocol is case-sensitive. For example, consider the following C# class:

```
public class ChatMessage
{
    public string Sender { get; }
    public string Message { get; }
}
```

When sending from the JavaScript client, you must use `PascalCased` property names, since the casing must match the C# class exactly. For example:

```
connection.invoke("SomeMethod", { Sender: "Sally", Message: "Hello!" });
```

Using `camelCased` names won't properly bind to the C# class. You can work around this by using the `Key`

attribute to specify a different name for the MessagePack property. For more information, see the MessagePack-CSharp documentation.

DateTime.Kind is not preserved when serializing/deserializing

The MessagePack protocol doesn't provide a way to encode the `Kind` value of a `DateTime`. As a result, when deserializing a date, the MessagePack Hub Protocol assumes the incoming date is in UTC format. If you're working with `DateTime` values in local time, we recommend converting to UTC before sending them. Convert them from UTC to local time when you receive them.

For more information on this limitation, see GitHub issue aspnet/SignalR#2632.

DateTime.MinValue is not supported by MessagePack in JavaScript

The msgpack5 library used by the SignalR JavaScript client doesn't support the `timestamp96` type in MessagePack. This type is used to encode very large date values (either very early in the past or very far in the future). The value of `DateTime.MinValue` is `January 1, 0001` which must be encoded in a `timestamp96` value. Because of this, sending `DateTime.MinValue` to a JavaScript client isn't supported. When `DateTime.MinValue` is received by the JavaScript client, the following error is thrown:

```
Uncaught Error: unable to find ext type 255 at decoder.js:427
```

Usually, `DateTime.MinValue` is used to encode a "missing" or `null` value. If you need to encode that value in MessagePack, use a nullable `DateTime` value (`DateTime?`) or encode a separate `bool` value indicating if the date is present.

For more information on this limitation, see GitHub issue aspnet/SignalR#2228.

MessagePack support in "ahead-of-time" compilation environment

The MessagePack-CSharp library used by the .NET client and server uses code generation to optimize serialization. As a result, it isn't supported by default on environments that use "ahead-of-time" compilation (such as Xamarin iOS or Unity). It's possible to use MessagePack in these environments by "pre-generating" the serializer/deserializer code. For more information, see the MessagePack-CSharp documentation. Once you have pre-generated the serializers, you can register them using the configuration delegate passed to `AddMessagePackProtocol`:

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.FormatterResolvers = new List<MessagePack.IFormatterResolver>()
        {
            MessagePack.Resolvers.GeneratedResolver.Instance,
            MessagePack.Resolvers.StandardResolver.Instance
        };
    });
```

Type checks are more strict in MessagePack

The JSON Hub Protocol will perform type conversions during deserialization. For example, if the incoming object has a property value that is a number (`{ foo: 42 }`) but the property on the .NET class is of type `string`, the value will be converted. However, MessagePack doesn't perform this conversion and will throw an exception that can be seen in server-side logs (and in the console):

```
InvalidDataException: Error binding arguments. Make sure that the types of the provided values match the
types of the hub method being invoked.
```

For more information on this limitation, see GitHub issue aspnet/SignalR#2937.

Related resources

- Get Started
- .NET client
- JavaScript client

Use streaming in ASP.NET Core SignalR

9/22/2020 • 9 minutes to read • Edit Online

By Brennan Conroy

ASP.NET Core SignalR supports streaming from client to server and from server to client. This is useful for scenarios where fragments of data arrive over time. When streaming, each fragment is sent to the client or server as soon as it becomes available, rather than waiting for all of the data to become available.

ASP.NET Core SignalR supports streaming return values of server methods. This is useful for scenarios where fragments of data arrive over time. When a return value is streamed to the client, each fragment is sent to the client as soon as it becomes available, rather than waiting for all the data to become available.

View or download sample code (how to download)

Set up a hub for streaming

A hub method automatically becomes a streaming hub method when it returns IAsyncEnumerable<T>, ChannelReader<T>, `Task<IAsyncEnumerable<T>>`, or `Task<ChannelReader<T>>`.

A hub method automatically becomes a streaming hub method when it returns a ChannelReader<T> or a `Task<ChannelReader<T>>`.

Server-to-client streaming

Streaming hub methods can return `IAsyncEnumerable<T>` in addition to `ChannelReader<T>`. The simplest way to return `IAsyncEnumerable<T>` is by making the hub method an async iterator method as the following sample demonstrates. Hub async iterator methods can accept a `CancellationToken` parameter that's triggered when the client unsubscribes from the stream. Async iterator methods avoid problems common with Channels, such as not returning the `ChannelReader` early enough or exiting the method without completing the ChannelWriter<T>.

> **NOTE**
> The following sample requires C# 8.0 or later.

```
public class AsyncEnumerableHub : Hub
{
    public async IAsyncEnumerable<int> Counter(
        int count,
        int delay,
        [EnumeratorCancellation]
        CancellationToken cancellationToken)
    {
        for (var i = 0; i < count; i++)
        {
            // Check the cancellation token regularly so that the server will stop
            // producing items if the client disconnects.
            cancellationToken.ThrowIfCancellationRequested();

            yield return i;

            // Use the cancellationToken in other APIs that accept cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
}
```

The following sample shows the basics of streaming data to the client using Channels. Whenever an object is written to the ChannelWriter<T>, the object is immediately sent to the client. At the end, the `ChannelWriter` is completed to tell the client the stream is closed.

NOTE

Write to the `ChannelWriter<T>` on a background thread and return the `ChannelReader` as soon as possible. Other hub invocations are blocked until a `ChannelReader` is returned.

Wrap logic in a `try ... catch`. Complete the `Channel` in the `catch` and outside the `catch` to make sure the hub method invocation is completed properly.

```
public ChannelReader<int> Counter(
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    var channel = Channel.CreateUnbounded<int>();

    // We don't want to await WriteItemsAsync, otherwise we'd end up waiting
    // for all the items to be written before returning the channel back to
    // the client.
    _ = WriteItemsAsync(channel.Writer, count, delay, cancellationToken);

    return channel.Reader;
}

private async Task WriteItemsAsync(
    ChannelWriter<int> writer,
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    Exception localException = null;
    try
    {
        for (var i = 0; i < count; i++)
        {
            await writer.WriteAsync(i, cancellationToken);

            // Use the cancellationToken in other APIs that accept cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
    catch (Exception ex)
    {
        localException = ex;
    }

    writer.Complete(localException);
}
```

```
public class StreamHub : Hub
{
    public ChannelReader<int> Counter(
        int count,
        int delay,
        CancellationToken cancellationToken)
    {
        var channel = Channel.CreateUnbounded<int>();

        // We don't want to await WriteItemsAsync, otherwise we'd end up waiting
        // for all the items to be written before returning the channel back to
        // the client.
        _ = WriteItemsAsync(channel.Writer, count, delay, cancellationToken);

        return channel.Reader;
    }

    private async Task WriteItemsAsync(
        ChannelWriter<int> writer,
        int count,
        int delay,
        CancellationToken cancellationToken)
    {
        try
        {
            for (var i = 0; i < count; i++)
            {
                // Check the cancellation token regularly so that the server will stop
                // producing items if the client disconnects.
                cancellationToken.ThrowIfCancellationRequested();
                await writer.WriteAsync(i);

                // Use the cancellationToken in other APIs that accept cancellation
                // tokens so the cancellation can flow down to them.
                await Task.Delay(delay, cancellationToken);
            }
        }
        catch (Exception ex)
        {
            writer.TryComplete(ex);
        }

        writer.TryComplete();
    }
}
```

```
public class StreamHub : Hub
{
    public ChannelReader<int> Counter(int count, int delay)
    {
        var channel = Channel.CreateUnbounded<int>();

        // We don't want to await WriteItemsAsync, otherwise we'd end up waiting
        // for all the items to be written before returning the channel back to
        // the client.
        _ = WriteItemsAsync(channel.Writer, count, delay);

        return channel.Reader;
    }

    private async Task WriteItemsAsync(
        ChannelWriter<int> writer,
        int count,
        int delay)
    {
        try
        {
            for (var i = 0; i < count; i++)
            {
                await writer.WriteAsync(i);
                await Task.Delay(delay);
            }
        }
        catch (Exception ex)
        {
            writer.TryComplete(ex);
        }

        writer.TryComplete();
    }
}
```

Server-to-client streaming hub methods can accept a `CancellationToken` parameter that's triggered when the client unsubscribes from the stream. Use this token to stop the server operation and release any resources if the client disconnects before the end of the stream.

Client-to-server streaming

A hub method automatically becomes a client-to-server streaming hub method when it accepts one or more objects of type ChannelReader<T> or IAsyncEnumerable<T>. The following sample shows the basics of reading streaming data sent from the client. Whenever the client writes to the ChannelWriter<T>, the data is written into the `ChannelReader` on the server from which the hub method is reading.

```
public async Task UploadStream(ChannelReader<string> stream)
{
    while (await stream.WaitToReadAsync())
    {
        while (stream.TryRead(out var item))
        {
            // do something with the stream item
            Console.WriteLine(item);
        }
    }
}
```

An IAsyncEnumerable<T> version of the method follows.

> **NOTE**
> The following sample requires C# 8.0 or later.

```
public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        Console.WriteLine(item);
    }
}
```

.NET client

Server-to-client streaming

The `StreamAsync` and `StreamAsChannelAsync` methods on `HubConnection` are used to invoke server-to-client streaming methods. Pass the hub method name and arguments defined in the hub method to `StreamAsync` or `StreamAsChannelAsync`. The generic parameter on `StreamAsync<T>` and `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method. An object of type `IAsyncEnumerable<T>` or `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

A `StreamAsync` example that returns `IAsyncEnumerable<int>`:

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var stream = await hubConnection.StreamAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

await foreach (var count in stream)
{
    Console.WriteLine($"{count}");
}

Console.WriteLine("Streaming completed");
```

A corresponding `StreamAsChannelAsync` example that returns `ChannelReader<int>`:

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var channel = await hubConnection.StreamAsChannelAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

The `StreamAsChannelAsync` method on `HubConnection` is used to invoke a server-to-client streaming method. Pass

the hub method name and arguments defined in the hub method to `StreamAsChannelAsync`. The generic parameter on `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method. A `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var channel = await hubConnection.StreamAsChannelAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

The `StreamAsChannelAsync` method on `HubConnection` is used to invoke a server-to-client streaming method. Pass the hub method name and arguments defined in the hub method to `StreamAsChannelAsync`. The generic parameter on `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method. A `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

```
var channel = await hubConnection
    .StreamAsChannelAsync<int>("Counter", 10, 500, CancellationToken.None);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

Client-to-server streaming

There are two ways to invoke a client-to-server streaming hub method from the .NET client. You can either pass in an `IAsyncEnumerable<T>` or a `ChannelReader` as an argument to `SendAsync`, `InvokeAsync`, or `StreamAsChannelAsync`, depending on the hub method invoked.

Whenever data is written to the `IAsyncEnumerable` or `ChannelWriter` object, the hub method on the server receives a new item with the data from the client.

If using an `IAsyncEnumerable` object, the stream ends after the method returning stream items exits.

> **NOTE**
>
> The following sample requires C# 8.0 or later.

```
async IAsyncEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
        var data = await FetchSomeData();
        yield return data;
    }
    //After the for loop has completed and the local function exits the stream completion will be sent.
}

await connection.SendAsync("UploadStream", clientStreamData());
```

Or if you're using a `ChannelWriter`, you complete the channel with `channel.Writer.Complete()`:

```
var channel = Channel.CreateBounded<string>(10);
await connection.SendAsync("UploadStream", channel.Reader);
await channel.Writer.WriteAsync("some data");
await channel.Writer.WriteAsync("some more data");
channel.Writer.Complete();
```

JavaScript client

Server-to-client streaming

JavaScript clients call server-to-client streaming methods on hubs with `connection.stream`. The `stream` method accepts two arguments:

- The name of the hub method. In the following example, the hub method name is `Counter`.
- Arguments defined in the hub method. In the following example, the arguments are a count for the number of stream items to receive and the delay between stream items.

`connection.stream` returns an `IStreamResult`, which contains a `subscribe` method. Pass an `IStreamSubscriber` to `subscribe` and set the `next`, `error`, and `complete` callbacks to receive notifications from the `stream` invocation.

```
connection.stream("Counter", 10, 500)
    .subscribe({
        next: (item) => {
            var li = document.createElement("li");
            li.textContent = item;
            document.getElementById("messagesList").appendChild(li);
        },
        complete: () => {
            var li = document.createElement("li");
            li.textContent = "Stream completed";
            document.getElementById("messagesList").appendChild(li);
        },
        error: (err) => {
            var li = document.createElement("li");
            li.textContent = err;
            document.getElementById("messagesList").appendChild(li);
        },
});
```

To end the stream from the client, call the `dispose` method on the `ISubscription` that's returned from the `subscribe` method. Calling this method causes cancellation of the `CancellationToken` parameter of the Hub method, if you provided one.

```
connection.stream("Counter", 10, 500)
    .subscribe({
        next: (item) => {
            var li = document.createElement("li");
            li.textContent = item;
            document.getElementById("messagesList").appendChild(li);
        },
        complete: () => {
            var li = document.createElement("li");
            li.textContent = "Stream completed";
            document.getElementById("messagesList").appendChild(li);
        },
        error: (err) => {
            var li = document.createElement("li");
            li.textContent = err;
            document.getElementById("messagesList").appendChild(li);
        },
});
```

To end the stream from the client, call the `dispose` method on the `ISubscription` that's returned from the `subscribe` method.

Client-to-server streaming

JavaScript clients call client-to-server streaming methods on hubs by passing in a `Subject` as an argument to `send`, `invoke`, or `stream`, depending on the hub method invoked. The `Subject` is a class that looks like a `Subject`. For example in RxJS, you can use the Subject class from that library.

```
const subject = new signalR.Subject();
yield connection.send("UploadStream", subject);
var iteration = 0;
const intervalHandle = setInterval(() => {
    iteration++;
    subject.next(iteration.toString());
    if (iteration === 10) {
        clearInterval(intervalHandle);
        subject.complete();
    }
}, 500);
```

Calling `subject.next(item)` with an item writes the item to the stream, and the hub method receives the item on the server.

To end the stream, call `subject.complete()`.

Java client

Server-to-client streaming

The SignalR Java client uses the `stream` method to invoke streaming methods. `stream` accepts three or more arguments:

- The expected type of the stream items.
- The name of the hub method.
- Arguments defined in the hub method.

```
hubConnection.stream(String.class, "ExampleStreamingHubMethod", "Arg1")
    .subscribe(
        (item) -> {/* Define your onNext handler here. */ },
        (error) -> {/* Define your onError handler here. */},
        () -> {/* Define your onCompleted handler here. */});
```

The `stream` method on `HubConnection` returns an Observable of the stream item type. The Observable type's `subscribe` method is where `onNext`, `onError` and `onCompleted` handlers are defined.

Additional resources

- Hubs
- .NET client
- JavaScript client
- Publish to Azure

Differences between ASP.NET SignalR and ASP.NET Core SignalR

9/22/2020 • 5 minutes to read • Edit Online

ASP.NET Core SignalR isn't compatible with clients or servers for ASP.NET SignalR. This article details features which have been removed or changed in ASP.NET Core SignalR.

How to identify the SignalR version

	ASP.NET SIGNALR	ASP.NET CORE SIGNALR
Server NuGet package	Microsoft.AspNet.SignalR	None. Included in the Microsoft.AspNetCore.App shared framework.
Client NuGet packages	Microsoft.AspNet.SignalR.Client Microsoft.AspNet.SignalR.JS	Microsoft.AspNetCore.SignalR.Client
JavaScript client npm package	signalr	`@microsoft/signalr`
Java client	GitHub Repository (deprecated)	Maven package com.microsoft.signalr
Server app type	ASP.NET (System.Web) or OWIN Self-Host	ASP.NET Core
Supported server platforms	.NET Framework 4.5 or later	.NET Core 3.0 or later

	ASP.NET SIGNALR	ASP.NET CORE SIGNALR
Server NuGet package	Microsoft.AspNet.SignalR	Microsoft.AspNetCore.App (.NET Core) Microsoft.AspNetCore.SignalR (.NET Framework)
Client NuGet packages	Microsoft.AspNet.SignalR.Client Microsoft.AspNet.SignalR.JS	Microsoft.AspNetCore.SignalR.Client
JavaScript client npm package	signalr	`@aspnet/signalr`
Java client	GitHub Repository (deprecated)	Maven package com.microsoft.signalr
Server app type	ASP.NET (System.Web) or OWIN Self-Host	ASP.NET Core
Supported server platforms	.NET Framework 4.5 or later	.NET Framework 4.6.1 or later .NET Core 2.1 or later

Feature differences

Automatic reconnects

In ASP.NET SignalR:

- By default, SignalR attempts to reconnect to the server if the connection is dropped.

In ASP.NET Core SignalR:

- Automatic reconnects are opt-in with both the .NET client and the JavaScript client:

```
HubConnection connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect()
    .Build();
```

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

Prior to ASP.NET Core 3.0, SignalR doesn't support automatic reconnects. If the client is disconnected, the user must explicitly start a new connection to reconnect. In ASP.NET SignalR, SignalR attempts to reconnect to the server if the connection is dropped.

Protocol support

ASP.NET Core SignalR supports JSON, as well as a new binary protocol based on MessagePack. Additionally, custom protocols can be created.

Transports

The Forever Frame transport isn't supported in ASP.NET Core SignalR.

Differences on the server

The ASP.NET Core SignalR server-side libraries are included in Microsoft.AspNetCore.App, which is used in the **ASP.NET Core Web Application** template for both Razor and MVC projects.

ASP.NET Core SignalR is an ASP.NET Core middleware. It must be configured by calling AddSignalR in `Startup.ConfigureServices`.

```
services.AddSignalR()
```

To configure routing, map routes to hubs inside the UseEndpoints method call in the `Startup.Configure` method.

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>("/hub");
});
```

To configure routing, map routes to hubs inside the UseSignalR method call in the `Startup.Configure` method.

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("/hub");
});
```

Sticky sessions

The scaleout model for ASP.NET SignalR allows clients to reconnect and send messages to any server in the farm. In ASP.NET Core SignalR, the client must interact with the same server for the duration of the connection. For scaleout using Redis, that means sticky sessions are required. For scaleout using Azure SignalR Service, sticky sessions aren't required because the service handles connections to clients.

Single hub per connection

In ASP.NET Core SignalR, the connection model has been simplified. Connections are made directly to a single hub, rather than a single connection being used to share access to multiple hubs.

Streaming

ASP.NET Core SignalR now supports streaming data from the hub to the client.

State

The ability to pass arbitrary state between clients and the hub (often called `HubState`) has been removed, as well as support for progress messages. There is no counterpart of hub proxies at the moment.

PersistentConnection removal

In ASP.NET Core SignalR, the PersistentConnection class has been removed.

GlobalHost

ASP.NET Core has dependency injection (DI) built into the framework. Services can use DI to access the HubContext. The `GlobalHost` object that is used in ASP.NET SignalR to get a `HubContext` doesn't exist in ASP.NET Core SignalR.

HubPipeline

ASP.NET Core SignalR doesn't have support for `HubPipeline` modules.

Differences on the client

TypeScript

The ASP.NET Core SignalR client is written in TypeScript. You can write in JavaScript or TypeScript when using the JavaScript client.

The JavaScript client is hosted at npm

In ASP.NET versions, the JavaScript client was obtained through a NuGet package in Visual Studio. In the ASP.NET Core versions, the `@microsoft/signalr` npm package contains the JavaScript libraries. This package isn't included in the **ASP.NET Core Web Application** template. Use npm to obtain and install the `@microsoft/signalr` npm package.

```
npm init -y
npm install @microsoft/signalr
```

In ASP.NET versions, the JavaScript client was obtained through a NuGet package in Visual Studio. In the ASP.NET Core versions, the `@aspnet/signalr` npm package contains the JavaScript libraries. This package isn't included in the **ASP.NET Core Web Application** template. Use npm to obtain and install the `@aspnet/signalr` npm package.

```
npm init -y
npm install @aspnet/signalr
```

jQuery

The dependency on jQuery has been removed, however projects can still use jQuery.

Internet Explorer support

ASP.NET Core SignalR requires Microsoft Internet Explorer 11 or later (ASP.NET SignalR supported Microsoft Internet Explorer 8 and later).

JavaScript client method syntax

The JavaScript syntax has changed from the ASP.NET version of SignalR. Rather than using the `$connection` object, create a connection using the HubConnectionBuilder API.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();
```

Use the on method to specify client methods that the hub can call.

The JavaScript syntax has changed from the ASP.NET version of SignalR. Rather than using the `$connection` object, create a connection using the HubConnectionBuilder API.

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();
```

Use the on method to specify client methods that the hub can call.

```
connection.on("ReceiveMessage", (user, message) => {
    const msg = message.replace(/&/g, "&amp;").replace(/</g, "&lt;").replace(/>/g, "&gt;");
    const encodedMsg = `${user} says ${msg}`;
    console.log(encodedMsg);
});
```

After creating the client method, start the hub connection. Chain a catch method to log or handle errors.

```
connection.start().catch(err => console.error(err));
```

Hub proxies

Hub proxies are no longer automatically generated. Instead, the method name is passed into the invoke API as a string.

Hub proxies are no longer automatically generated. Instead, the method name is passed into the invoke API as a string.

.NET and other clients

The Microsoft.AspNetCore.SignalR.Client NuGet package contains the .NET client libraries for ASP.NET Core SignalR.

Use the HubConnectionBuilder to create and build an instance of a connection to a hub.

```
connection = new HubConnectionBuilder()
    .WithUrl("url")
    .Build();
```

Scaleout differences

ASP.NET SignalR supports SQL Server and Redis. ASP.NET Core SignalR supports Azure SignalR Service and Redis.

ASP.NET

- SignalR scaleout with Azure Service Bus

- SignalR scaleout with Redis
- SignalR scaleout with SQL Server

ASP.NET Core

- Azure SignalR Service
- Redis Backplane

Additional resources

- Hubs
- JavaScript client
- .NET client
- Supported platforms

WebSockets support in ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

By Tom Dykstra and Andrew Stanton-Nurse

This article explains how to get started with WebSockets in ASP.NET Core. WebSocket (RFC 6455) is a protocol that enables two-way persistent communication channels over TCP connections. It's used in apps that benefit from fast, real-time communication, such as chat, dashboard, and game apps.

View or download sample code (how to download). How to run.

SignalR

ASP.NET Core SignalR is a library that simplifies adding real-time web functionality to apps. It uses WebSockets whenever possible.

For most applications, we recommend SignalR over raw WebSockets. SignalR provides transport fallback for environments where WebSockets is not available. It also provides a simple remote procedure call app model. And in most scenarios, SignalR has no significant performance disadvantage compared to using raw WebSockets.

Prerequisites

- ASP.NET Core 1.1 or later
- Any OS that supports ASP.NET Core:
  - Windows 7 / Windows Server 2008 or later
  - Linux
  - macOS
- If the app runs on Windows with IIS:
  - Windows 8 / Windows Server 2012 or later
  - IIS 8 / IIS 8 Express
  - WebSockets must be enabled (See the IIS/IIS Express support section.).
- If the app runs on HTTP.sys:
  - Windows 8 / Windows Server 2012 or later
- For supported browsers, see https://caniuse.com/#feat=websockets.

NuGet package

Install the Microsoft.AspNetCore.WebSockets package.

Configure the middleware

Add the WebSockets middleware in the `Configure` method of the `Startup` class:

```
app.UseWebSockets();
```

The following settings can be configured:

- `KeepAliveInterval` - How frequently to send "ping" frames to the client to ensure proxies keep the connection open. The default is two minutes.
- `ReceiveBufferSize` - The size of the buffer used to receive data. Advanced users may need to change this for performance tuning based on the size of the data. The default is 4 KB.

The following settings can be configured:

- `KeepAliveInterval` - How frequently to send "ping" frames to the client to ensure proxies keep the connection open. The default is two minutes.
- ReceiveBufferSize - The size of the buffer used to receive data. Advanced users may need to change this for performance tuning based on the size of the data. The default is 4 KB.
- `AllowedOrigins` - A list of allowed Origin header values for WebSocket requests. By default, all origins are allowed. See "WebSocket origin restriction" below for details.

```
var webSocketOptions = new WebSocketOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(120),
    ReceiveBufferSize = 4 * 1024
};

app.UseWebSockets(webSocketOptions);
```

Accept WebSocket requests

Somewhere later in the request life cycle (later in the `Configure` method or in an action method, for example) check if it's a WebSocket request and accept the WebSocket request.

The following example is from later in the `Configure` method:

```
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
            await Echo(context, webSocket);
        }
        else
        {
            context.Response.StatusCode = 400;
        }
    }
    else
    {
        await next();
    }

});
```

A WebSocket request could come in on any URL, but this sample code only accepts requests for `/ws`.

When using a WebSocket, you **must** keep the middleware pipeline running for the duration of the connection. If you attempt to send or receive a WebSocket message after the middleware pipeline ends, you may get an exception like the following:

```
System.Net.WebSockets.WebSocketException (0x80004005): The remote party closed the WebSocket connection
without completing the close handshake. ---> System.ObjectDisposedException: Cannot write to the response
body, the response has completed.
Object name: 'HttpResponseStream'.
```

If you're using a background service to write data to a WebSocket, make sure you keep the middleware pipeline running. Do this by using a TaskCompletionSource<TResult>. Pass the `TaskCompletionSource` to your background service and have it call TrySetResult when you finish with the WebSocket. Then `await` the Task property during the request, as shown in the following example:

```
app.Use(async (context, next) => {
    var socket = await context.WebSockets.AcceptWebSocketAsync();
    var socketFinishedTcs = new TaskCompletionSource<object>();

    BackgroundSocketProcessor.AddSocket(socket, socketFinishedTcs);

    await socketFinishedTcs.Task;
});
```

The WebSocket closed exception can also happen if you return too soon from an action method. If you accept a socket in an action method, wait for the code that uses the socket to complete before returning from the action method.

Never use `Task.Wait()`, `Task.Result`, or similar blocking calls to wait for the socket to complete, as that can cause serious threading issues. Always use `await`.

Send and receive messages

The `AcceptWebSocketAsync` method upgrades the TCP connection to a WebSocket connection and provides a WebSocket object. Use the `WebSocket` object to send and receive messages.

The code shown earlier that accepts the WebSocket request passes the `WebSocket` object to an `Echo` method. The code receives a message and immediately sends back the same message. Messages are sent and received in a loop until the client closes the connection:

```
private async Task Echo(HttpContext context, WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
CancellationToken.None);
    while (!result.CloseStatus.HasValue)
    {
        await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, result.Count), result.MessageType,
result.EndOfMessage, CancellationToken.None);

        result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);
    }
    await webSocket.CloseAsync(result.CloseStatus.Value, result.CloseStatusDescription,
CancellationToken.None);
}
```

When accepting the WebSocket connection before beginning the loop, the middleware pipeline ends. Upon closing the socket, the pipeline unwinds. That is, the request stops moving forward in the pipeline when the WebSocket is accepted. When the loop is finished and the socket is closed, the request proceeds back up the pipeline.

Handle client disconnects

The server is not automatically informed when the client disconnects due to loss of connectivity. The server receives a disconnect message only if the client sends it, which can't be done if the internet connection is lost. If you want to take some action when that happens, set a timeout after nothing is received from the client within a certain time window.

If the client isn't always sending messages and you don't want to timeout just because the connection goes idle, have the client use a timer to send a ping message every X seconds. On the server, if a message hasn't arrived within 2*X seconds after the previous one, terminate the connection and report that the client disconnected. Wait for twice the expected time interval to leave extra time for network delays that might hold up the ping message.

WebSocket origin restriction

The protections provided by CORS don't apply to WebSockets. Browsers do **not**:

- Perform CORS pre-flight requests.
- Respect the restrictions specified in `Access-Control` headers when making WebSocket requests.

However, browsers do send the `Origin` header when issuing WebSocket requests. Applications should be configured to validate these headers to ensure that only WebSockets coming from the expected origins are allowed.

If you're hosting your server on "https://server.com" and hosting your client on "https://client.com", add "https://client.com" to the `AllowedOrigins` list for WebSockets to verify.

```
var webSocketOptions = new WebSocketOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(120),
    ReceiveBufferSize = 4 * 1024
};
webSocketOptions.AllowedOrigins.Add("https://client.com");
webSocketOptions.AllowedOrigins.Add("https://www.client.com");

app.UseWebSockets(webSocketOptions);
```

> **NOTE**
>
> The `Origin` header is controlled by the client and, like the `Referer` header, can be faked. Do **not** use these headers as an authentication mechanism.

IIS/IIS Express support

Windows Server 2012 or later and Windows 8 or later with IIS/IIS Express 8 or later has support for the WebSocket protocol.

> **NOTE**
>
> WebSockets are always enabled when using IIS Express.

Enabling WebSockets on IIS

To enable support for the WebSocket protocol on Windows Server 2012 or later:

> **NOTE**
> These steps are not required when using IIS Express

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**.
2. Select **Role-based or Feature-based Installation**. Select **Next**.
3. Select the appropriate server (the local server is selected by default). Select **Next**.
4. Expand **Web Server (IIS)** in the **Roles** tree, expand **Web Server**, and then expand **Application Development**.
5. Select **WebSocket Protocol**. Select **Next**.
6. If additional features aren't needed, select **Next**.
7. Select **Install**.
8. When the installation completes, select **Close** to exit the wizard.

To enable support for the WebSocket protocol on Windows 8 or later:

> **NOTE**
> These steps are not required when using IIS Express

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the following nodes: **Internet Information Services** > **World Wide Web Services** > **Application Development Features**.
3. Select the **WebSocket Protocol** feature. Select **OK**.

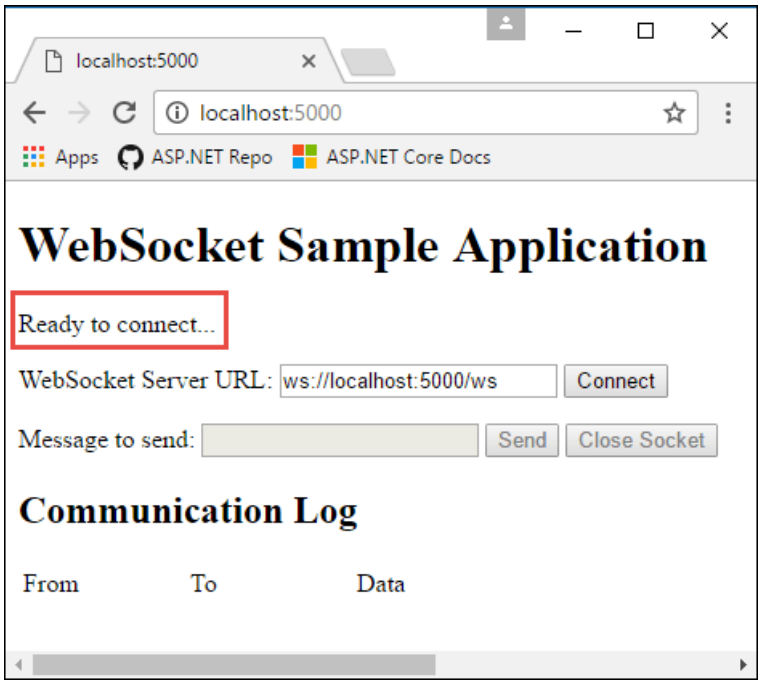
Disable WebSocket when using socket.io on Node.js

If using the WebSocket support in socket.io on Node.js, disable the default IIS WebSocket module using the `webSocket` element in *web.config* or *applicationHost.config*. If this step isn't performed, the IIS WebSocket module attempts to handle the WebSocket communication rather than Node.js and the app.

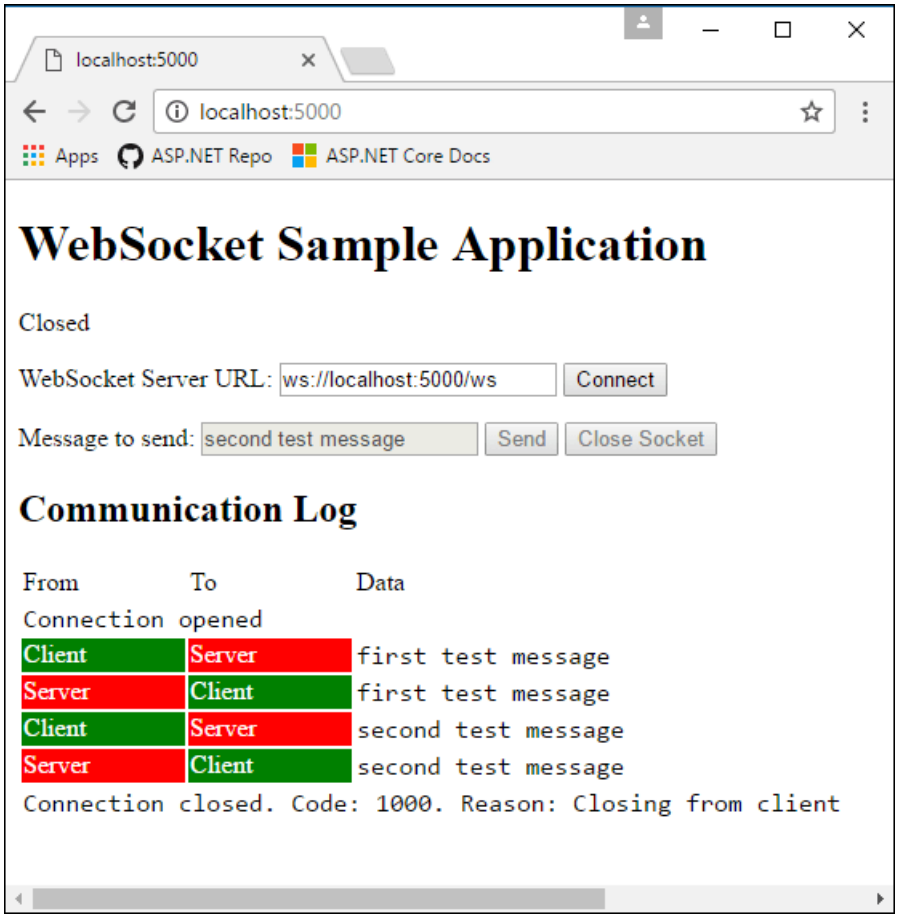
```
<system.webServer>
  <webSocket enabled="false" />
</system.webServer>
```

Sample app

The sample app that accompanies this article is an echo app. It has a web page that makes WebSocket connections, and the server resends any messages it receives back to the client. Run the app from a command prompt (it's not set up to run from Visual Studio with IIS Express) and navigate to http://localhost:5000. The web page shows the connection status in the upper left:



Select **Connect** to send a WebSocket request to the URL shown. Enter a test message and select **Send**. When done, select **Close Socket**. The **Communication Log** section reports each open, send, and close action as it happens.



Logging and diagnostics in ASP.NET Core SignalR

9/22/2020 • 8 minutes to read • Edit Online

By Andrew Stanton-Nurse

This article provides guidance for gathering diagnostics from your ASP.NET Core SignalR app to help troubleshoot issues.

Server-side logging

> **WARNING**
> Server-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

Since SignalR is part of ASP.NET Core, it uses the ASP.NET Core logging system. In the default configuration, SignalR logs very little information, but this can configured. See the documentation on ASP.NET Core logging for details on configuring ASP.NET Core logging.

SignalR uses two logger categories:

- `Microsoft.AspNetCore.SignalR`: For logs related to Hub Protocols, activating Hubs, invoking methods, and other Hub-related activities.
- `Microsoft.AspNetCore.Http.Connections`: For logs related to transports, such as WebSockets, Long Polling, Server-Sent Events, and low-level SignalR infrastructure.

To enable detailed logs from SignalR, configure both of the preceding prefixes to the `Debug` level in your *appsettings.json* file by adding the following items to the `LogLevel` sub-section in `Logging`:

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Debug",
            "System": "Information",
            "Microsoft": "Information",
            "Microsoft.AspNetCore.SignalR": "Debug",
            "Microsoft.AspNetCore.Http.Connections": "Debug"
        }
    }
}
```

You can also configure this in code in your `CreateWebHostBuilder` method:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.AddFilter("Microsoft.AspNetCore.SignalR", LogLevel.Debug);
            logging.AddFilter("Microsoft.AspNetCore.Http.Connections", LogLevel.Debug);
        })
        .UseStartup<Startup>();
```

If you aren't using JSON-based configuration, set the following configuration values in your configuration system:

- `Logging:LogLevel:Microsoft.AspNetCore.SignalR` = `Debug`
- `Logging:LogLevel:Microsoft.AspNetCore.Http.Connections` = `Debug`

Check the documentation for your configuration system to determine how to specify nested configuration values. For example, when using environment variables, two `_` characters are used instead of the `:` (for example, `Logging__LogLevel__Microsoft.AspNetCore.SignalR`).

We recommend using the `Debug` level when gathering more detailed diagnostics for your app. The `Trace` level produces very low-level diagnostics and is rarely needed to diagnose issues in your app.

Access server-side logs

How you access server-side logs depends on the environment in which you're running.

As a console app outside IIS

If you're running in a console app, the Console logger should be enabled by default. SignalR logs will appear in the console.

Within IIS Express from Visual Studio

Visual Studio displays the log output in the **Output** window. Select the **ASP.NET Core Web Server** drop down option.

Azure App Service

Enable the **Application Logging (Filesystem)** option in the **Diagnostics logs** section of the Azure App Service portal and configure the **Level** to `Verbose`. Logs should be available from the **Log streaming** service and in logs on the file system of the App Service. For more information, see Azure log streaming.

Other environments

If the app is deployed to another environment (for example, Docker, Kubernetes, or Windows Service), see Logging in .NET Core and ASP.NET Core for more information on how to configure logging providers suitable for the environment.

JavaScript client logging

> **WARNING**
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

When using the JavaScript client, you can configure logging options using the `configureLogging` method on `HubConnectionBuilder`:

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/my/hub/url")
    .configureLogging(signalR.LogLevel.Debug)
    .build();
```

To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

The following table shows log levels available to the JavaScript client. Setting the log level to one of these values enables logging at that level and all levels above it in the table.

LEVEL	DESCRIPTION
`None`	No messages are logged.
`Critical`	Messages that indicate a failure in the entire app.
`Error`	Messages that indicate a failure in the current operation.
`Warning`	Messages that indicate a non-fatal problem.
`Information`	Informational messages.
`Debug`	Diagnostic messages useful for debugging.
`Trace`	Very detailed diagnostic messages designed for diagnosing specific issues.

Once you've configured the verbosity, the logs will be written to the Browser Console (or Standard Output in a NodeJS app).

If you want to send logs to a custom logging system, you can provide a JavaScript object implementing the `ILogger` interface. The only method that needs to be implemented is `log`, which takes the level of the event and the message associated with the event. For example:

```
import { ILogger, LogLevel, HubConnectionBuilder } from "@aspnet/signalr";

export class MyLogger implements ILogger {
    log(logLevel: LogLevel, message: string) {
        // Use `message` and `logLevel` to record the log message to your own system
    }
}

// later on, when configuring your connection...

let connection = new HubConnectionBuilder()
    .withUrl("/my/hub/url")
    .configureLogging(new MyLogger())
    .build();
```

.NET client logging

> **WARNING**
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

To get logs from the .NET client, you can use the `ConfigureLogging` method on `HubConnectionBuilder`. This works the same way as the `ConfigureLogging` method on `WebHostBuilder` and `HostBuilder`. You can configure the same logging providers you use in ASP.NET Core. However, you have to manually install and enable the NuGet packages for the individual logging providers.

To add .NET client logging to a Blazor WebAssembly app, see ASP.NET Core Blazor logging.

Console logging

In order to enable Console logging, add the Microsoft.Extensions.Logging.Console package. Then, use the

`AddConsole` method to configure the console logger:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Log to the Console
        logging.AddConsole();

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug);
    })
    .Build();
```

Debug output window logging

You can also configure logs to go to the **Output** window in Visual Studio. Install the Microsoft.Extensions.Logging.Debug package and use the `AddDebug` method:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Log to the Output Window
        logging.AddDebug();

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug)
    })
    .Build();
```

Other logging providers

SignalR supports other logging providers such as Serilog, Seq, NLog, or any other logging system that integrates with `Microsoft.Extensions.Logging`. If your logging system provides an `ILoggerProvider`, you can register it with `AddProvider`:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Log to your custom provider
        logging.AddProvider(new MyCustomLoggingProvider());

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug)
    })
    .Build();
```

Control verbosity

If you are logging from other places in your app, changing the default level to `Debug` may be too verbose. You can use a Filter to configure the logging level for SignalR logs. This can be done in code, in much the same way as on the server:

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Register your providers

        // Set the default log level to Information, but to Debug for SignalR-related loggers.
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddFilter("Microsoft.AspNetCore.SignalR", LogLevel.Debug);
        logging.AddFilter("Microsoft.AspNetCore.Http.Connections", LogLevel.Debug);
    })
    .Build();
```

Network traces

> **WARNING**
> A network trace contains the full contents of every message sent by your app. **Never** post raw network traces from production apps to public forums like GitHub.

If you encounter an issue, a network trace can sometimes provide a lot of helpful information. This is particularly useful if you're going to file an issue on our issue tracker.

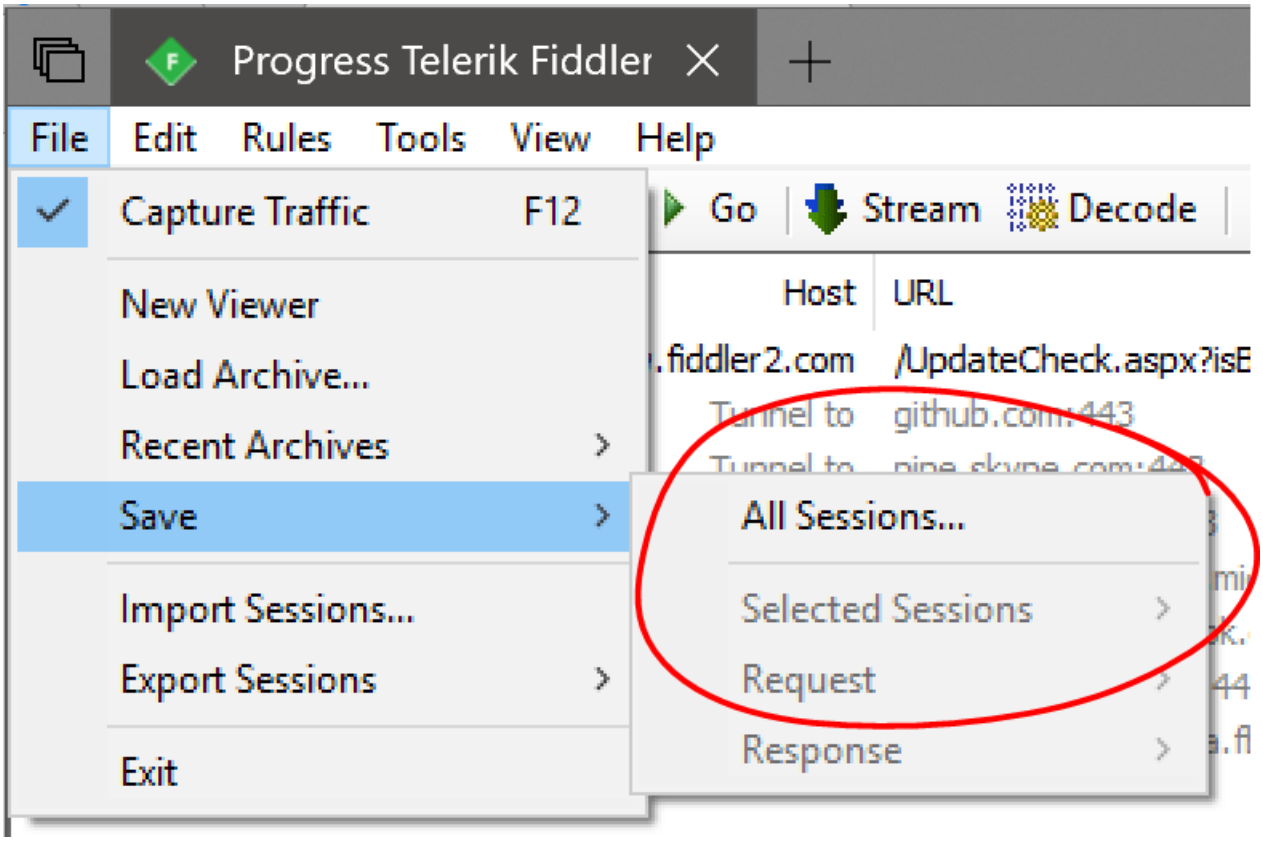
Collect a network trace with Fiddler (preferred option)

This method works for all apps.

Fiddler is a very powerful tool for collecting HTTP traces. Install it from telerik.com/fiddler, launch it, and then run your app and reproduce the issue. Fiddler is available for Windows, and there are beta versions for macOS and Linux.

If you connect using HTTPS, there are some extra steps to ensure Fiddler can decrypt the HTTPS traffic. For more details, see the Fiddler documentation.

Once you've collected the trace, you can export the trace by choosing **File** > **Save** > **All Sessions** from the menu bar.



Collect a network trace with tcpdump (macOS and Linux only)

This method works for all apps.

You can collect raw TCP traces using tcpdump by running the following command from a command shell. You may need to be `root` or prefix the command with `sudo` if you get a permissions error:

```
tcpdump -i [interface] -w trace.pcap
```

Replace `[interface]` with the network interface you wish to capture on. Usually, this is something like `/dev/eth0` (for your standard Ethernet interface) or `/dev/lo0` (for localhost traffic). For more information, see the `tcpdump` man page on your host system.

Collect a network trace in the browser

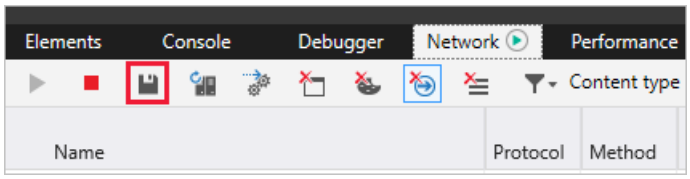
This method only works for browser-based apps.

Most browser Developer Tools have a "Network" tab that allows you to capture network activity between the browser and the server. However, these traces don't include WebSocket and Server-Sent Event messages. If you are using those transports, using a tool like Fiddler or TcpDump (described below) is a better approach.

Microsoft Edge and Internet Explorer

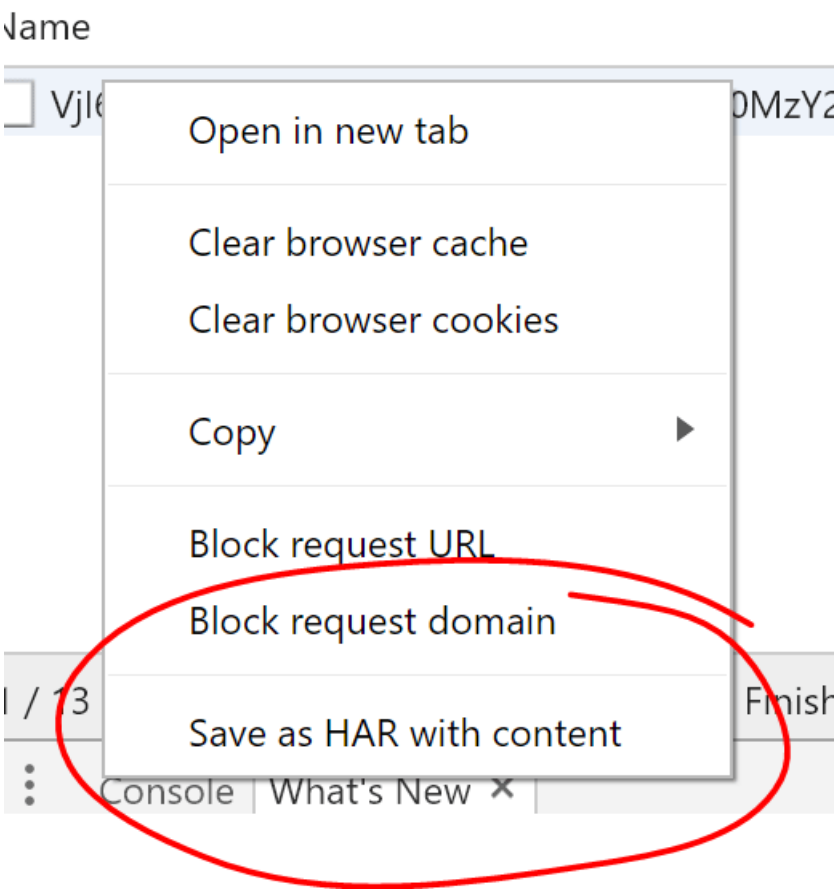
(The instructions are the same for both Edge and Internet Explorer)

1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Click the Save icon in the toolbar to export the trace as a "HAR" file:



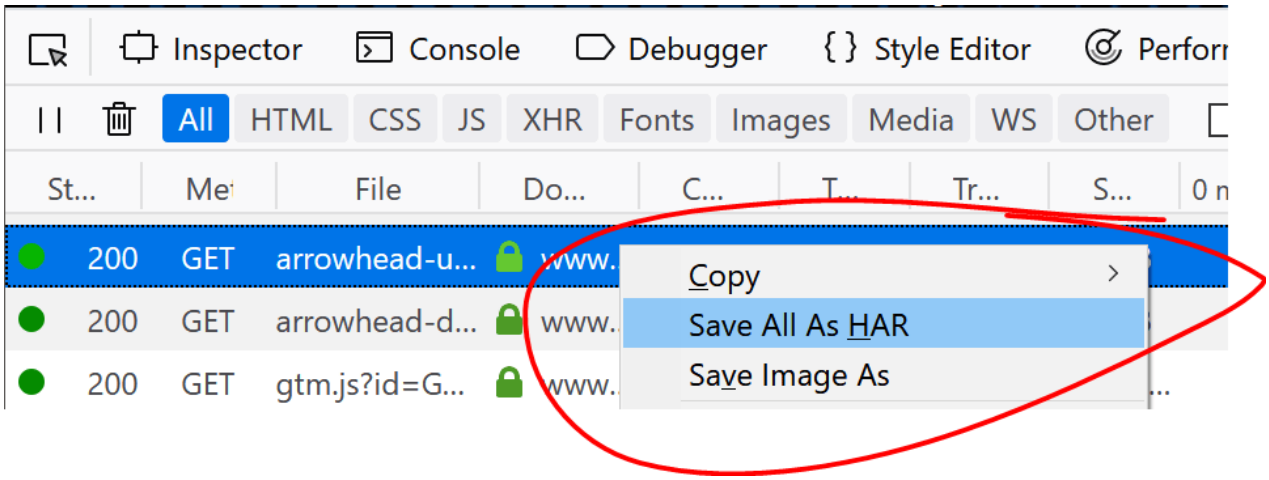
Google Chrome

1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Right click anywhere in the list of requests and choose "Save as HAR with content":



Mozilla Firefox

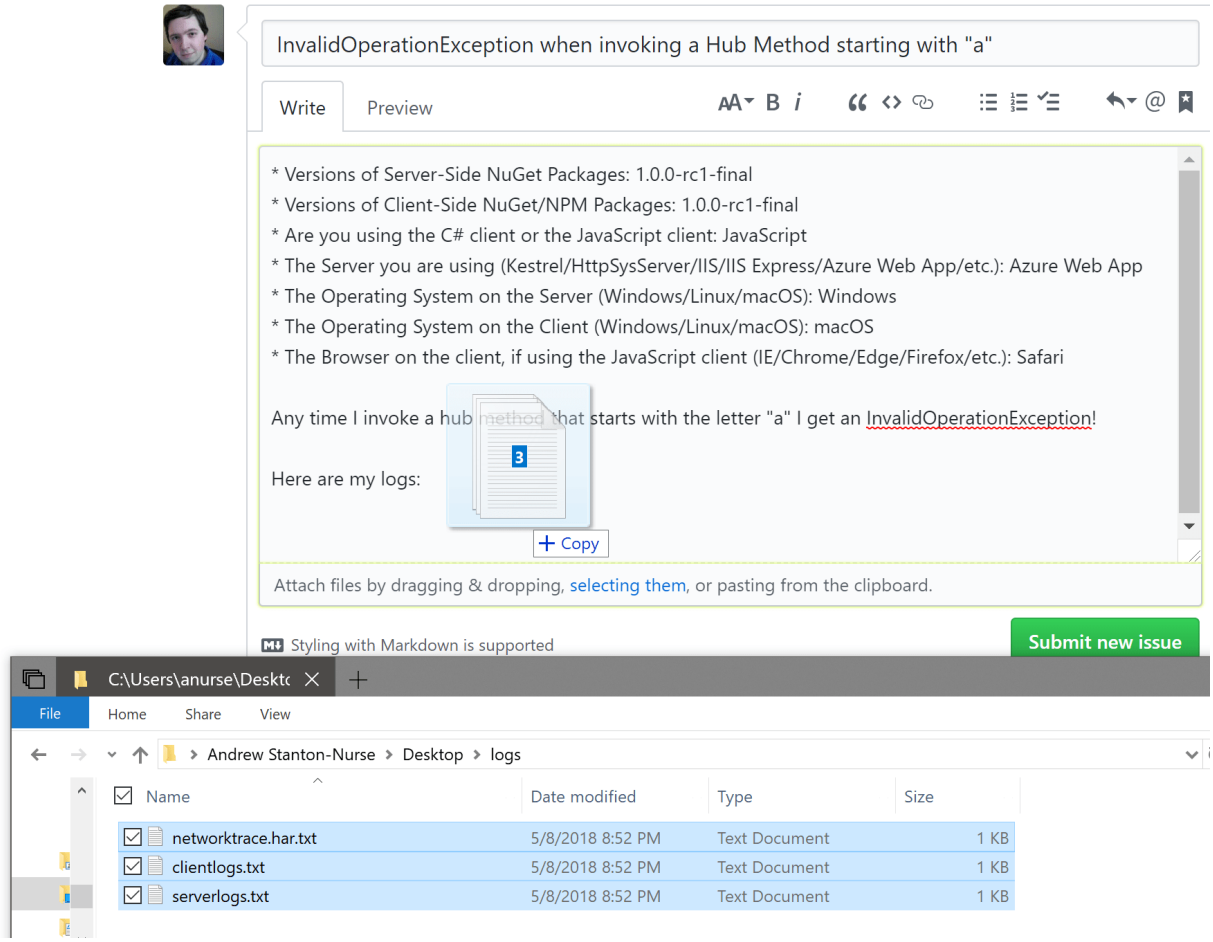
1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Right click anywhere in the list of requests and choose "Save All As HAR"



Attach diagnostics files to GitHub issues

You can attach Diagnostics files to GitHub issues by renaming them so they have a `.txt` extension and then dragging and dropping them on to the issue.

> **NOTE**
> Please don't paste the content of log files or network traces into a GitHub issue. These logs and traces can be quite large, and GitHub usually truncates them.



Metrics

Metrics is a representation of data measures over intervals of time. For example, requests per second. Metrics data allows observation of the state of an app at a high level. .NET gRPC metrics are emitted using EventCounter.

SignalR server metrics

SignalR server metrics are reported on the Microsoft.AspNetCore.Http.Connections event source.

NAME	DESCRIPTION
`connections-started`	Total connections started
`connections-stopped`	Total connections stopped
`connections-timed-out`	Total connections timed out
`current-connections`	Current connections
`connections-duration`	Average connection duration

Observe metrics

dotnet-counters is a performance monitoring tool for ad-hoc health monitoring and first-level performance investigation. Monitor a .NET app with `Microsoft.AspNetCore.Http.Connections` as the provider name. For example:

```
> dotnet-counters monitor --process-id 37016 Microsoft.AspNetCore.Http.Connections

Press p to pause, r to resume, q to quit.
    Status: Running
[Microsoft.AspNetCore.Http.Connections]
    Average Connection Duration (ms)       16,040.56
    Current Connections                            1
    Total Connections Started                      8
    Total Connections Stopped                      7
    Total Connections Timed Out                    0
```

Additional resources

- ASP.NET Core SignalR configuration
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client

Introduction to gRPC on .NET Core

9/22/2020 • 2 minutes to read • Edit Online

By John Luo and James Newton-King

gRPC is a language agnostic, high-performance Remote Procedure Call (RPC) framework.

The main benefits of gRPC are:

- Modern, high-performance, lightweight RPC framework.
- Contract-first API development, using Protocol Buffers by default, allowing for language agnostic implementations.
- Tooling available for many languages to generate strongly-typed servers and clients.
- Supports client, server, and bi-directional streaming calls.
- Reduced network usage with Protobuf binary serialization.

These benefits make gRPC ideal for:

- Lightweight microservices where efficiency is critical.
- Polyglot systems where multiple languages are required for development.
- Point-to-point real-time services that need to handle streaming requests or responses.

> **WARNING**
>
> ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this GitHub issue.

C# Tooling support for .proto files

gRPC uses a contract-first approach to API development. Services and messages are defined in **.proto* files:

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

.NET types for services, clients and messages are automatically generated by including **.proto* files in a project:

- Add a package reference to Grpc.Tools package.
- Add **.proto* files to the `<Protobuf>` item group.

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" />
</ItemGroup>
```

For more information on gRPC tooling support, see gRPC services with C#.

gRPC services on ASP.NET Core

gRPC services can be hosted on ASP.NET Core. Services have full integration with popular ASP.NET Core features such as logging, dependency injection (DI), authentication and authorization.

The gRPC service project template provides a starter service:

```
public class GreeterService : Greeter.GreeterBase
{
    private readonly ILogger<GreeterService> _logger;

    public GreeterService(ILogger<GreeterService> logger)
    {
        _logger = logger;
    }

    public override Task<HelloReply> SayHello(HelloRequest request,
        ServerCallContext context)
    {
        _logger.LogInformation("Saying hello to {Name}", request.Name);
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

`GreeterService` inherits from the `GreeterBase` type, which is generated from the `Greeter` service in the **.proto* file. The service is made accessible to clients in *Startup.cs*:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGrpcService<GreeterService>();
});
```

To learn more about gRPC services on ASP.NET Core, see gRPC services with ASP.NET Core.

Call gRPC services with a .NET client

gRPC clients are concrete client types that are generated from **.proto* files. The concrete gRPC client has methods that translate to the gRPC service in the **.proto* file.

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);

var response = await client.SayHelloAsync(
    new HelloRequest { Name = "World" });

Console.WriteLine(response.Message);
```

A gRPC client is created using a channel, which represents a long-lived connection to a gRPC service. A channel can be created using `GrpcChannel.ForAddress`.

For more information on creating clients, and calling different service methods, see Call gRPC services with the .NET client.

Additional resources

- gRPC services with C#
- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client
- gRPC client factory integration in .NET Core
- Create a .NET Core gRPC client and server in ASP.NET Core

Tutorial: Create a gRPC client and server in ASP.NET Core

9/22/2020 • 9 minutes to read • Edit Online

By John Luo

This tutorial shows how to create a .NET Core gRPC client and an ASP.NET Core gRPC Server.

At the end, you'll have a gRPC client that communicates with the gRPC Greeter service.

View or download sample code (how to download).

In this tutorial, you:

- Create a gRPC Server.
- Create a gRPC client.
- Test the gRPC client service with the gRPC Greeter service.

Prerequisites

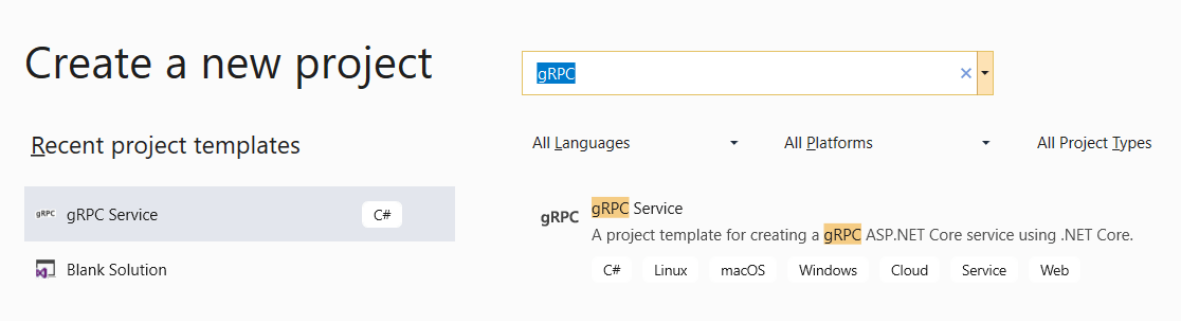
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create a gRPC service

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Start Visual Studio and select **Create a new project**. Alternatively, from the Visual Studio **File** menu, select **New** > **Project**.
- In the **Create a new project** dialog, select **gRPC Service** and select **Next**:



- Name the project **GrpcGreeter**. It's important to name the project *GrpcGreeter* so the namespaces will match when you copy and paste code.
- Select **Create**.

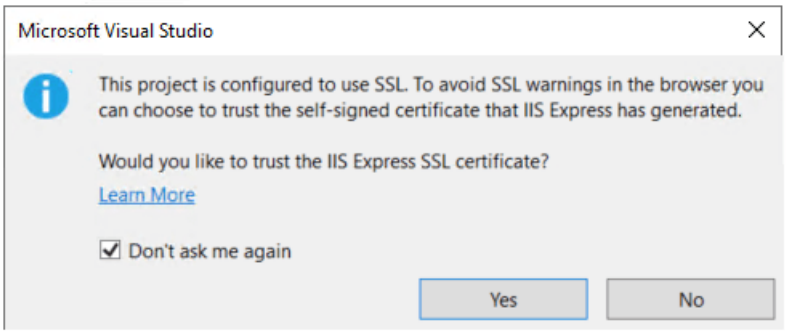
- In the **Create a new gRPC service** dialog:
  - The **gRPC Service** template is selected.
  - Select **Create**.

Run the service

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

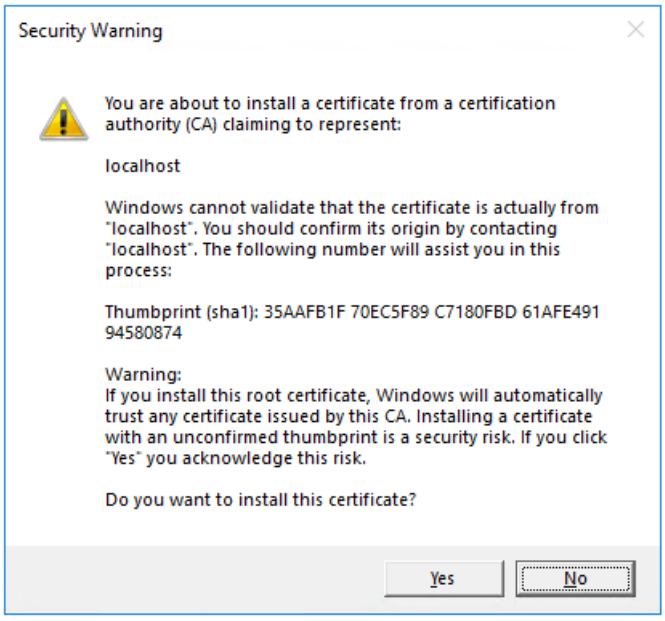
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog:



  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:



  Select **Yes** if you agree to trust the development certificate.

  Visual Studio starts IIS Express and runs the app. The address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for the local computer. Localhost only serves web requests from the local computer. When Visual Studio creates a web project, a random port is used for the web server.

The logs show the service listening on `https://localhost:5001`.

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

> **NOTE**
>
> The gRPC template is configured to use Transport Layer Security (TLS). gRPC clients need to use HTTPS to call the server.
>
> macOS doesn't support ASP.NET Core gRPC with TLS. Additional configuration is required to successfully run gRPC services on macOS. For more information, see Unable to start ASP.NET Core gRPC app on macOS.

Examine the project files

GrpcGreeter project files:

- *greet.proto*: The *Protos/greet.proto* file defines the `Greeter` gRPC and is used to generate the gRPC server assets. For more information, see Introduction to gRPC.
- *Services* folder: Contains the implementation of the `Greeter` service.
- *appSettings.json*: Contains configuration data, such as protocol used by Kestrel. For more information, see Configuration in ASP.NET Core.
- *Program.cs*: Contains the entry point for the gRPC service. For more information, see .NET Generic Host.
- *Startup.cs*: Contains code that configures app behavior. For more information, see App startup.

Create the gRPC client in a .NET console app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Open a second instance of Visual Studio and select **Create a new project**.
- In the **Create a new project** dialog, select **Console App (.NET Core)** and select **Next**.
- In the **Project name** text box, enter **GrpcGreeterClient** and select **Create**.

Add required packages

The gRPC client project requires the following packages:

- Grpc.Net.Client, which contains the .NET Core client.
- Google.Protobuf, which contains protobuf message APIs for C#.
- Grpc.Tools, which contains C# tooling support for protobuf files. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Install the packages using either the Package Manager Console (PMC) or Manage NuGet Packages.

PMC option to install packages

- From Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**
- From the **Package Manager Console** window, run `cd GrpcGreeterClient` to change directories to the folder containing the *GrpcGreeterClient.csproj* files.

- Run the following commands:

```
Install-Package Grpc.Net.Client
Install-Package Google.Protobuf
Install-Package Grpc.Tools
```

Manage NuGet Packages option to install packages

- Right-click the project in **Solution Explorer** > **Manage NuGet Packages**
- Select the **Browse** tab.
- Enter **Grpc.Net.Client** in the search box.
- Select the **Grpc.Net.Client** package from the **Browse** tab and select **Install**.
- Repeat for `Google.Protobuf` and `Grpc.Tools`.

Add greet.proto

- Create a *Protos* folder in the gRPC client project.
- Copy the *Protos\greet.proto* file from the gRPC Greeter service to the gRPC client project.
- Edit the *GrpcGreeterClient.csproj* project file:
  - Visual Studio
  - Visual Studio Code
  - Visual Studio for Mac

  Right-click the project and select **Edit Project File**.

- Add an item group with a `<Protobuf>` element that refers to the *greet.proto* file:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Create the Greeter client

Build the project to create the types in the `GrpcGreeter` namespace. The `GrpcGreeter` types are generated automatically by the build process.

Update the gRPC client *Program.cs* file with the following code:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using GrpcGreeter;
using Grpc.Net.Client;

namespace GrpcGreeterClient
{
    class Program
    {
        static async Task Main(string[] args)
        {
            // The port number(5001) must match the port of the gRPC server.
            using var channel = GrpcChannel.ForAddress("https://localhost:5001");
            var client =  new Greeter.GreeterClient(channel);
            var reply = await client.SayHelloAsync(
                              new HelloRequest { Name = "GreeterClient" });
            Console.WriteLine("Greeting: " + reply.Message);
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

Program.cs contains the entry point and logic for the gRPC client.

The Greeter client is created by:

- Instantiating a `GrpcChannel` containing the information for creating the connection to the gRPC service.
- Using the `GrpcChannel` to construct the Greeter client:

```
static async Task Main(string[] args)
{
    // The port number(5001) must match the port of the gRPC server.
    using var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client =  new Greeter.GreeterClient(channel);
    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
}
```

The Greeter client calls the asynchronous `SayHello` method. The result of the `SayHello` call is displayed:

```
static async Task Main(string[] args)
{
    // The port number(5001) must match the port of the gRPC server.
    using var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client =  new Greeter.GreeterClient(channel);
    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
}
```

Test the gRPC client with the gRPC Greeter service

- Visual Studio

- Visual Studio Code
- Visual Studio for Mac

- In the Greeter service, press `Ctrl+F5` to start the server without the debugger.
- In the `GrpcGreeterClient` project, press `Ctrl+F5` to start the client without the debugger.

The client sends a greeting to the service with a message containing its name, *GreeterClient*. The service sends the message "Hello GreeterClient" as a response. The "Hello GreeterClient" response is displayed in the command prompt:

```
Greeting: Hello GreeterClient
Press any key to exit...
```

The gRPC service records the details of the successful call in the logs written to the command prompt:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\GH\aspnet\docs\4\Docs\aspnetcore\tutorials\grpc\grpc-start\sample\GrpcGreeter
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5001/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 78.32260000000001ms 200 application/grpc
```

> **NOTE**
>
> The code in this article requires the ASP.NET Core HTTPS development certificate to secure the gRPC service. If the .NET gRPC client fails with the message `The remote certificate is invalid according to the validation procedure.` or `The SSL connection could not be established.`, the development certificate isn't trusted. To fix this issue, see Call a gRPC service with an untrusted/invalid certificate.

> **WARNING**
>
> ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this GitHub issue.

Next steps

- Introduction to gRPC on .NET Core
- gRPC services with C#
- Migrating gRPC services from C-core to ASP.NET Core

gRPC services with C#

9/22/2020 • 3 minutes to read • Edit Online

This document outlines the concepts needed to write gRPC apps in C#. The topics covered here apply to both C-core-based and ASP.NET Core-based gRPC apps.

> **WARNING**
>
> ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this GitHub issue.

proto file

gRPC uses a contract-first approach to API development. Protocol buffers (protobuf) are used as the Interface Definition Language (IDL) by default. The **.proto* file contains:

- The definition of the gRPC service.
- The messages sent between clients and servers.

For more information on the syntax of protobuf files, see Create Protobuf messages for .NET apps.

For example, consider the *greet.proto* file used in Get started with gRPC service:

- Defines a `Greeter` service.
- The `Greeter` service defines a `SayHello` call.
- `SayHello` sends a `HelloRequest` message and receives a `HelloReply` message:

```
syntax = "proto3";

option csharp_namespace = "GrpcGreeter";

package greet;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

Add a .proto file to a C# app

The **.proto* file is included in a project by adding it to the `<Protobuf>` item group:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

By default, a `<Protobuf>` reference generates a concrete client and a service base class. The reference element's `GrpcServices` attribute can be used to limit C# asset generation. Valid `GrpcServices` options are:

- `Both` (default when not present)
- `Server`
- `Client`
- `None`

C# Tooling support for .proto files

The tooling package Grpc.Tools is required to generate the C# assets from **.proto* files. The generated assets (files):

- Are generated on an as-needed basis each time the project is built.
- Aren't added to the project or checked into source control.
- Are a build artifact contained in the *obj* directory.

This package is required by both the server and client projects. The `Grpc.AspNetCore` metapackage includes a reference to `Grpc.Tools`. Server projects can add `Grpc.AspNetCore` using the Package Manager in Visual Studio or by adding a `<PackageReference>` to the project file:

```
<PackageReference Include="Grpc.AspNetCore" Version="2.28.0" />
```

Client projects should directly reference `Grpc.Tools` alongside the other packages required to use the gRPC client. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`:

```
<PackageReference Include="Google.Protobuf" Version="3.11.4" />
<PackageReference Include="Grpc.Net.Client" Version="2.28.0" />
<PackageReference Include="Grpc.Tools" Version="2.28.1">
```

Generated C# assets

The tooling package generates the C# types representing the messages defined in the included **.proto* files.

For server-side assets, an abstract service base type is generated. The base type contains the definitions of all the gRPC calls contained in the *.proto* file. Create a concrete service implementation that derives from this base type and implements the logic for the gRPC calls. For the `greet.proto`, the example described previously, an abstract `GreeterBase` type that contains a virtual `SayHello` method is generated. A concrete implementation `GreeterService` overrides the method and implements the logic handling the gRPC call.

```
public class GreeterService : Greeter.GreeterBase
{
    private readonly ILogger<GreeterService> _logger;
    public GreeterService(ILogger<GreeterService> logger)
    {
        _logger = logger;
    }

    public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

For client-side assets, a concrete client type is generated. The gRPC calls in the *.proto* file are translated into methods on the concrete type, which can be called. For the `greet.proto`, the example described previously, a concrete `GreeterClient` type is generated. Call `GreeterClient.SayHelloAsync` to initiate a gRPC call to the server.

```
static async Task Main(string[] args)
{
    // The port number(5001) must match the port of the gRPC server.
    using var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client =  new Greeter.GreeterClient(channel);
    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
}
```

By default, server and client assets are generated for each **.proto* file included in the `<Protobuf>` item group. To ensure only the server assets are generated in a server project, the `GrpcServices` attribute is set to `Server`.

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

Similarly, the attribute is set to `Client` in client projects.

Additional resources

- Introduction to gRPC on .NET Core
- Create a .NET Core gRPC client and server in ASP.NET Core
- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client

Create gRPC services and methods

9/22/2020 • 5 minutes to read • Edit Online

By James Newton-King

This document explains how to create gRPC services and methods in C#. Topics include:

- How to define services and methods in *.proto* files.
- Generated code using gRPC C# tooling.
- Implementing gRPC services and methods.

Create new gRPC services

gRPC services with C# introduced gRPC's contract-first approach to API development. Services and messages are defined in *.proto* files. C# tooling then generates code from *.proto* files. For server-side assets, an abstract base type is generated for each service, along with classes for any messages.

The following *.proto* file:

- Defines a `Greeter` service.
- The `Greeter` service defines a `SayHello` call.
- `SayHello` sends a `HelloRequest` message and receives a `HelloReply` message

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

C# tooling generates the C# `GreeterBase` base type:

```
public abstract partial class GreeterBase
{
    public virtual Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
    {
        throw new RpcException(new Status(StatusCode.Unimplemented, ""));
    }
}

public class HelloRequest
{
    public string Name { get; set; }
}

public class HelloReply
{
    public string Message { get; set; }
}
```

By default the generated `GreeterBase` doesn't do anything. Its virtual `SayHello` method will return an `UNIMPLEMENTED` error to any clients that call it. For the service to be useful an app must create a concrete implementation of `GreeterBase`:

```
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> UnaryCall(HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloRequest { Message = $"Hello {request.Name}" });
    }
}
```

The service implementation is registered with the app. If the service is hosted by ASP.NET Core gRPC, it should be added to the routing pipeline with the `MapGrpcService` method.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGrpcService<GreeterService>();
});
```

See gRPC services with ASP.NET Core for more information.

Implement gRPC methods

A gRPC service can have different types of methods. How messages are sent and received by a service depends on the type of method defined. The gRPC method types are:

- Unary
- Server streaming
- Client streaming
- Bi-directional streaming

Streaming calls are specified with the `stream` keyword in the *.proto* file. `stream` can be placed on a call's request message, response message, or both.

```
syntax = "proto3";

service ExampleService {
  // Unary
  rpc UnaryCall (ExampleRequest) returns (ExampleResponse);

  // Server streaming
  rpc StreamingFromServer (ExampleRequest) returns (stream ExampleResponse);

  // Client streaming
  rpc StreamingFromClient (stream ExampleRequest) returns (ExampleResponse);

  // Bi-directional streaming
  rpc StreamingBothWays (stream ExampleRequest) returns (stream ExampleResponse);
}
```

Each call type has a different method signature. Overriding generated methods from the abstract base service type in a concrete implementation ensures the correct arguments and return type are used.

Unary method

A unary method gets the request message as a parameter, and returns the response. A unary call is complete when the response is returned.

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request,
    ServerCallContext context)
{
    var response = new ExampleResponse();
    return Task.FromResult(response);
}
```

Unary calls are the most similar to actions on web API controllers. One important difference gRPC methods have from actions is gRPC methods are not able to bind parts of a request to different method arguments. gRPC methods always have one message argument for the incoming request data. Multiple values can still be sent to a gRPC service by making them fields on the request message:

```
message ExampleRequest {
    int pageIndex = 1;
    int pageSize = 2;
    bool isDescending = 3;
}
```

Server streaming method

A server streaming method gets the request message as a parameter. Because multiple messages can be streamed back to the caller, `responseStream.WriteAsync` is used to send response messages. A server streaming call is complete when the method returns.

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    for (var i = 0; i < 5; i++)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

The client has no way to send additional messages or data once the server streaming method has started. Some

streaming methods are designed to run forever. For continuous streaming methods, a client can cancel the call when it's no longer needed. When cancellation happens the client sends a signal to the server and the ServerCallContext.CancellationToken is raised. The `CancellationToken` token should be used on the server with async methods so that:

- Any asynchronous work is canceled together with the streaming call.
- The method exits quickly.

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    while (!context.CancellationToken.IsCancellationRequested)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1), context.CancellationToken);
    }
}
```

Client streaming method

A client streaming method starts *without* the method receiving a message. The `requestStream` parameter is used to read messages from the client. A client streaming call is complete when a response message is returned:

```
public override async Task<ExampleResponse> StreamingFromClient(
    IAsyncStreamReader<ExampleRequest> requestStream, ServerCallContext context)
{
    while (await requestStream.MoveNext())
    {
        var message = requestStream.Current;
        // ...
    }
    return new ExampleResponse();
}
```

When using C# 8 or later, the `await foreach` syntax can be used to read messages. The `IAsyncStreamReader<T>.ReadAllAsync()` extension method reads all messages from the request stream:

```
public override async Task<ExampleResponse> StreamingFromClient(
    IAsyncStreamReader<ExampleRequest> requestStream, ServerCallContext context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        // ...
    }
    return new ExampleResponse();
}
```

Bi-directional streaming method

A bi-directional streaming method starts *without* the method receiving a message. The `requestStream` parameter is used to read messages from the client. The method can choose to send messages with `responseStream.WriteAsync`.
A bi-directional streaming call is complete when the the method returns:

```
public override async Task StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        await responseStream.WriteAsync(new ExampleResponse());
    }
}
```

The preceding code:

- Sends a response for each request.
- Is a basic usage of bi-directional streaming.

It is possible to support more complex scenarios, such as reading requests and sending responses simultaneously:

```
public override async Task StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext context)
{
    // Read requests in a background task.
    var readTask = Task.Run(async () =>
    {
        await foreach (var message in requestStream.ReadAllAsync())
        {
            // Process request.
        }
    });

    // Send responses until the client signals that it is complete.
    while (!readTask.IsCompleted)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1), context.CancellationToken);
    }
}
```

In a bi-directional streaming method, the client and service can send messages to each other at any time. The best implementation of a bi-directional method varies depending upon requirements.

Access gRPC request headers

A request message is not the only way for a client to send data to a gRPC service. Header values are available in a service using `ServerCallContext.RequestHeaders`.

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request, ServerCallContext context)
{
    var userAgent = context.RequestHeaders.GetValue("user-agent");
    // ...

    return Task.FromResult(new ExampleResponse());
}
```

Additional resources

- gRPC services with C#
- Call gRPC services with the .NET client

Create Protobuf messages for .NET apps

9/22/2020 • 9 minutes to read • Edit Online

By James Newton-King and Mark Rendle

gRPC uses Protobuf as its Interface Definition Language (IDL). Protobuf IDL is a language neutral format for specifying the messages sent and received by gRPC services. Protobuf messages are defined in `.proto` files. This document explains how Protobuf concepts map to .NET.

Protobuf messages

Messages are the main data transfer object in Protobuf. They are conceptually similar to .NET classes.

```
syntax = "proto3";

option csharp_namespace = "Contoso.Messages";

message Person {
    int32 id = 1;
    string first_name = 2;
    string last_name = 3;
}
```

The preceding message definition specifies three fields as name-value pairs. Like properties on .NET types, each field has a name and a type. The field type can be a Protobuf scalar value type, e.g. `int32`, or another message.

In addition to a name, each field in the message definition has a unique number. Field numbers are used to identify fields when the message is serialized to Protobuf. Serializing a small number is faster than serializing the entire field name. Because field numbers identify a field it is important to take care when changing them. For more information about changing Protobuf messages see Versioning gRPC services.

When an app is built the Protobuf tooling generates .NET types from `.proto` files. The `Person` message generates a .NET class:

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

For more information about Protobuf messages see the Protobuf language guide.

Scalar Value Types

Protobuf supports a range of native scalar value types. The following table lists them all with their equivalent C# type:

PROTOBUF TYPE	C# TYPE
`double`	`double`

PROTOBUF TYPE	C# TYPE
`float`	`float`
`int32`	`int`
`int64`	`long`
`uint32`	`uint`
`uint64`	`ulong`
`sint32`	`int`
`sint64`	`long`
`fixed32`	`uint`
`fixed64`	`ulong`
`sfixed32`	`int`
`sfixed64`	`long`
`bool`	`bool`
`string`	`string`
`bytes`	`ByteString`

Scalar values always have a default value and can't be set to `null`. This constraint includes `string` and `ByteString` which are C# classes. `string` defaults to an empty string value and `ByteString` defaults to an empty bytes value. Attempting to set them to `null` throws an error.

Nullable wrapper types can be used to support null values.

Dates and times

The native scalar types don't provide for date and time values, equivalent to .NET's DateTimeOffset, DateTime, and TimeSpan. These types can be specified by using some of Protobuf's *Well-Known Types* extensions. These extensions provide code generation and runtime support for complex field types across the supported platforms.

The following table shows the date and time types:

.NET TYPE	PROTOBUF WELL-KNOWN TYPE
`DateTimeOffset`	`google.protobuf.Timestamp`
`DateTime`	`google.protobuf.Timestamp`
`TimeSpan`	`google.protobuf.Duration`

```
syntax = "proto3"

import "google/protobuf/duration.proto";
import "google/protobuf/timestamp.proto";

message Meeting {
    string subject = 1;
    google.protobuf.Timestamp start = 2;
    google.protobuf.Duration duration = 3;
}
```

The generated properties in the C# class aren't the .NET date and time types. The properties use the `Timestamp` and `Duration` classes in the `Google.Protobuf.WellKnownTypes` namespace. These classes provide methods for converting to and from `DateTimeOffset`, `DateTime`, and `TimeSpan`.

```
// Create Timestamp and Duration from .NET DateTimeOffset and TimeSpan.
var meeting = new Meeting
{
    Time = Timestamp.FromDateTimeOffset(meetingTime), // also FromDateTime()
    Duration = Duration.FromTimeSpan(meetingLength)
};

// Convert Timestamp and Duration to .NET DateTimeOffset and TimeSpan.
var time = meeting.Time.ToDateTimeOffset();
var duration = meeting.Duration?.ToTimeSpan();
```

> **NOTE**
>
> The `Timestamp` type works with UTC times. `DateTimeOffset` values always have an offset of zero, and the `DateTime.Kind` property is always `DateTimeKind.Utc`.

Nullable types

The Protobuf code generation for C# uses the native types, such as `int` for `int32`. So the values are always included and can't be `null`.

For values that require explicit `null`, such as using `int?` in C# code, Protobuf's Well-Known Types include wrappers that are compiled to nullable C# types. To use them, import `wrappers.proto` into your `.proto` file, like the following code:

```
syntax = "proto3"

import "google/protobuf/wrappers.proto"

message Person {
    // ...
    google.protobuf.Int32Value age = 5;
}
```

`wrappers.proto` types aren't exposed in generated properties. Protobuf automatically maps them to appropriate .NET nullable types in C# messages. For example, a `google.protobuf.Int32Value` field generates an `int?` property. Reference type properties like `string` and `ByteString` are unchanged except `null` can be assigned to them without error.

The following table shows the complete list of wrapper types with their equivalent C# type:

C# TYPE	WELL-KNOWN TYPE WRAPPER
`bool?`	`google.protobuf.BoolValue`
`double?`	`google.protobuf.DoubleValue`
`float?`	`google.protobuf.FloatValue`
`int?`	`google.protobuf.Int32Value`
`long?`	`google.protobuf.Int64Value`
`uint?`	`google.protobuf.UInt32Value`
`ulong?`	`google.protobuf.UInt64Value`
`string`	`google.protobuf.StringValue`
`ByteString`	`google.protobuf.BytesValue`

Bytes

Binary payloads are supported in Protobuf with the `bytes` scalar value type. A generated property in C# uses `ByteString` as the property type.

Use `ByteString.CopyFrom(byte[] data)` to create a new instance from a byte array:

```
var data = await File.ReadAllBytesAsync(path);

var payload = new PayloadResponse();
payload.Data = ByteString.CopyFrom(data);
```

`ByteString` data is accessed directly using `ByteString.Span` or `ByteString.Memory`. Or call `ByteString.ToByteArray()` to convert an instance back into a byte array:

```
var payload = await client.GetPayload(new PayloadRequest());

await File.WriteAllBytesAsync(path, payload.Data.ToByteArray());
```

Decimals

Protobuf doesn't natively support the .NET `decimal` type, just `double` and `float`. There's an ongoing discussion in the Protobuf project about the possibility of adding a standard decimal type to the Well-Known Types, with platform support for languages and frameworks that support it. Nothing has been implemented yet.

It's possible to create a message definition to represent the `decimal` type that works for safe serialization between .NET clients and servers. But developers on other platforms would have to understand the format being used and implement their own handling for it.

Creating a custom decimal type for Protobuf

```
package CustomTypes;

// Example: 12345.6789 -> { units = 12345, nanos = 678900000 }
message DecimalValue {

    // Whole units part of the amount
    int64 units = 1;

    // Nano units of the amount (10^-9)
    // Must be same sign as units
    sfixed32 nanos = 2;
}
```

The `nanos` field represents values from `0.999_999_999` to `-0.999_999_999`. For example, the `decimal` value `1.5m` would be represented as `{ units = 1, nanos = 500_000_000 }`. This is why the `nanos` field in this example uses the `sfixed32` type, which encodes more efficiently than `int32` for larger values. If the `units` field is negative, the `nanos` field should also be negative.

> **NOTE**
> There are multiple other algorithms for encoding `decimal` values as byte strings, but this message is easier to understand than any of them. The values are not affected by big-endian or little-endian on different platforms.

Conversion between this type and the BCL `decimal` type might be implemented in C# like this:

```
namespace CustomTypes
{
    public partial class DecimalValue
    {
        private const decimal NanoFactor = 1_000_000_000;
        public DecimalValue(long units, int nanos)
        {
            Units = units;
            Nanos = nanos;
        }

        public long Units { get; }
        public int Nanos { get; }

        public static implicit operator decimal(CustomTypes.DecimalValue grpcDecimal)
        {
            return grpcDecimal.Units + grpcDecimal.Nanos / NanoFactor;
        }

        public static implicit operator CustomTypes.DecimalValue(decimal value)
        {
            var units = decimal.ToInt64(value);
            var nanos = decimal.ToInt32((value - units) * NanoFactor);
            return new CustomTypes.DecimalValue(units, nanos);
        }
    }
}
```

Collections

Lists

Lists in Protobuf are specified by using the `repeated` prefix keyword on a field. The following example shows how to create a list:

```
message Person {
    // ...
    repeated string roles = 8;
}
```

In the generated code, `repeated` fields are represented by the `Google.Protobuf.Collections.RepeatedField<T>` generic type.

```
public class Person
{
    // ...
    public RepeatedField<string> Roles { get; }
}
```

`RepeatedField<T>` implements IList<T>. So you can use LINQ queries or convert it to an array or a list. `RepeatedField<T>` properties don't have a public setter. Items should be added to the existing collection.

```
var person = new Person();

// Add one item.
person.Roles.Add("user");

// Add all items from another collection.
var roles = new [] { "admin", "manager" };
person.Roles.Add(roles);
```

Dictionaries

The .NET IDictionary<TKey,TValue> type is represented in Protobuf using `map<key_type, value_type>`.

```
message Person {
    // ...
    map<string, string> attributes = 9;
}
```

In generated .NET code, `map` fields are represented by the `Google.Protobuf.Collections.MapField<TKey, TValue>` generic type. `MapField<TKey, TValue>` implements IDictionary<TKey,TValue>. Like `repeated` properties, `map` properties don't have a public setter. Items should be added to the existing collection.

```
var person = new Person();

// Add one item.
person.Attributes["created_by"] = "James";

// Add all items from another collection.
var attributes = new Dictionary<string, string>
{
    ["last_modified"] = DateTime.UtcNow.ToString()
};
person.Attributes.Add(attributes);
```

Unstructured and conditional messages

Protobuf is a contract-first messaging format. An app's messages, including its fields and types, must be specified in `.proto` files when the app is built. Protobuf's contract-first design is great at enforcing message content but can limit scenarios where a strict contract isn't required:

- Messages with unknown payloads. For example, a message with a field that could contain any message.
- Conditional messages. For example, a message returned from a gRPC service might be a success result or an error result.
- Dynamic values. For example, a message with a field that contains an unstructured collection of values, similar to JSON.

Protobuf offers language features and types to support these scenarios.

Any

The `Any` type lets you use messages as embedded types without having their `.proto` definition. To use the `Any` type, import `any.proto`.

```
import "google/protobuf/any.proto";

message Status {
    string message = 1;
    google.protobuf.Any detail = 2;
}
```

```
// Create a status with a Person message set to detail.
var status = new ErrorStatus();
status.Detail = Any.Pack(new Person { FirstName = "James" });

// Read Person message from detail.
if (status.Detail.Is(Person.Descriptor))
{
    var person = status.Detail.Unpack<Person>();
    // ...
}
```

Oneof

`oneof` fields are a language feature. The compiler handles the `oneof` keyword when it generates the message class. Using `oneof` to specify a response message that could either return a `Person` or `Error` might look like this:

```
message Person {
    // ...
}

message Error {
    // ...
}

message ResponseMessage {
  oneof result {
    Error error = 1;
    Person person = 2;
  }
}
```

Fields within the `oneof` set must have unique field numbers in the overall message declaration.

When using `oneof`, the generated C# code includes an enum that specifies which of the fields has been set. You can test the enum to find which field is set. Fields that aren't set return `null` or the default value, rather than throwing an exception.

```
var response = await client.GetPersonAsync(new RequestMessage());

switch (response.ResultCase)
{
    case ResponseMessage.ResultOneofCase.Person:
        HandlePerson(response.Person);
        break;
    case ResponseMessage.ResultOneofCase.Error:
        HandleError(response.Error);
        break;
    default:
        throw new ArgumentException("Unexpected result.");
}
```

Value

The `Value` type represents a dynamically typed value. It can be either `null`, a number, a string, a boolean, a dictionary of values (`Struct`), or a list of values (`ValueList`). `Value` is a Protobuf Well-Known Type that uses the previously discussed `oneof` feature. To use the `Value` type, import `struct.proto`.

```
import "google/protobuf/struct.proto";

message Status {
    // ...
    google.protobuf.Value data = 3;
}
```

```
// Create dynamic values.
var status = new Status();
status.Data = Value.FromStruct(new Struct
{
    Fields =
    {
        ["enabled"] = Value.ForBoolean(true),
        ["metadata"] = Value.ForList(
            Value.FromString("value1"),
            Value.FromString("value2"))
    }
});

// Read dynamic values.
switch (status.Data.KindCase)
{
    case Value.KindOneofCase.StructValue:
        foreach (var field in status.Data.StructValue.Fields)
        {
            // Read struct fields...
        }
        break;
    // ...
}
```

Using `Value` directly can be verbose. An alternative way to use `Value` is with Protobuf's built-in support for mapping messages to JSON. Protobuf's `JsonFormatter` and `JsonWriter` types can be used with any Protobuf message. `Value` is particularly well suited to being converted to and from JSON.

This is the JSON equivalent of the previous code:

```
// Create dynamic values from JSON.
var status = new Status();
status.Data = Value.Parser.ParseJson(@"{
    ""enabled"": true,
    ""metadata"": [ ""value1"", ""value2"" ]
}");

// Convert dynamic values to JSON.
// JSON can be read with a library like System.Text.Json or Newtonsoft.Json
var json = JsonFormatter.Default.Format(status.Metadata);
var document = JsonDocument.Parse(json);
```

Additional resources

- Protobuf language guide
- Versioning gRPC services

Versioning gRPC services

9/22/2020 • 5 minutes to read • Edit Online

By James Newton-King

New features added to an app can require gRPC services provided to clients to change, sometimes in unexpected and breaking ways. When gRPC services change:

- Consideration should be given on how changes impact clients.
- A versioning strategy to support changes should be implemented.

Backwards compatibility

The gRPC protocol is designed to support services that change over time. Generally, additions to gRPC services and methods are non-breaking. Non-breaking changes allow existing clients to continue working without changes. Changing or deleting gRPC services are breaking changes. When gRPC services have breaking changes, clients using that service have to be updated and redeployed.

Making non-breaking changes to a service has a number of benefits:

- Existing clients continue to run.
- Avoids work involved with notifying clients of breaking changes, and updating them.
- Only one version of the service needs to be documented and maintained.

Non-breaking changes

These changes are non-breaking at a gRPC protocol level and .NET binary level.

- **Adding a new service**
- **Adding a new method to a service**
- **Adding a field to a request message** - Fields added to a request message are deserialized with the default value on the server when not set. To be a non-breaking change, the service must succeed when the new field isn't set by older clients.
- **Adding a field to a response message** - Fields added to a response message are deserialized into the message's unknown fields collection on the client.
- **Adding a value to an enum** - Enums are serialized as a numeric value. New enum values are deserialized on the client to the enum value without an enum name. To be a non-breaking change, older clients must run correctly when receiving the new enum value.

Binary breaking changes

The following changes are non-breaking at a gRPC protocol level, but the client needs to be updated if it upgrades to the latest *.proto* contract or client .NET assembly. Binary compatibility is important if you plan to publish a gRPC library to NuGet.

- **Removing a field** - Values from a removed field are deserialized to a message's unknown fields. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract. It's important that a removed field number isn't accidentally reused in the future. To ensure this doesn't happen, specify deleted field numbers and names on the message using Protobuf's reserved keyword.
- **Renaming a message** - Message names aren't typically sent on the network, so this isn't a gRPC protocol breaking change. The client will need to be updated if it upgrades to the latest contract. One situation where message names **are** sent on the network is with Any fields, when the message name is used to identify the message type.

- **Changing csharp_namespace** - Changing `csharp_namespace` will change the namespace of generated .NET types. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract.

Protocol breaking changes

The following items are protocol and binary breaking changes:

- **Renaming a field** - With Protobuf content, the field names are only used in generated code. The field number is used to identify fields on the network. Renaming a field isn't a protocol breaking change for Protobuf. However, if a server is using JSON content then renaming a field is a breaking change.
- **Changing a field data type** - Changing a field's data type to an incompatible type will cause errors when deserializing the message. Even if the new data type is compatible, it's likely the client needs to be updated to support the new type if it upgrades to the latest contract.
- **Changing a field number** - With Protobuf payloads, the field number is used to identify fields on the network.
- **Renaming a package, service or method** - gRPC uses the package name, service name, and method name to build the URL. The client gets an *UNIMPLEMENTED* status from the server.
- **Removing a service or method** - The client gets an *UNIMPLEMENTED* status from the server when calling the removed method.

Behavior breaking changes

When making non-breaking changes, you must also consider whether older clients can continue working with the new service behavior. For example, adding a new field to a request message:

- Isn't a protocol breaking change.
- Returning an error status on the server if the new field isn't set makes it a breaking change for old clients.

Behavior compatibility is determined by your app-specific code.

Version number services

Services should strive to remain backwards compatible with old clients. Eventually changes to your app may require breaking changes. Breaking old clients and forcing them to be updated along with your service isn't a good user experience. A way to maintain backwards compatibility while making breaking changes is to publish multiple versions of a service.

gRPC supports an optional package specifier, which functions much like a .NET namespace. In fact, the `package` will be used as the .NET namespace for generated .NET types if `option csharp_namespace` is not set in the *.proto* file. The package can be used to specify a version number for your service and its messages:

```
syntax = "proto3";

package greet.v1;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

The package name is combined with the service name to identify a service address. A service address allows

multiple versions of a service to be hosted side-by-side:

- `greet.v1.Greeter`
- `greet.v2.Greeter`

Implementations of the versioned service are registered in *Startup.cs*:

```
app.UseEndpoints(endpoints =>
{
    // Implements greet.v1.Greeter
    endpoints.MapGrpcService<GreeterServiceV1>();

    // Implements greet.v2.Greeter
    endpoints.MapGrpcService<GreeterServiceV2>();
});
```

Including a version number in the package name gives you the opportunity to publish a *v2* version of your service with breaking changes, while continuing to support older clients who call the *v1* version. Once clients have updated to use the *v2* service, you can choose to remove the old version. When planning to publish multiple versions of a service:

- Avoid breaking changes if reasonable.
- Don't update the version number unless making breaking changes.
- Do update the version number when you make breaking changes.

Publishing multiple versions of a service duplicates it. To reduce duplication, consider moving business logic from the service implementations to a centralized location that can be reused by the old and new implementations:

```
using Greet.V1;
using Grpc.Core;
using System.Threading.Tasks;

namespace Services
{
    public class GreeterServiceV1 : Greeter.GreeterBase
    {
        private readonly IGreeter _greeter;
        public GreeterServiceV1(IGreeter greeter)
        {
            _greeter = greeter;
        }

        public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
        {
            return Task.FromResult(new HelloReply
            {
                Message = _greeter.GetHelloMessage(request.Name)
            });
        }
    }
}
```

Services and messages generated with different package names are **different .NET types**. Moving business logic to a centralized location requires mapping messages to common types.

Additional resources

- Create Protobuf messages for .NET apps

Call gRPC services with the .NET client

9/22/2020 • 7 minutes to read • Edit Online

A .NET gRPC client library is available in the Grpc.Net.Client NuGet package. This document explains how to:

- Configure a gRPC client to call gRPC services.
- Make gRPC calls to unary, server streaming, client streaming, and bi-directional streaming methods.

Configure gRPC client

gRPC clients are concrete client types that are generated from **.proto* files. The concrete gRPC client has methods that translate to the gRPC service in the **.proto* file. For example, a service called `Greeter` generates a `GreeterClient` type with methods to call the service.

A gRPC client is created from a channel. Start by using `GrpcChannel.ForAddress` to create a channel, and then use the channel to create a gRPC client:

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greet.GreeterClient(channel);
```

A channel represents a long-lived connection to a gRPC service. When a channel is created, it is configured with options related to calling a service. For example, the `HttpClient` used to make calls, the maximum send and receive message size, and logging can be specified on `GrpcChannelOptions` and used with `GrpcChannel.ForAddress`. For a complete list of options, see client configuration options.

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");

var greeterClient = new Greet.GreeterClient(channel);
var counterClient = new Count.CounterClient(channel);

// Use clients to call gRPC services
```

Configure TLS

A gRPC client must use the same connection-level security as the called service. gRPC client Transport Layer Security (TLS) is configured when the gRPC channel is created. A gRPC client throws an error when it calls a service and the connection-level security of the channel and service don't match.

To configure a gRPC channel to use TLS, ensure the server address starts with `https`. For example, `GrpcChannel.ForAddress("https://localhost:5001")` uses HTTPS protocol. The gRPC channel automatically negotiates a connection secured by TLS and uses a secure connection to make gRPC calls.

> **TIP**
>
> gRPC supports client certificate authentication over TLS. For information on configuring client certificates with a gRPC channel, see Authentication and authorization in gRPC for ASP.NET Core.

To call unsecured gRPC services, ensure the server address starts with `http`. For example, `GrpcChannel.ForAddress("http://localhost:5000")` uses HTTP protocol. In .NET Core 3.1 or later, additional configuration is required to call insecure gRPC services with the .NET client.

Client performance

Channel and client performance and usage:

- Creating a channel can be an expensive operation. Reusing a channel for gRPC calls provides performance benefits.
- gRPC clients are created with channels. gRPC clients are lightweight objects and don't need to be cached or reused.
- Multiple gRPC clients can be created from a channel, including different types of clients.
- A channel and clients created from the channel can safely be used by multiple threads.
- Clients created from the channel can make multiple simultaneous calls.

`GrpcChannel.ForAddress` isn't the only option for creating a gRPC client. If calling gRPC services from an ASP.NET Core app, consider gRPC client factory integration. gRPC integration with `HttpClientFactory` offers a centralized alternative to creating gRPC clients.

> **NOTE**
> Calling gRPC over HTTP/2 with `Grpc.Net.Client` is currently not supported on Xamarin. We are working to improve HTTP/2 support in a future Xamarin release. Grpc.Core and gRPC-Web are viable alternatives that work today.

Make gRPC calls

A gRPC call is initiated by calling a method on the client. The gRPC client will handle message serialization and addressing the gRPC call to the correct service.

gRPC has different types of methods. How the client is used to make a gRPC call depends on the type of method called. The gRPC method types are:

- Unary
- Server streaming
- Client streaming
- Bi-directional streaming

Unary call

A unary call starts with the client sending a request message. A response message is returned when the service finishes.

```
var client = new Greet.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = "World" });

Console.WriteLine("Greeting: " + response.Message);
// Greeting: Hello World
```

Each unary service method in the **.proto* file will result in two .NET methods on the concrete gRPC client type for calling the method: an asynchronous method and a blocking method. For example, on `GreeterClient` there are two ways of calling `SayHello`:

- `GreeterClient.SayHelloAsync` - calls `Greeter.SayHello` service asynchronously. Can be awaited.
- `GreeterClient.SayHello` - calls `Greeter.SayHello` service and blocks until complete. Don't use in asynchronous code.

Server streaming call

A server streaming call starts with the client sending a request message. `ResponseStream.MoveNext()` reads

messages streamed from the service. The server streaming call is complete when `ResponseStream.MoveNext()` returns `false`.

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

while (await call.ResponseStream.MoveNext())
{
    Console.WriteLine("Greeting: " + call.ResponseStream.Current.Message);
    // "Greeting: Hello World" is written multiple times
}
```

When using C# 8 or later, the `await foreach` syntax can be used to read messages. The `IAsyncStreamReader<T>.ReadAllAsync()` extension method reads all messages from the response stream:

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

await foreach (var response in call.ResponseStream.ReadAllAsync())
{
    Console.WriteLine("Greeting: " + response.Message);
    // "Greeting: Hello World" is written multiple times
}
```

Client streaming call

A client streaming call starts *without* the client sending a message. The client can choose to send messages with `RequestStream.WriteAsync`. When the client has finished sending messages, `RequestStream.CompleteAsync()` should be called to notify the service. The call is finished when the service returns a response message.

```
var client = new Counter.CounterClient(channel);
using var call = client.AccumulateCount();

for (var i = 0; i < 3; i++)
{
    await call.RequestStream.WriteAsync(new CounterRequest { Count = 1 });
}
await call.RequestStream.CompleteAsync();

var response = await call;
Console.WriteLine($"Count: {response.Count}");
// Count: 3
```

Bi-directional streaming call

A bi-directional streaming call starts *without* the client sending a message. The client can choose to send messages with `RequestStream.WriteAsync`. Messages streamed from the service are accessible with `ResponseStream.MoveNext()` or `ResponseStream.ReadAllAsync()`. The bi-directional streaming call is complete when the `ResponseStream` has no more messages.

```
var client = new Echo.EchoClient(channel);
using var call = client.Echo();

Console.WriteLine("Starting background task to receive messages");
var readTask = Task.Run(async () =>
{
    await foreach (var response in call.ResponseStream.ReadAllAsync())
    {
        Console.WriteLine(response.Message);
        // Echo messages sent to the service
    }
});

Console.WriteLine("Starting to send messages");
Console.WriteLine("Type a message to echo then press enter.");
while (true)
{
    var result = Console.ReadLine();
    if (string.IsNullOrEmpty(result))
    {
        break;
    }

    await call.RequestStream.WriteAsync(new EchoMessage { Message = result });
}

Console.WriteLine("Disconnecting");
await call.RequestStream.CompleteAsync();
await readTask;
```

For best performance, and to avoid unnecessary errors in the client and service, try to complete bi-directional streaming calls gracefully. A bi-directional call completes gracefully when the server has finished reading the request stream and the client has finished reading the response stream. The preceding sample call is one example of a bi-directional call that ends gracefully. In the call, the client:

1. Starts a new bi-directional streaming call by calling `EchoClient.Echo`.
2. Creates a background task to read messages from the service using `ResponseStream.ReadAllAsync()`.
3. Sends messages to the server with `RequestStream.WriteAsync`.
4. Notifies the server it has finished sending messages with `RequestStream.CompleteAsync()`.
5. Waits until the background task has read all incoming messages.

During a bi-directional streaming call, the client and service can send messages to each other at any time. The best client logic for interacting with a bi-directional call varies depending upon the service logic.

Access gRPC trailers

gRPC calls may return gRPC trailers. gRPC trailers are used to provide name/value metadata about a call. Trailers provide similar functionality to HTTP headers, but are received at the end of the call.

gRPC trailers are accessible using `GetTrailers()`, which returns a collection of metadata. Trailers are returned after the response is complete, therefore, you must await all response messages before accessing the trailers.

Unary and client streaming calls must await `ResponseAsync` before calling `GetTrailers()`:

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHelloAsync(new HelloRequest { Name = "World" });
var response = await call.ResponseAsync;

Console.WriteLine("Greeting: " + response.Message);
// Greeting: Hello World

var trailers = call.GetTrailers();
var myValue = trailers.GetValue("my-trailer-name");
```

Server and bidirectional streaming calls must finish awaiting the response stream before calling `GetTrailers()`:

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

await foreach (var response in call.ResponseStream.ReadAllAsync())
{
    Console.WriteLine("Greeting: " + response.Message);
    // "Greeting: Hello World" is written multiple times
}

var trailers = call.GetTrailers();
var myValue = trailers.GetValue("my-trailer-name");
```

gRPC trailers are also accessible from `RpcException`. A service may return trailers together with a non-OK gRPC status. In this situation the trailers are retrieved from the exception thrown by the gRPC client:

```
var client = new Greet.GreeterClient(channel);
string myValue = null;

try
{
    using var call = client.SayHelloAsync(new HelloRequest { Name = "World" });
    var response = await call.ResponseAsync;

    Console.WriteLine("Greeting: " + response.Message);
    // Greeting: Hello World

    var trailers = call.GetTrailers();
    myValue = trailers.GetValue("my-trailer-name");
}
catch (RpcException ex)
{
    var trailers = ex.Trailers;
    myValue = trailers.GetValue("my-trailer-name");
}
```

Configure deadline

Configuring a gRPC call deadline is recommended because it provides an upper limit on how long a call can run for. It stops misbehaving services from running forever and exhausting server resources. Deadlines are a useful tool for building reliable apps.

Configure `CallOptions.Deadline` to set a deadline for a gRPC call:

```
var client = new Greet.GreeterClient(channel);

try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = "World" },
        deadline: DateTime.UtcNow.AddSeconds(5));

    // Greeting: Hello World
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded)
{
    Console.WriteLine("Greeting timeout.");
}
```

For more information, see Reliable gRPC services with deadlines and cancellation.

Additional resources

- gRPC client factory integration in .NET Core
- Reliable gRPC services with deadlines and cancellation
- gRPC services with C#

gRPC client factory integration in .NET Core

9/22/2020 • 2 minutes to read • Edit Online

gRPC integration with `HttpClientFactory` offers a centralized way to create gRPC clients. It can be used as an alternative to configuring stand-alone gRPC client instances. Factory integration is available in the Grpc.Net.ClientFactory NuGet package.

The factory offers the following benefits:

- Provides a central location for configuring logical gRPC client instances
- Manages the lifetime of the underlying `HttpClientMessageHandler`
- Automatic propagation of deadline and cancellation in an ASP.NET Core gRPC service

Register gRPC clients

To register a gRPC client, the generic `AddGrpcClient` extension method can be used within `Startup.ConfigureServices`, specifying the gRPC typed client class and service address:

```
services.AddGrpcClient<Greeter.GreeterClient>(o =>
{
    o.Address = new Uri("https://localhost:5001");
});
```

The gRPC client type is registered as transient with dependency injection (DI). The client can now be injected and consumed directly in types created by DI. ASP.NET Core MVC controllers, SignalR hubs and gRPC services are places where gRPC clients can automatically be injected:

```
public class AggregatorService : Aggregator.AggregatorBase
{
    private readonly Greeter.GreeterClient _client;

    public AggregatorService(Greeter.GreeterClient client)
    {
        _client = client;
    }

    public override async Task SayHellos(HelloRequest request,
        IServerStreamWriter<HelloReply> responseStream, ServerCallContext context)
    {
        // Forward the call on to the greeter service
        using (var call = _client.SayHellos(request))
        {
            await foreach (var response in call.ResponseStream.ReadAllAsync())
            {
                await responseStream.WriteAsync(response);
            }
        }
    }
}
```

Configure HttpClient

`HttpClientFactory` creates the `HttpClient` used by the gRPC client. Standard `HttpClientFactory` methods can be used to add outgoing request middleware or to configure the underlying `HttpClientHandler` of the `HttpClient`:

```
services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        var handler = new HttpClientHandler();
        handler.ClientCertificates.Add(LoadCertificate());
        return handler;
    });
```

For more information, see Make HTTP requests using IHttpClientFactory.

Configure Channel and Interceptors

gRPC-specific methods are available to:

- Configure a gRPC client's underlying channel.
- Add `Interceptor` instances that the client will use when making gRPC calls.

```
services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddInterceptor(() => new LoggingInterceptor())
    .ConfigureChannel(o =>
    {
        o.Credentials = new CustomCredentials();
    });
```

Deadline and cancellation propagation

gRPC clients created by the factory in a gRPC service can be configured with `EnableCallContextPropagation()` to automatically propagate the deadline and cancellation token to child calls. The `EnableCallContextPropagation()` extension method is available in the Grpc.AspNetCore.Server.ClientFactory NuGet package.

Call context propagation works by reading the deadline and cancellation token from the current gRPC request context and automatically propagating them to outgoing calls made by the gRPC client. Call context propagation is an excellent way of ensuring that complex, nested gRPC scenarios always propagate the deadline and cancellation.

```
services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation();
```

By default, `EnableCallContextPropagation` raises an error if the client is used outside the context of a gRPC call. The error is designed to alert you that there isn't a call context to propagate. If you want to use the client outside of a call context, suppress the error when the client is configured with `SuppressContextNotFoundErrors`:

```
services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation(o => o.SuppressContextNotFoundErrors = true);
```

For more information about deadlines and RPC cancellation, see RPC life cycle.

Additional resources

- Call gRPC services with the .NET client
- Make HTTP requests using IHttpClientFactory in ASP.NET Core

Reliable gRPC services with deadlines and cancellation

9/22/2020 • 3 minutes to read • Edit Online

By James Newton-King

Deadlines and cancellation are features used by gRPC clients to abort in-progress calls. This article discusses why deadlines and cancellation are important, and how to use them in .NET gRPC apps.

Deadlines

A deadline allows a gRPC client to specify how long it will wait for a call to complete. When a deadline is exceeded, the call is canceled. Setting a deadline is important because it provides an upper limit on how long a call can run for. It stops misbehaving services from running forever and exhausting server resources. Deadlines are a useful tool for building reliable apps and should be configured.

Deadline configuration:

- A deadline is configured using `CallOptions.Deadline` when a call is made.
- There is no default deadline value. gRPC calls aren't time limited unless a deadline is specified.
- A deadline is the UTC time of when the deadline is exceeded. For example, `DateTime.UtcNow.AddSeconds(5)` is a deadline of 5 seconds from now.
- If a past or current time is used then the call immediately exceeds the deadline.
- The deadline is sent with the gRPC call to the service and is independently tracked by both the client and the service. It is possible that a gRPC call completes on one machine, but by the time the response has returned to the client the deadline has been exceeded.

If a deadline is exceeded, the client and service have different behavior:

- The client immediately aborts the underlying HTTP request and throws a `DeadlineExceeded` error. The client app can choose to catch the error and display a timeout message to the user.
- On the server, the executing HTTP request is aborted and ServerCallContext.CancellationToken is raised. Although the HTTP request is aborted, the gRPC call continues to run on the server until the method completes. It's important that the cancellation token is passed to async methods so they are cancelled along with the call. For example, passing a cancellation token to async database queries and HTTP requests. Passing a cancellation token allows the canceled call to complete quickly on the server and free up resources for other calls.

Configure `CallOptions.Deadline` to set a deadline for a gRPC call:

```
var client = new Greet.GreeterClient(channel);

try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = "World" },
        deadline: DateTime.UtcNow.AddSeconds(5));

    // Greeting: Hello World
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded)
{
    Console.WriteLine("Greeting timeout.");
}
```

Using `ServerCallContext.CancellationToken` in a gRPC service:

```
public override async Task<HelloReply> SayHello(HelloRequest request,
    ServerCallContext context)
{
    var user = await _databaseContext.GetUserAsync(request.Name,
        context.CancellationToken);

    return new HelloReply { Message = "Hello " + user.DisplayName };
}
```

Propagating deadlines

When a gRPC call is made from an executing gRPC service, the deadline should be propagated. For example:

1. Client app calls `FrontendService.GetUser` with a deadline.
2. `FrontendService` calls `UserService.GetUser`. The deadline specified by the client should be specified with the new gRPC call.
3. `UserService.GetUser` receives the deadline. It correctly times-out if the client app's deadline is exceeded.

The call context provides the deadline with `ServerCallContext.Deadline`:

```
public override async Task<UserResponse> GetUser(UserRequest request,
    ServerCallContext context)
{
    var client = new User.UserServiceClient(_channel);
    var response = await client.GetUserAsync(
        new UserRequest { Id = request.Id },
        deadline: context.Deadline);

    return response;
}
```

Manually propagating deadlines can be cumbersome. The deadline needs to be passed to every call, and it's easy to accidentally miss. An automatic solution is available with gRPC client factory. Specifying `EnableCallContextPropagation`:

- Automatically propagates the deadline and cancellation token to child calls.
- Is an excellent way of ensuring that complex, nested gRPC scenarios always propagate the deadline and cancellation.

```
services
    .AddGrpcClient<User.UserServiceClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation();
```

For more information, see gRPC client factory integration in .NET Core.

Cancellation

Cancellation allows a gRPC client to cancel long running calls that are no longer needed. For example, a gRPC call that streams realtime updates is started when the user visits a page on a website. The stream should be canceled when the user navigates away from the page.

A gRPC call can be canceled in the client by passing a cancellation token with CallOptions.CancellationToken or calling `Dispose` on the call.

```
private AsyncServerStreamingCall<HelloReply> _call;

public void StartStream()
{
    _call = client.SayHellos(new HelloRequest { Name = "World" });

    // Read response in background task.
    _ = Task.Run(async () =>
    {
        await foreach (var response in _call.ResponseStream.ReadAllAsync())
        {
            Console.WriteLine("Greeting: " + response.Message);
        }
    });
}

public void StopStream()
{
    _call.Dispose();
}
```

gRPC services that can be cancelled should:

- Pass `ServerCallContext.CancellationToken` to async methods. Canceling async methods allows the call on the server to complete quickly.
- Propagate the cancellation token to child calls. Propagating the cancellation token ensures that child calls are canceled with their parent. gRPC client factory and `EnableCallContextPropagation()` automatically propagates the cancellation token.

Additional resources

- Call gRPC services with the .NET client
- gRPC client factory integration in .NET Core

gRPC services with ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

This document shows how to get started with gRPC services using ASP.NET Core.

> **WARNING**
> ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this GitHub issue.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK or later

Get started with gRPC service in ASP.NET Core

View or download sample code (how to download).

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

See Get started with gRPC services for detailed instructions on how to create a gRPC project.

Add gRPC services to an ASP.NET Core app

gRPC requires the Grpc.AspNetCore package.

Configure gRPC

In *Startup.cs*:

- gRPC is enabled with the `AddGrpc` method.
- Each gRPC service is added to the routing pipeline through the `MapGrpcService` method.

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddGrpc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            // Communication with gRPC endpoints must be made through a gRPC client.
            // To learn how to create a client, visit: https://go.microsoft.com/fwlink/?linkid=2086909
            endpoints.MapGrpcService<GreeterService>();
        });
    }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

ASP.NET Core middlewares and features share the routing pipeline, therefore an app can be configured to serve additional request handlers. The additional request handlers, such as MVC controllers, work in parallel with the configured gRPC services.

Configure Kestrel

Kestrel gRPC endpoints:

- Require HTTP/2.
- Should be secured with Transport Layer Security (TLS).

HTTP/2

gRPC requires HTTP/2. gRPC for ASP.NET Core validates HttpRequest.Protocol is `HTTP/2`.

Kestrel supports HTTP/2 on most modern operating systems. Kestrel endpoints are configured to support HTTP/1.1 and HTTP/2 connections by default.

TLS

Kestrel endpoints used for gRPC should be secured with TLS. In development, an endpoint secured with TLS is automatically created at `https://localhost:5001` when the ASP.NET Core development certificate is present. No configuration is required. An `https` prefix verifies the Kestrel endpoint is using TLS.

In production, TLS must be explicitly configured. In the following *appsettings.json* example, an HTTP/2 endpoint secured with TLS is provided:

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Protocols": "Http2",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      }
    }
  }
}
```

Alternatively, Kestrel endpoints can be configured in *Program.cs*:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(options =>
            {
                options.Listen(IPAddress.Any, 5001, listenOptions =>
                {
                    listenOptions.Protocols = HttpProtocols.Http2;
                    listenOptions.UseHttps("<path to .pfx file>",
                        "<certificate password>");
                });
            });
            webBuilder.UseStartup<Startup>();
        });
```

Protocol negotiation

TLS is used for more than securing communication. The TLS Application-Layer Protocol Negotiation (ALPN) handshake is used to negotiate the connection protocol between the client and the server when an endpoint supports multiple protocols. This negotiation determines whether the connection uses HTTP/1.1 or HTTP/2.

If an HTTP/2 endpoint is configured without TLS, the endpoint's ListenOptions.Protocols must be set to `HttpProtocols.Http2`. An endpoint with multiple protocols (for example, `HttpProtocols.Http1AndHttp2`) can't be used without TLS because there is no negotiation. All connections to the unsecured endpoint default to HTTP/1.1, and gRPC calls fail.

For more information on enabling HTTP/2 and TLS with Kestrel, see Kestrel endpoint configuration.

> **NOTE**
> macOS doesn't support ASP.NET Core gRPC with TLS. Additional configuration is required to successfully run gRPC services on macOS. For more information, see Unable to start ASP.NET Core gRPC app on macOS.

Integration with ASP.NET Core APIs

gRPC services have full access to the ASP.NET Core features such as Dependency Injection (DI) and Logging. For example, the service implementation can resolve a logger service from the DI container via the constructor:

```
public class GreeterService : Greeter.GreeterBase
{
    public GreeterService(ILogger<GreeterService> logger)
    {
    }
}
```

By default, the gRPC service implementation can resolve other DI services with any lifetime (Singleton, Scoped, or Transient).

Resolve HttpContext in gRPC methods

The gRPC API provides access to some HTTP/2 message data, such as the method, host, header, and trailers. Access is through the `ServerCallContext` argument passed to each gRPC method:

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(
        HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

`ServerCallContext` does not provide full access to `HttpContext` in all ASP.NET APIs. The `GetHttpContext` extension method provides full access to the `HttpContext` representing the underlying HTTP/2 message in ASP.NET APIs:

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(
        HelloRequest request, ServerCallContext context)
    {
        var httpContext = context.GetHttpContext();
        var clientCertificate = httpContext.Connection.ClientCertificate;

        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name + " from " + clientCertificate.Issuer
        });
    }
}
```

Additional resources

- Create a .NET Core gRPC client and server in ASP.NET Core
- Introduction to gRPC on .NET Core
- gRPC services with C#
- Kestrel web server implementation in ASP.NET Core

Use gRPC in browser apps

9/22/2020 • 5 minutes to read • Edit Online

By James Newton-King

Learn how to configure an existing ASP.NET Core gRPC service to be callable from browser apps, using the gRPC-Web protocol. gRPC-Web allows browser JavaScript and Blazor apps to call gRPC services. It's not possible to call an HTTP/2 gRPC service from a browser-based app. gRPC services hosted in ASP.NET Core can be configured to support gRPC-Web alongside HTTP/2 gRPC.

For instructions on adding a gRPC service to an existing ASP.NET Core app, see Add gRPC services to an ASP.NET Core app.

For instructions on creating a gRPC project, see Create a .NET Core gRPC client and server in ASP.NET Core.

gRPC-Web in ASP.NET Core vs. Envoy

There are two choices for how to add gRPC-Web to an ASP.NET Core app:

- Support gRPC-Web alongside gRPC HTTP/2 in ASP.NET Core. This option uses middleware provided by the `Grpc.AspNetCore.Web` package.
- Use the Envoy proxy's gRPC-Web support to translate gRPC-Web to gRPC HTTP/2. The translated call is then forwarded onto the ASP.NET Core app.

There are pros and cons to each approach. If an app's environment is already using Envoy as a proxy, it might make sense to also use Envoy to provide gRPC-Web support. For a basic solution for gRPC-Web that only requires ASP.NET Core, `Grpc.AspNetCore.Web` is a good choice.

Configure gRPC-Web in ASP.NET Core

gRPC services hosted in ASP.NET Core can be configured to support gRPC-Web alongside HTTP/2 gRPC. gRPC-Web does not require any changes to services. The only modification is startup configuration.

To enable gRPC-Web with an ASP.NET Core gRPC service:

- Add a reference to the Grpc.AspNetCore.Web package.
- Configure the app to use gRPC-Web by adding `UseGrpcWeb` and `EnableGrpcWeb` to *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();
}

public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseGrpcWeb(); // Must be added between UseRouting and UseEndpoints

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGrpcService<GreeterService>().EnableGrpcWeb();
    });
}
```

The preceding code:

- Adds the gRPC-Web middleware, `UseGrpcWeb`, after routing and before endpoints.
- Specifies the `endpoints.MapGrpcService<GreeterService>()` method supports gRPC-Web with `EnableGrpcWeb`.

Alternatively, the gRPC-Web middleware can be configured so all services support gRPC-Web by default and `EnableGrpcWeb` isn't required. Specify `new GrpcWebOptions { DefaultEnabled = true }` when the middleware is added.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddGrpc();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseRouting();

        app.UseGrpcWeb(new GrpcWebOptions { DefaultEnabled = true });

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGrpcService<GreeterService>();
        });
    }
}
```

> **NOTE**
>
> There is a known issue that causes gRPC-Web to fail when hosted by Http.sys in .NET Core 3.x.
>
> A workaround to get gRPC-Web working on Http.sys is available here.

gRPC-Web and CORS

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction applies to making gRPC-Web calls with browser apps. For example, a browser app served by `https://www.contoso.com` is blocked from calling gRPC-Web services hosted on `https://services.contoso.com`. Cross Origin Resource Sharing (CORS) can be used to relax this restriction.

To allow a browser app to make cross-origin gRPC-Web calls, set up CORS in ASP.NET Core. Use the built-in CORS support, and expose gRPC-specific headers with WithExposedHeaders.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();

    services.AddCors(o => o.AddPolicy("AllowAll", builder =>
    {
        builder.AllowAnyOrigin()
               .AllowAnyMethod()
               .AllowAnyHeader()
               .WithExposedHeaders("Grpc-Status", "Grpc-Message", "Grpc-Encoding", "Grpc-Accept-Encoding");
    }));
}

public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseGrpcWeb();
    app.UseCors();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGrpcService<GreeterService>().EnableGrpcWeb()
                                                   .RequireCors("AllowAll");
    });
}
```

The preceding code:

- Calls `AddCors` to add CORS services and configures a CORS policy that exposes gRPC-specific headers.
- Calls `UseCors` to add the CORS middleware after routing and before endpoints.
- Specifies the `endpoints.MapGrpcService<GreeterService>()` method supports CORS with `RequiresCors`.

gRPC-Web and streaming

Traditional gRPC over HTTP/2 supports streaming in all directions. gRPC-Web offers limited support for streaming:

- gRPC-Web browser clients don't support calling client streaming and bidirectional streaming methods.
- ASP.NET Core gRPC services hosted on Azure App Service and IIS don't support bidirectional streaming.

When using gRPC-Web, we only recommend the use of unary methods and server streaming methods.

Call gRPC-Web from the browser

Browser apps can use gRPC-Web to call gRPC services. There are some requirements and limitations when calling gRPC services with gRPC-Web from the browser:

- The server must have been configured to support gRPC-Web.
- Client streaming and bidirectional streaming calls aren't supported. Server streaming is supported.
- Calling gRPC services on a different domain requires CORS to be configured on the server.

JavaScript gRPC-Web client

There is a JavaScript gRPC-Web client. For instructions on how to use gRPC-Web from JavaScript, see write JavaScript client code with gRPC-Web.

Configure gRPC-Web with the .NET gRPC client

The .NET gRPC client can be configured to make gRPC-Web calls. This is useful for Blazor WebAssembly apps, which are hosted in the browser and have the same HTTP limitations of JavaScript code. Calling gRPC-Web with a .NET client is the same as HTTP/2 gRPC. The only modification is how the channel is created.

To use gRPC-Web:

- Add a reference to the Grpc.Net.Client.Web package.
- Ensure the reference to Grpc.Net.Client package is 2.29.0 or greater.
- Configure the channel to use the `GrpcWebHandler`:

```
var channel = GrpcChannel.ForAddress("https://localhost:5001", new GrpcChannelOptions
    {
        HttpHandler = new GrpcWebHandler(new HttpClientHandler())
    });

var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = ".NET" });
```

The preceding code:

- Configures a channel to use gRPC-Web.
- Creates a client and makes a call using the channel.

`GrpcWebHandler` has the following configuration options:

- **InnerHandler**: The underlying HttpMessageHandler that makes the gRPC HTTP request, for example, `HttpClientHandler`.
- **GrpcWebMode**: An enumeration type that specifies whether the gRPC HTTP request `Content-Type` is `application/grpc-web` or `application/grpc-web-text`.
  - `GrpcWebMode.GrpcWeb` configures content to be sent without encoding. Default value.
  - `GrpcWebMode.GrpcWebText` configures content to be base64 encoded. Required for server streaming calls in browsers.
- **HttpVersion**: HTTP protocol `Version` used to set HttpRequestMessage.Version on the underlying gRPC HTTP request. gRPC-Web doesn't require a specific version and doesn't override the default unless specified.

> **IMPORTANT**
> Generated gRPC clients have sync and async methods for calling unary methods. For example, `SayHello` is sync and `SayHelloAsync` is async. Calling a sync method in a Blazor WebAssembly app will cause the app to become unresponsive. Async methods must always be used in Blazor WebAssembly.

Use gRPC client factory with gRPC-Web

A gRPC-Web compatible .NET client can be created using gRPC's integration with HttpClientFactory.

To use gRPC-Web with client factory:

- Add package references to the project file for the following packages:
  - Grpc.Net.Client.Web
  - Grpc.Net.ClientFactory
- Register a gRPC client with dependency injection (DI) using the generic `AddGrpcClient` extension method. In a Blazor WebAssembly app, services are registered with DI in `Program.cs`.
- Configure `GrpcWebHandler` using the ConfigurePrimaryHttpMessageHandler extension method.

```
builder.Services
    .AddGrpcClient<Greet.GreeterClient>((services, options) =>
    {
        options.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(
        () => new GrpcWebHandler(GrpcWebMode.GrpcWebText, new HttpClientHandler()));
```

For more information, see gRPC client factory integration in .NET Core.

Additional resources

- gRPC for Web Clients GitHub project
- Enable Cross-Origin Requests (CORS) in ASP.NET Core
- Create JSON Web APIs from gRPC

gRPC for .NET configuration

9/22/2020 • 4 minutes to read • Edit Online

Configure services options

gRPC services are configured with `AddGrpc` in *Startup.cs*. The following table describes options for configuring gRPC services:

OPTION	DEFAULT VALUE	DESCRIPTION
MaxSendMessageSize	`null`	The maximum message size in bytes that can be sent from the server. Attempting to send a message that exceeds the configured maximum message size results in an exception. When set to `null`, the message size is unlimited.
MaxReceiveMessageSize	4 MB	The maximum message size in bytes that can be received by the server. If the server receives a message that exceeds this limit, it throws an exception. Increasing this value allows the server to receive larger messages, but can negatively impact memory consumption. When set to `null`, the message size is unlimited.
EnableDetailedErrors	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a service method. The default is `false`. Setting `EnableDetailedErrors` to `true` can leak sensitive information.
CompressionProviders	gzip	A collection of compression providers used to compress and decompress messages. Custom compression providers can be created and added to the collection. The default configured providers support **gzip** compression.
ResponseCompressionAlgorithm	`null`	The compression algorithm used to compress messages sent from the server. The algorithm must match a compression provider in `CompressionProviders`. For the algorithm to compress a response, the client must indicate it supports the algorithm by sending it in the **grpc-accept-encoding** header.
ResponseCompressionLevel	`null`	The compress level used to compress messages sent from the server.

OPTION	DEFAULT VALUE	DESCRIPTION
Interceptors	None	A collection of interceptors that are run with each gRPC call. Interceptors are run in the order they are registered. Globally configured interceptors are run before interceptors configured for a single service. For more information about gRPC interceptors, see gRPC Interceptors vs. Middleware.
IgnoreUnknownServices	`false`	If `true`, calls to unknown services and methods don't return an **UNIMPLEMENTED** status, and the request passes to the next registered middleware in ASP.NET Core.

Options can be configured for all services by providing an options delegate to the `AddGrpc` call in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.EnableDetailedErrors = true;
        options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
        options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
    });
}
```

Options for a single service override the global options provided in `AddGrpc` and can be configured using `AddServiceOptions<TService>`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc().AddServiceOptions<MyService>(options =>
    {
        options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
        options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
    });
}
```

Configure client options

gRPC client configuration is set on `GrpcChannelOptions`. The following table describes options for configuring gRPC channels:

OPTION	DEFAULT VALUE	DESCRIPTION

OPTION	DEFAULT VALUE	DESCRIPTION
HttpHandler	New instance	The `HttpMessageHandler` used to make gRPC calls. A client can be set to configure a custom `HttpClientHandler` or add additional handlers to the HTTP pipeline for gRPC calls. If no `HttpMessageHandler` is specified, a new `HttpClientHandler` instance is created for the channel with automatic disposal.
HttpClient	`null`	The `HttpClient` used to make gRPC calls. This setting is an alternative to `HttpHandler`.
DisposeHttpClient	`false`	If set to `true` and an `HttpMessageHandler` or `HttpClient` is specified, then either the `HttpHandler` or `HttpClient`, respectively, is disposed when the `GrpcChannel` is disposed.
LoggerFactory	`null`	The `LoggerFactory` used by the client to log information about gRPC calls. A `LoggerFactory` instance can be resolved from dependency injection or created using `LoggerFactory.Create`. For examples of configuring logging, see Logging and diagnostics in gRPC on .NET.
MaxSendMessageSize	`null`	The maximum message size in bytes that can be sent from the client. Attempting to send a message that exceeds the configured maximum message size results in an exception. When set to `null`, the message size is unlimited.
MaxReceiveMessageSize	4 MB	The maximum message size in bytes that can be received by the client. If the client receives a message that exceeds this limit, it throws an exception. Increasing this value allows the client to receive larger messages, but can negatively impact memory consumption. When set to `null`, the message size is unlimited.
Credentials	`null`	A `ChannelCredentials` instance. Credentials are used to add authentication metadata to gRPC calls.

OPTION	DEFAULT VALUE	DESCRIPTION
CompressionProviders	gzip	A collection of compression providers used to compress and decompress messages. Custom compression providers can be created and added to the collection. The default configured providers support **gzip** compression.

The following code:

- Sets the maximum send and receive message size on the channel.
- Creates a client.

```
static async Task Main(string[] args)
{
    var channel = GrpcChannel.ForAddress("https://localhost:5001", new GrpcChannelOptions
    {
        MaxReceiveMessageSize = 5 * 1024 * 1024, // 5 MB
        MaxSendMessageSize = 2 * 1024 * 1024 // 2 MB
    });
    var client = new Greeter.GreeterClient(channel);

    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
}
```

WARNING

ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this GitHub issue.

Additional resources

- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client
- Logging and diagnostics in gRPC on .NET
- Create a .NET Core gRPC client and server in ASP.NET Core

Authentication and authorization in gRPC for ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By James Newton-King

View or download sample code (how to download)

Authenticate users calling a gRPC service

gRPC can be used with ASP.NET Core authentication to associate a user with each call.

The following is an example of `Startup.Configure` which uses gRPC and ASP.NET Core authentication:

```
public void Configure(IApplicationBuilder app)
{
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGrpcService<GreeterService>();
    });
}
```

> **NOTE**
> The order in which you register the ASP.NET Core authentication middleware matters. Always call `UseAuthentication` and `UseAuthorization` after `UseRouting` and before `UseEndpoints`.

The authentication mechanism your app uses during a call needs to be configured. Authentication configuration is added in `Startup.ConfigureServices` and will be different depending upon the authentication mechanism your app uses. For examples of how to secure ASP.NET Core apps, see Authentication samples.

Once authentication has been setup, the user can be accessed in a gRPC service methods via the `ServerCallContext`.

```
public override Task<BuyTicketsResponse> BuyTickets(
    BuyTicketsRequest request, ServerCallContext context)
{
    var user = context.GetHttpContext().User;

    // ... access data from ClaimsPrincipal ...
}
```

Bearer token authentication

The client can provide an access token for authentication. The server validates the token and uses it to identify the user.

On the server, bearer token authentication is configured using the JWT Bearer middleware.

In the .NET gRPC client, the token can be sent with calls as a header:

```
public bool DoAuthenticatedCall(
    Ticketer.TicketerClient client, string token)
{
    var headers = new Metadata();
    headers.Add("Authorization", $"Bearer {token}");

    var request = new BuyTicketsRequest { Count = 1 };
    var response = await client.BuyTicketsAsync(request, headers);

    return response.Success;
}
```

Configuring `ChannelCredentials` on a channel is an alternative way to send the token to the service with gRPC calls. The credential is run each time a gRPC call is made, which avoids the need to write code in multiple places to pass the token yourself.

The credential in the following example configures the channel to send the token with every gRPC call:

```
private static GrpcChannel CreateAuthenticatedChannel(string address)
{
    var credentials = CallCredentials.FromInterceptor((context, metadata) =>
    {
        if (!string.IsNullOrEmpty(_token))
        {
            metadata.Add("Authorization", $"Bearer {_token}");
        }
        return Task.CompletedTask;
    });

    // SslCredentials is used here because this channel is using TLS.
    // CallCredentials can't be used with ChannelCredentials.Insecure on non-TLS channels.
    var channel = GrpcChannel.ForAddress(address, new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Create(new SslCredentials(), credentials)
    });
    return channel;
}
```

Client certificate authentication

A client could alternatively provide a client certificate for authentication. Certificate authentication happens at the TLS level, long before it ever gets to ASP.NET Core. When the request enters ASP.NET Core, the client certificate authentication package allows you to resolve the certificate to a `ClaimsPrincipal`.

> **NOTE**
>
> Configure the server to accept client certificates. For information on accepting client certificates in Kestrel, IIS, and Azure, see Configure certificate authentication in ASP.NET Core.

In the .NET gRPC client, the client certificate is added to `HttpClientHandler` that is then used to create the gRPC client:

```
public Ticketer.TicketerClient CreateClientWithCert(
    string baseAddress,
    X509Certificate2 certificate)
{
    // Add client cert to the handler
    var handler = new HttpClientHandler();
    handler.ClientCertificates.Add(certificate);

    // Create the gRPC channel
    var channel = GrpcChannel.ForAddress(baseAddress, new GrpcChannelOptions
    {
        HttpHandler = handler
    });

    return new Ticketer.TicketerClient(channel);
}
```

Other authentication mechanisms

Many ASP.NET Core supported authentication mechanisms work with gRPC:

- Azure Active Directory
- Client Certificate
- IdentityServer
- JWT Token
- OAuth 2.0
- OpenID Connect
- WS-Federation

For more information on configuring authentication on the server, see ASP.NET Core authentication.

Configuring the gRPC client to use authentication will depend on the authentication mechanism you are using. The previous bearer token and client certificate examples show a couple of ways the gRPC client can be configured to send authentication metadata with gRPC calls:

- Strongly typed gRPC clients use `HttpClient` internally. Authentication can be configured on HttpClientHandler, or by adding custom HttpMessageHandler instances to the `HttpClient`.
- Each gRPC call has an optional `CallOptions` argument. Custom headers can be sent using the option's headers collection.

> **NOTE**
> Windows Authentication (NTLM/Kerberos/Negotiate) can't be used with gRPC. gRPC requires HTTP/2, and HTTP/2 doesn't support Windows Authentication.

Authorize users to access services and service methods

By default, all methods in a service can be called by unauthenticated users. To require authentication, apply the `[Authorize]` attribute to the service:

```
[Authorize]
public class TicketerService : Ticketer.TicketerBase
{
}
```

You can use the constructor arguments and properties of the `[Authorize]` attribute to restrict access to only users

matching specific authorization policies. For example, if you have a custom authorization policy called `MyAuthorizationPolicy`, ensure that only users matching that policy can access the service using the following code:

```
[Authorize("MyAuthorizationPolicy")]
public class TicketerService : Ticketer.TicketerBase
{
}
```

Individual service methods can have the `[Authorize]` attribute applied as well. If the current user doesn't match the policies applied to **both** the method and the class, an error is returned to the caller:

```
[Authorize]
public class TicketerService : Ticketer.TicketerBase
{
    public override Task<AvailableTicketsResponse> GetAvailableTickets(
        Empty request, ServerCallContext context)
    {
        // ... buy tickets for the current user ...
    }

    [Authorize("Administrators")]
    public override Task<BuyTicketsResponse> RefundTickets(
        BuyTicketsRequest request, ServerCallContext context)
    {
        // ... refund tickets (something only Administrators can do) ..
    }
}
```

Additional resources

- Bearer Token authentication in ASP.NET Core
- Configure Client Certificate authentication in ASP.NET Core

Logging and diagnostics in gRPC on .NET

9/22/2020 • 7 minutes to read • Edit Online

By James Newton-King

This article provides guidance for gathering diagnostics from a gRPC app to help troubleshoot issues. Topics covered include:

- **Logging** - Structured logs written to .NET Core logging. ILogger is used by app frameworks to write logs, and by users for their own logging in an app.
- **Tracing** - Events related to an operation written using `DiaganosticSource` and `Activity`. Traces from diagnostic source are commonly used to collect app telemetry by libraries such as Application Insights and OpenTelemetry.
- **Metrics** - Representation of data measures over intervals of time, for example, requests per second. Metrics are emitted using `EventCounter` and can be observed using dotnet-counters command line tool or with Application Insights.

Logging

gRPC services and the gRPC client write logs using .NET Core logging. Logs are a good place to start when you need to debug unexpected behavior in your apps.

gRPC services logging

> **WARNING**
> Server-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

Since gRPC services are hosted on ASP.NET Core, it uses the ASP.NET Core logging system. In the default configuration, gRPC logs very little information, but this can configured. See the documentation on ASP.NET Core logging for details on configuring ASP.NET Core logging.

gRPC adds logs under the `Grpc` category. To enable detailed logs from gRPC, configure the `Grpc` prefixes to the `Debug` level in your *appsettings.json* file by adding the following items to the `LogLevel` sub-section in `Logging`:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Grpc": "Debug"
    }
  }
}
```

You can also configure this in *Startup.cs* with `ConfigureLogging`:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.AddFilter("Grpc", LogLevel.Debug);
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

If you aren't using JSON-based configuration, set the following configuration value in your configuration system:

- `Logging:LogLevel:Grpc` = `Debug`

Check the documentation for your configuration system to determine how to specify nested configuration values. For example, when using environment variables, two `_` characters are used instead of the `:` (for example, `Logging__LogLevel__Grpc`).

We recommend using the `Debug` level when gathering more detailed diagnostics for your app. The `Trace` level produces very low-level diagnostics and is rarely needed to diagnose issues in your app.

Sample logging output

Here is an example of console output at the `Debug` level of a gRPC service:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:5001/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
dbug: Grpc.AspNetCore.Server.ServerCallHandler[1]
      Reading message.
info: GrpcService.GreeterService[0]
      Hello World
dbug: Grpc.AspNetCore.Server.ServerCallHandler[6]
      Sending message.
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 1.4113ms 200 application/grpc
```

Access server-side logs

How you access server-side logs depends on the environment in which you're running.

As a console app

If you're running in a console app, the Console logger should be enabled by default. gRPC logs will appear in the console.

Other environments

If the app is deployed to another environment (for example, Docker, Kubernetes, or Windows Service), see Logging in .NET Core and ASP.NET Core for more information on how to configure logging providers suitable for the environment.

gRPC client logging

> **WARNING**
>
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

To get logs from the .NET client, you can set the `GrpcChannelOptions.LoggerFactory` property when the client's channel is created. If you are calling a gRPC service from an ASP.NET Core app then the logger factory can be resolved from dependency injection (DI):

```
[ApiController]
[Route("[controller]")]
public class GreetingController : ControllerBase
{
    private ILoggerFactory _loggerFactory;

    public GreetingController(ILoggerFactory loggerFactory)
    {
        _loggerFactory = loggerFactory;
    }

    [HttpGet]
    public async Task<ActionResult<string>> Get(string name)
    {
        var channel = GrpcChannel.ForAddress("https://localhost:5001",
            new GrpcChannelOptions { LoggerFactory = _loggerFactory });
        var client = new Greeter.GreeterClient(channel);

        var reply = await client.SayHelloAsync(new HelloRequest { Name = name });
        return Ok(reply.Message);
    }
}
```

An alternative way to enable client logging is to use the gRPC client factory to create the client. A gRPC client registered with the client factory and resolved from DI will automatically use the app's configured logging.

If your app isn't using DI then you can create a new `ILoggerFactory` instance with LoggerFactory.Create. To access this method add the Microsoft.Extensions.Logging package to your app.

```
var loggerFactory = LoggerFactory.Create(logging =>
{
    logging.AddConsole();
    logging.SetMinimumLevel(LogLevel.Debug);
});

var channel = GrpcChannel.ForAddress("https://localhost:5001",
    new GrpcChannelOptions { LoggerFactory = loggerFactory });

var client = Greeter.GreeterClient(channel);
```

gRPC client log scopes

The gRPC client adds a logging scope to logs made during a gRPC call. The scope has metadata related to the gRPC call:

- **GrpcMethodType** - The gRPC method type. Possible values are names from `Grpc.Core.MethodType` enum, e.g. Unary
- **GrpcUri** - The relative URI of the gRPC method, e.g. /greet.Greeter/SayHellos

Sample logging output

Here is an example of console output at the `Debug` level of a gRPC client:

```
dbug: Grpc.Net.Client.Internal.GrpcCall[1]
      Starting gRPC call. Method type: 'Unary', URI: 'https://localhost:5001/Greet.Greeter/SayHello'.
dbug: Grpc.Net.Client.Internal.GrpcCall[6]
      Sending message.
dbug: Grpc.Net.Client.Internal.GrpcCall[1]
      Reading message.
dbug: Grpc.Net.Client.Internal.GrpcCall[4]
      Finished gRPC call.
```

Tracing

gRPC services and the gRPC client provide information about gRPC calls using DiagnosticSource and Activity.

- .NET gRPC uses an activity to represent a gRPC call.
- Tracing events are written to the diagnostic source at the start and stop of the gRPC call activity.
- Tracing doesn't capture information about when messages are sent over the lifetime of gRPC streaming calls.

gRPC service tracing

gRPC services are hosted on ASP.NET Core which reports events about incoming HTTP requests. gRPC specific metadata is added to the existing HTTP request diagnostics that ASP.NET Core provides.

- Diagnostic source name is `Microsoft.AspNetCore`.
- Activity name is `Microsoft.AspNetCore.Hosting.HttpRequestIn`.
  - Name of the gRPC method invoked by the gRPC call is added as a tag with the name `grpc.method`.
  - Status code of the gRPC call when it is complete is added as a tag with the name `grpc.status_code`.

gRPC client tracing

The .NET gRPC client uses `HttpClient` to make gRPC calls. Although `HttpClient` writes diagnostic events, the .NET gRPC client provides a custom diagnostic source, activity and events so that complete information about a gRPC call can be collected.

- Diagnostic source name is `Grpc.Net.Client`.
- Activity name is `Grpc.Net.Client.GrpcOut`.
  - Name of the gRPC method invoked by the gRPC call is added as a tag with the name `grpc.method`.
  - Status code of the gRPC call when it is complete is added as a tag with the name `grpc.status_code`.

Collecting tracing

The easiest way to use `DiagnosticSource` is to configure a telemetry library such as Application Insights or OpenTelemetry in your app. The library will process information about gRPC calls along-side other app telemetry.

Tracing can be viewed in a managed service like Application Insights, or you can choose to run your own distributed tracing system. OpenTelemetry supports exporting tracing data to Jaeger and Zipkin.

`DiagnosticSource` can consume tracing events in code using `DiagnosticListener`. For information about listening to a diagnostic source with code, see the DiagnosticSource user's guide.

> **NOTE**
>
> Telemetry libraries do not capture gRPC specific `Grpc.Net.Client.GrpcOut` telemetry currently. Work to improve telemetry libraries capturing this tracing is ongoing.

Metrics

Metrics is a representation of data measures over intervals of time, for example, requests per second. Metrics data

allows observation of the state of an app at a high-level. .NET gRPC metrics are emitted using `EventCounter`.

gRPC service metrics

gRPC server metrics are reported on `Grpc.AspNetCore.Server` event source.

NAME	DESCRIPTION
`total-calls`	Total Calls
`current-calls`	Current Calls
`calls-failed`	Total Calls Failed
`calls-deadline-exceeded`	Total Calls Deadline Exceeded
`messages-sent`	Total Messages Sent
`messages-received`	Total Messages Received
`calls-unimplemented`	Total Calls Unimplemented

ASP.NET Core also provides its own metrics on `Microsoft.AspNetCore.Hosting` event source.

gRPC client metrics

gRPC client metrics are reported on `Grpc.Net.Client` event source.

NAME	DESCRIPTION
`total-calls`	Total Calls
`current-calls`	Current Calls
`calls-failed`	Total Calls Failed
`calls-deadline-exceeded`	Total Calls Deadline Exceeded
`messages-sent`	Total Messages Sent
`messages-received`	Total Messages Received

Observe metrics

dotnet-counters is a performance monitoring tool for ad-hoc health monitoring and first-level performance investigation. Monitor a .NET app with either `Grpc.AspNetCore.Server` or `Grpc.Net.Client` as the provider name.

```
> dotnet-counters monitor --process-id 1902 Grpc.AspNetCore.Server

Press p to pause, r to resume, q to quit.
    Status: Running
[Grpc.AspNetCore.Server]
    Total Calls                                 300
    Current Calls                               5
    Total Calls Failed                          0
    Total Calls Deadline Exceeded               0
    Total Messages Sent                         295
    Total Messages Received                     300
    Total Calls Unimplemented                   0
```

Another way to observe gRPC metrics is to capture counter data using Application Insights's Microsoft.ApplicationInsights.EventCounterCollector package. Once setup, Application Insights collects common .NET counters at runtime. gRPC's counters are not collected by default, but App Insights can be customized to include additional counters.

Specify the gRPC counters for Application Insight to collect in *Startup.cs*:

```
    using Microsoft.ApplicationInsights.Extensibility.EventCounterCollector;

    public void ConfigureServices(IServiceCollection services)
    {
        //... other code...

        services.ConfigureTelemetryModule<EventCounterCollectionModule>(
            (module, o) =>
            {
                // Configure App Insights to collect gRPC counters gRPC services hosted in an ASP.NET Core app
                module.Counters.Add(new EventCounterCollectionRequest("Grpc.AspNetCore.Server", "current-
calls"));
                module.Counters.Add(new EventCounterCollectionRequest("Grpc.AspNetCore.Server", "total-
calls"));
                module.Counters.Add(new EventCounterCollectionRequest("Grpc.AspNetCore.Server", "calls-
failed"));
            }
        );
    }
```

Additional resources

- Logging in .NET Core and ASP.NET Core
- gRPC for .NET configuration
- gRPC client factory integration in .NET Core

Security considerations in gRPC for ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By James Newton-King

This article provides information on securing gRPC with .NET Core.

Transport security

gRPC messages are sent and received using HTTP/2. We recommend:

- Transport Layer Security (TLS) be used to secure messages in production gRPC apps.
- gRPC services should only listen and respond over secured ports.

TLS is configured in Kestrel. For more information on configuring Kestrel endpoints, see Kestrel endpoint configuration.

Exceptions

Exception messages are generally considered sensitive data that shouldn't be revealed to a client. By default, gRPC doesn't send the details of an exception thrown by a gRPC service to the client. Instead, the client receives a generic message indicating an error occurred. Exception message delivery to the client can be overridden (for example, in development or test) with EnableDetailedErrors. Exception messages shouldn't be exposed to the client in production apps.

Message size limits

Incoming messages to gRPC clients and services are loaded into memory. Message size limits are a mechanism to help prevent gRPC from consuming excessive resources.

gRPC uses per-message size limits to manage incoming and outgoing messages. By default, gRPC limits incoming messages to 4 MB. There is no limit on outgoing messages.

On the server, gRPC message limits can be configured for all services in an app with `AddGrpc`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.MaxReceiveMessageSize = 1 * 1024 * 1024; // 1 MB
        options.MaxSendMessageSize = 1 * 1024 * 1024; // 1 MB
    });
}
```

Limits can also be configured for an individual service using `AddServiceOptions<TService>`. For more information on configuring message size limits, see gRPC configuration.

Client certificate validation

Client certificates are initially validated when the connection is established. By default, Kestrel doesn't perform additional validation of a connection's client certificate.

We recommend that gRPC services secured by client certificates use the

Microsoft.AspNetCore.Authentication.Certificate package. ASP.NET Core certification authentication will perform additional validation on a client certificate, including:

- Certificate has a valid extended key use (EKU)
- Is within its validity period
- Check certificate revocation

Performance best practices with gRPC

9/22/2020 • 6 minutes to read • Edit Online

By James Newton-King

gRPC is designed for high-performance services. This document explains how to get the best performance possible from gRPC.

Reuse gRPC channels

A gRPC channel should be reused when making gRPC calls. Reusing a channel allows calls to be multiplexed through an existing HTTP/2 connection.

If a new channel is created for each gRPC call then the amount of time it takes to complete can increase significantly. Each call will require multiple network round-trips between the client and the server to create a new HTTP/2 connection:

1. Opening a socket
2. Establishing TCP connection
3. Negotiating TLS
4. Starting HTTP/2 connection
5. Making the gRPC call

Channels are safe to share and reuse between gRPC calls:

- gRPC clients are created with channels. gRPC clients are lightweight objects and don't need to be cached or reused.
- Multiple gRPC clients can be created from a channel, including different types of clients.
- A channel and clients created from the channel can safely be used by multiple threads.
- Clients created from the channel can make multiple simultaneous calls.

gRPC client factory offers a centralized way to configure channels. It automatically reuses underlying channels. For more information, see gRPC client factory integration in .NET Core.

Connection concurrency

HTTP/2 connections typically have a limit on the number of maximum concurrent streams (active HTTP requests) on a connection at one time. By default, most servers set this limit to 100 concurrent streams.

A gRPC channel uses a single HTTP/2 connection, and concurrent calls are multiplexed on that connection. When the number of active calls reaches the connection stream limit, additional calls are queued in the client. Queued calls wait for active calls to complete before they are sent. Applications with high load, or long running streaming gRPC calls, could see performance issues caused by calls queuing because of this limit.

.NET 5 introduces the `SocketsHttpHandler.EnableMultipleHttp2Connections` property. When set to `true`, additional HTTP/2 connections are created by a channel when the concurrent stream limit is reached. When a `GrpcChannel` is created its internal `SocketsHttpHandler` is automatically configured to create additional HTTP/2 connections. If an app configures its own handler, consider setting `EnableMultipleHttp2Connections` to `true`:

```
var channel = GrpcChannel.ForAddress("https://localhost", new GrpcChannelOptions
{
    HttpHandler = new SocketsHttpHandler
    {
        EnableMultipleHttp2Connections = true,

        // ...configure other handler settings
    }
});
```

There are a couple of workarounds for .NET Core 3.1 apps:

- Create separate gRPC channels for areas of the app with high load. For example, the `Logger` gRPC service might have a high load. Use a separate channel to create the `LoggerClient` in the app.
- Use a pool of gRPC channels, for example, create a list of gRPC channels. `Random` is used to pick a channel from the list each time a gRPC channel is needed. Using `Random` randomly distributes calls over multiple connections.

> **IMPORTANT**
>
> Increasing the maximum concurrent stream limit on the server is another way to solve this problem. In Kestrel this is configured with MaxStreamsPerConnection.
>
> Increasing the maximum concurrent stream limit is not recommended. Too many streams on a single HTTP/2 connection introduces new performance issues:
>
> - Thread contention between streams trying to write to the connection.
> - Connection packet loss causes all calls to be blocked at the TCP layer.

Load balancing

Some load balancers don't work effectively with gRPC. L4 (transport) load balancers operate at a connection level, by distributing TCP connections across endpoints. This approach works well for loading balancing API calls made with HTTP/1.1. Concurrent calls made with HTTP/1.1 are sent on different connections, allowing calls to be load balanced across endpoints.

Because L4 load balancers operate at a connection level, they don't work well with gRPC. gRPC uses HTTP/2, which multiplexes multiple calls on a single TCP connection. All gRPC calls over that connection go to one endpoint.

There are two options to effectively load balance gRPC:

- Client-side load balancing
- L7 (application) proxy load balancing

> **NOTE**
>
> Only gRPC calls can be load balanced between endpoints. Once a streaming gRPC call is established, all messages sent over the stream go to one endpoint.

Client-side load balancing

With client-side load balancing, the client knows about endpoints. For each gRPC call it selects a different endpoint to send the call to. Client-side load balancing is a good choice when latency is important. There is no proxy between the client and the service so the call is sent to the service directly. The downside to client-side load balancing is that each client must keep track of available endpoints it should use.

Lookaside client load balancing is a technique where load balancing state is stored in a central location. Clients

periodically query the central location for information to use when making load balancing decisions.

`Grpc.Net.Client` currently doesn't support client-side load balancing. Grpc.Core is a good choice if client-side load balancing is required in .NET.

Proxy load balancing

An L7 (application) proxy works at a higher level than an L4 (transport) proxy. L7 proxies understand HTTP/2, and are able to distribute gRPC calls multiplexed to the proxy on one HTTP/2 connection across multiple endpoints. Using a proxy is simpler than client-side load balancing, but can add extra latency to gRPC calls.

There are many L7 proxies available. Some options are:

- Envoy - A popular open source proxy.
- Linkerd - Service mesh for Kubernetes.
- YARP: A Reverse Proxy - A preview open source proxy written in .NET.

Inter-process communication

gRPC calls between a client and service are usually sent over TCP sockets. TCP is great for communicating across a network, but inter-process communication (IPC) is more efficient when the client and service are on the same machine.

Consider using a transport like Unix domain sockets or named pipes for gRPC calls between processes on the same machine. For more information, see Inter-process communication with gRPC.

Keep alive pings

Keep alive pings can be used to keep HTTP/2 connections alive during periods of inactivity. Having an existing HTTP/2 connection ready when an app resumes activity allows for the initial gRPC calls to be made quickly, without a delay caused by the connection being reestablished.

Keep alive pings are configured on SocketsHttpHandler:

```
var handler = new SocketsHttpHandler
{
    PooledConnectionIdleTimeout = Timeout.InfiniteTimeSpan,
    KeepAlivePingDelay = TimeSpan.FromSeconds(60),
    KeepAlivePingTimeout = TimeSpan.FromSeconds(30),
    EnableMultipleHttp2Connections = true
};

var channel = GrpcChannel.ForAddress("https://localhost:5001", new GrpcChannelOptions
{
    HttpHandler = handler
});
```

The preceding code configures a channel that sends a keep alive ping to the server every 60 seconds during periods of inactivity. The ping ensures the server and any proxies in use won't close the connection because of inactivity.

Streaming

gRPC bidirectional streaming can be used to replace unary gRPC calls in high-performance scenarios. Once a bidirectional stream has started, streaming messages back and forth is faster than sending messages with multiple unary gRPC calls. Streamed messages are sent as data on an existing HTTP/2 request and eliminates the overhead of creating a new HTTP/2 request for each unary call.

Example service:

```
public override async Task SayHello(IAsyncStreamReader<HelloRequest> requestStream,
    IServerStreamWriter<HelloReply> responseStream, ServerCallContext context)
{
    await foreach (var request in requestStream.ReadAllAsync())
    {
        var helloReply = new HelloReply { Message = "Hello " + request.Name };

        await responseStream.WriteAsync(helloReply);
    }
}
```

Example client:

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHello();

Console.WriteLine("Type a name then press enter.");
while (true)
{
    var text = Console.ReadLine();

    // Send and receive messages over the stream
    await call.RequestStream.WriteAsync(new HelloRequest { Name = text });
    await call.ResponseStream.MoveNext();

    Console.WriteLine($"Greeting: {call.ResponseStream.Current.Message}");
}
```

Replacing unary calls with bidirectional streaming for performance reasons is an advanced technique and is not appropriate in many situations.

Using streaming calls is a good choice when:

1. High throughput or low latency is required.
2. gRPC and HTTP/2 are identified as a performance bottleneck.
3. A worker in the client is sending or receiving regular messages with a gRPC service.

Be aware of the additional complexity and limitations of using streaming calls instead of unary:

1. A stream can be interrupted by a service or connection error. Logic is required to restart stream if there is an error.
2. `RequestStream.WriteAsync` is not safe for multi-threading. Only one message can be written to a stream at a time. Sending messages from multiple threads over a single stream requires a producer/consumer queue like Channel<T> to marshall messages.
3. A gRPC streaming method is limited to receiving one type of message and sending one type of message. For example, `rpc StreamingCall(stream RequestMessage) returns (stream ResponseMessage)` receives `RequestMessage` and sends `ResponseMessage`. Protobuf's support for unknown or conditional messages using `Any` and `oneof` can work around this limitation.

Inter-process communication with gRPC

9/22/2020 • 2 minutes to read • Edit Online

By James Newton-King

gRPC calls between a client and service are usually sent over TCP sockets. TCP was designed for communicating across a network. Inter-process communication (IPC) is more efficient than TCP when the client and service are on the same machine. This document explains how to use gRPC with custom transports in IPC scenarios.

Server configuration

Custom transports are supported by Kestrel. Kestrel is configured in *Program.cs*:

```
public static readonly string SocketPath = Path.Combine(Path.GetTempPath(), "socket.tmp");

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
            webBuilder.ConfigureKestrel(options =>
            {
                if (File.Exists(SocketPath))
                {
                    File.Delete(SocketPath);
                }
                options.ListenUnixSocket(SocketPath);
            });
        });
```

The preceding example:

- Configures Kestrel's endpoints in `ConfigureKestrel`.
- Calls ListenUnixSocket to listen to a Unix domain socket (UDS) with the specified path.

Kestrel has built-in support for UDS endpoints. UDS are supported on Linux, macOS and modern versions of Windows.

Client configuration

`GrpcChannel` supports making gRPC calls over custom transports. When a channel is created, it can be configured with a `SocketsHttpHandler` that has a custom `ConnectCallback`. The callback allows the client to make connections over custom transports and then send HTTP requests over that transport.

> **IMPORTANT**
>
> `SocketsHttpHandler.ConnectCallback` is a new API in .NET 5 release candidate 2.

Unix domain sockets connection factory example:

```
public class UnixDomainSocketConnectionFactory
{
    private readonly EndPoint _endPoint;

    public UnixDomainSocketConnectionFactory(EndPoint endPoint)
    {
        _endPoint = endPoint;
    }

    public async ValueTask<Stream> ConnectAsync(SocketsHttpConnectionContext _,
        CancellationToken cancellationToken = default)
    {
        var socket = new Socket(AddressFamily.Unix, SocketType.Stream, ProtocolType.Unspecified);

        try
        {
            await socket.ConnectAsync(_endPoint, cancellationToken).ConfigureAwait(false);
            return new NetworkStream(socket, true);
        }
        catch
        {
            socket.Dispose();
            throw;
        }
    }
}
```

Using the custom connection factory to create a channel:

```
public static readonly string SocketPath = Path.Combine(Path.GetTempPath(), "socket.tmp");

public static GrpcChannel CreateChannel()
{
    var udsEndPoint = new UnixDomainSocketEndPoint(SocketPath);
    var connectionFactory = new UnixDomainSocketConnectionFactory(udsEndPoint);
    var socketsHttpHandler = new SocketsHttpHandler
    {
        ConnectCallback = connectionFactory.ConnectAsync
    };

    return GrpcChannel.ForAddress("http://localhost", new GrpcChannelOptions
    {
        HttpHandler = socketsHttpHandler
    });
}
```

Channels created using the preceding code send gRPC calls over Unix domain sockets. Support for other IPC technologies can be implemented using the extensibility in Kestrel and `SocketsHttpHandler`.

Create JSON Web APIs from gRPC

9/22/2020 • 3 minutes to read • Edit Online

By James Newton-King

> **IMPORTANT**
>
> gRPC HTTP API is an experimental project, not a committed product. We want to:
>
> - Test that our approach to creating JSON Web APIs for gRPC services works.
> - Get feedback on if this approach is useful to .NET developers.
>
> Please leave feedback to ensure we build something that developers like and are productive with.

gRPC is a modern way to communicate between apps. gRPC uses HTTP/2, streaming, Protobuf and message contracts to create high-performance, real-time services.

One limitation with gRPC is not every platform can use it. Browsers don't fully support HTTP/2, making REST and JSON the primary way to get data into browser apps. Even with the benefits that gRPC brings, REST and JSON have an important place in modern apps. Building gRPC ***and*** JSON Web APIs adds unwanted overhead to app development.

This document discusses how to create JSON Web APIs using gRPC services.

gRPC HTTP API

gRPC HTTP API is an experimental extension for ASP.NET Core that creates RESTful JSON APIs for gRPC services. Once configured, gRPC HTTP API allows apps to call gRPC services with familiar HTTP concepts:

- HTTP verbs
- URL parameter binding
- JSON requests/responses

gRPC can still be used to call services.

Usage

1. Add a package reference to Microsoft.AspNetCore.Grpc.HttpApi.
2. Register services in *Startup.cs* with `AddGrpcHttpApi`.
3. Add google/api/http.proto and google/api/annotations.proto files to your project.
4. Annotate gRPC methods in your *.proto* files with HTTP bindings and routes:

```
syntax = "proto3";

import "google/api/annotations.proto";

package greet;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "v1/greeter/{name}"
    };
  }
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

The `SayHello` gRPC method can now be invoked as gRPC+Protobuf and as an HTTP API:

- Request: `HTTP/1.1 GET /v1/greeter/world`
- Response: `{ "message": "Hello world" }`

Server logs show that the HTTP call is executed by a gRPC service. gRPC HTTP API maps the incoming HTTP request to a gRPC message, and then converts the response message to JSON.

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:5001/v1/greeter/world
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - v1/greeter/{name}'
info: Server.GreeterService[0]
      Sending hello to world
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - v1/greeter/{name}'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 1.996ms 200 application/json
```

This is a basic example. See HttpRule for more customization options.

gRPC HTTP API vs gRPC-Web

Both gRPC HTTP API and gRPC-Web allow gRPC services to be called from a browser. However, the way each does this is different:

- gRPC-Web lets browser apps call gRPC services from the browser with the gRPC-Web client and Protobuf. gRPC-Web requires the browser app generate a gRPC client, and has the advantage of sending small, fast Protobuf messages.
- gRPC HTTP API allows browser apps to call gRPC services as if they were RESTful APIs with JSON. The browser app doesn't need to generate a gRPC client or know anything about gRPC.

No generated client is created for gRPC HTTP API. The previous `Greeter` service can be called using browser JavaScript APIs:

```
var name = nameInput.value;

fetch("/v1/greeter/" + name).then(function (response) {
  response.json().then(function (data) {
    console.log("Result: " + data.message);
  });
});
```

Experimental status

gRPC HTTP API is an experiment. It is not complete and it is not supported. We're interested in this technology, and the ability it gives app developers to quickly create gRPC and JSON services at the same time. There is no commitment to completing the gRPC HTTP API.

We want to gauge developer interest in gRPC HTTP API. If gRPC HTTP API is interesting to you then please give feedback.

grpc-gateway

grpc-gateway is another technology for creating RESTful JSON APIs from gRPC services. It uses the same *.proto* annotations to map HTTP concepts to gRPC services.

The biggest difference between grpc-gateway and gRPC HTTP API is grpc-gateway uses code generation to create a reverse-proxy server. The reverse-proxy translates RESTful calls into gRPC and then sends them on to the gRPC service.

For installation and usage of grpc-gateway, see the grpc-gateway documentation.

Additional resources

- google.api.HttpRule documentation
- Use gRPC in browser apps

Manage Protobuf references with dotnet-grpc

9/22/2020 • 4 minutes to read • Edit Online

By John Luo

`dotnet-grpc` is a .NET Core Global Tool for managing Protobuf (*.proto*) references within a .NET gRPC project. The tool can be used to add, refresh, remove, and list Protobuf references.

Installation

To install the `dotnet-grpc` .NET Core Global Tool, run the following command:

```
dotnet tool install -g dotnet-grpc
```

Add references

`dotnet-grpc` can be used to add Protobuf references as `<Protobuf />` items to the *.csproj* file:

```
<Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
```

The Protobuf references are used to generate the C# client and/or server assets. The `dotnet-grpc` tool can:

- Create a Protobuf reference from local files on disk.
- Create a Protobuf reference from a remote file specified by a URL.
- Ensure the correct gRPC package dependencies are added to the project.

For example, the `Grpc.AspNetCore` package is added to a web app. `Grpc.AspNetCore` contains gRPC server and client libraries and tooling support. Alternatively, the `Grpc.Net.Client`, `Grpc.Tools` and `Google.Protobuf` packages, which contain only the gRPC client libraries and tooling support, are added to a Console app.

Add file

The `add-file` command is used to add local files on disk as Protobuf references. The file paths provided:

- Can be relative to the current directory or absolute paths.
- May contain wild cards for pattern-based file globbing.

If any files are outside the project directory, a `Link` element is added to display the file under the folder `Protos` in Visual Studio.

Usage

```
dotnet grpc add-file [options] <files>...
```

Arguments

ARGUMENT	DESCRIPTION
files	The protobuf file references. These can be a path to glob for local protobuf files.

Options

SHORT OPTION	LONG OPTION	DESCRIPTION
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.
-s	--services	The type of gRPC services that should be generated. If `Default` is specified, `Both` is used for Web projects and `Client` is used for non-Web projects. Accepted values are `Both`, `Client`, `Default`, `None`, `Server`.
-i	--additional-import-dirs	Additional directories to be used when resolving imports for the protobuf files. This is a semicolon separated list of paths.
	--access	The access modifier to use for the generated C# classes. The default value is `Public`. Accepted values are `Internal` and `Public`.

Add URL

The `add-url` command is used to add a remote file specified by an source URL as Protobuf reference. A file path must be provided to specify where to download the remote file. The file path can be relative to the current directory or an absolute path. If the file path is outside the project directory, a `Link` element is added to display the file under the virtual folder `Protos` in Visual Studio.

Usage

```
dotnet-grpc add-url [options] <url>
```

Arguments

ARGUMENT	DESCRIPTION
url	The URL to a remote protobuf file.

Options

SHORT OPTION	LONG OPTION	DESCRIPTION
-o	--output	Specifies the download path for the remote protobuf file. This is a required option.
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.

SHORT OPTION	LONG OPTION	DESCRIPTION
-s	--services	The type of gRPC services that should be generated. If `Default` is specified, `Both` is used for Web projects and `Client` is used for non-Web projects. Accepted values are `Both`, `Client`, `Default`, `None`, `Server`.
-i	--additional-import-dirs	Additional directories to be used when resolving imports for the protobuf files. This is a semicolon separated list of paths.
	--access	The access modifier to use for the generated C# classes. Default value is `Public`. Accepted values are `Internal` and `Public`.

Remove

The `remove` command is used to remove Protobuf references from the *.csproj* file. The command accepts path arguments and source URLs as arguments. The tool:

- Only removes the Protobuf reference.
- Does not delete the *.proto* file, even if it was originally downloaded from a remote URL.

Usage

```
dotnet-grpc remove [options] <references>...
```

Arguments

ARGUMENT	DESCRIPTION
references	The URLs or file paths of the protobuf references to remove.

Options

SHORT OPTION	LONG OPTION	DESCRIPTION
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.

Refresh

The `refresh` command is used to update a remote reference with the latest content from the source URL. Both the download file path and the source URL can be used to specify the reference to be updated. Note:

- The hashes of the file contents are compared to determine whether the local file should be updated.
- No timestamp information is compared.

The tool always replaces the local file with the remote file if an update is needed.

Usage

```
dotnet-grpc refresh [options] [<references>...]
```

Arguments

ARGUMENT	DESCRIPTION
references	The URLs or file paths to remote protobuf references that should be updated. Leave this argument empty to refresh all remote references.

Options

SHORT OPTION	LONG OPTION	DESCRIPTION
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.
	--dry-run	Outputs a list of files that would be updated without downloading any new content.

List

The `list` command is used to display all the Protobuf references in the project file. If all values of a column are default values, the column may be omitted.

Usage

```
dotnet-grpc list [options]
```

Options

SHORT OPTION	LONG OPTION	DESCRIPTION
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.

Additional resources

- Introduction to gRPC on .NET Core
- gRPC services with C#
- gRPC services with ASP.NET Core

Test gRPC services with gRPCurl in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By James Newton-King

Tooling is available for gRPC that allows developers to test services without building client apps:

- gRPCurl is a command-line tool that provides interaction with gRPC services.
- gRPCui builds on top of gRPCurl and adds an interactive web UI for gRPC, similar to tools such as Postman and Swagger UI.

This article discusses how to:

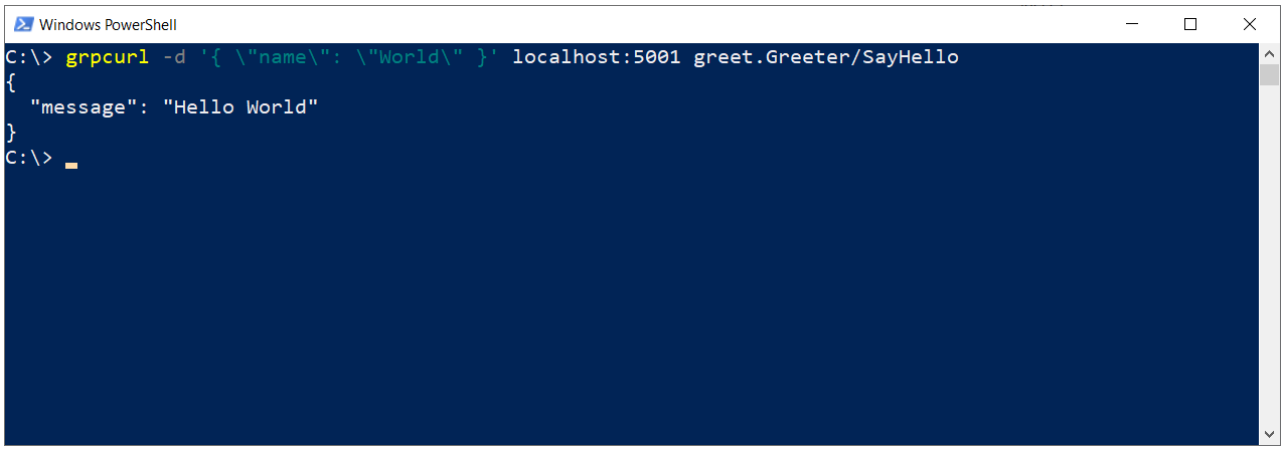
- Download and install gRPCurl and gRPCui.
- Set up gRPC reflection with a gRPC ASP.NET Core app.
- Discover and test gRPC services with `grpcurl`.
- Interact with gRPC services via a browser using `grpcui`.

About gRPCurl

gRPCurl is a command-line tool created by the gRPC community. Its features include:

- Calling gRPC services, including streaming services.
- Service discovery using gRPC reflection.
- Listing and describing gRPC services.
- Works with secure (TLS) and insecure (plain-text) servers.

For information about downloading and installing `grpcurl`, see the gRPCurl GitHub homepage.



Set up gRPC reflection

`grpcurl` must know the Protobuf contract of services before it can call them. There are two ways to do this:

- Set up gRPC reflection on the server. gRPCurl automatically discovers service contracts.
- Specify `.proto` files in command-line arguments to gRPCurl.

It's easier to use gRPCurl with gRPC reflection. gRPC reflection adds a new gRPC service to the app that clients can call to discover services.

gRPC ASP.NET Core has built-in support for gRPC reflection with the `Grpc.AspNetCore.Server.Reflection` package. To configure reflection in an app:

- Add a `Grpc.AspNetCore.Server.Reflection` package reference.
- Register reflection in `Startup.cs`:
  - `AddGrpcReflection` to register services that enable reflection.
  - `MapGrpcReflectionService` to add a reflection service endpoint.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();
    services.AddGrpcReflection();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGrpcService<GreeterService>();

        if (env.IsDevelopment())
        {
            endpoints.MapGrpcReflectionService();
        }
    });
}
```

When gRPC reflection is set up:

- A gRPC reflection service is added to the server app.
- Client apps that support gRPC reflection can call the reflection service to discover services hosted by the server.
- gRPC services are still called from the client. Reflection only enables service discovery and doesn't bypass server-side security. Endpoints protected by authentication and authorization require the caller to pass credentials for the endpoint to be called successfully.

Use `grpcurl`

The `-help` argument explains `grpcurl` command-line options:

```
$ grpcurl -help
```

Discover services

Use the `describe` verb to view the services defined by the server:

```
$ grpcurl localhost:5001 describe
greet.Greeter is a service:
service Greeter {
  rpc SayHello ( .greet.HelloRequest ) returns ( .greet.HelloReply );
  rpc SayHellos ( .greet.HelloRequest ) returns ( stream .greet.HelloReply );
}
grpc.reflection.v1alpha.ServerReflection is a service:
service ServerReflection {
  rpc ServerReflectionInfo ( stream .grpc.reflection.v1alpha.ServerReflectionRequest ) returns ( stream
.grpc.reflection.v1alpha.ServerReflectionResponse );
}
```

The preceding example:

- Runs the `describe` verb on server `localhost:5001`.
- Prints services and methods returned by gRPC reflection.
  - `Greeter` is a service implemented by the app.
  - `ServerReflection` is the service added by the `Grpc.AspNetCore.Server.Reflection` package.

Combine `describe` with a service, method, or message name to view its detail:

```
$ grpcurl localhost:5001 describe greet.HelloRequest
greet.HelloRequest is a message:
message HelloRequest {
  string name = 1;
}
```

Call gRPC services

Call a gRPC service by specifying a service and method name along with a JSON argument that represents the request message. The JSON is converted into Protobuf and sent to the service.

```
$ grpcurl -d '{ \"name\": \"World\" }' localhost:5001 greet.Greeter/SayHello
{
  "message": "Hello World"
}
```

In the preceding example:

- The `-d` argument specifies a request message with JSON. This argument must come before the server address and method name.
- Calls the `SayHello` method on the `greeter.Greeter` service.
- Prints the response message as JSON.

About gRPCui

gRPCui is an interactive web UI for gRPC. It builds on top of gRPCurl and offers a GUI for discovering and testing gRPC services, similar to HTTP tools such as Postman or Swagger UI.

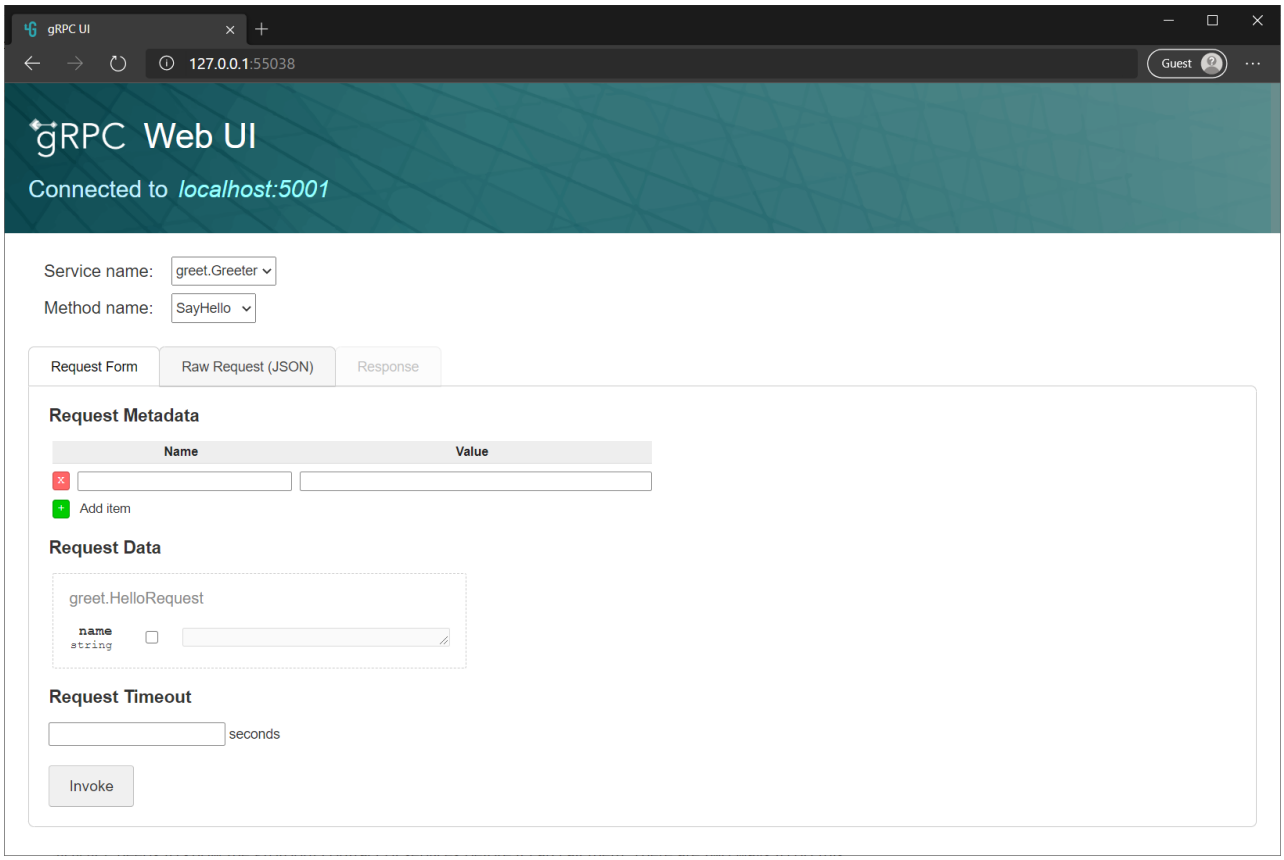
For information about downloading and installing `grpcui`, see the gRPCui GitHub homepage.

Using `grpcui`

Run `grpcui` with the server address to interact with as an argument:

```
$ grpcui localhost:5001
gRPC Web UI available at http://127.0.0.1:55038/
```

The tool launches a browser window with the interactive web UI. gRPC services are automatically discovered using gRPC reflection.



Additional resources

- gRPCurl GitHub homepage
- gRPCui GitHub homepage
- `Grpc.AspNetCore.Server.Reflection`

Migrating gRPC services from C-core to ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By John Luo

Due to the implementation of the underlying stack, not all features work in the same way between C-core-based gRPC apps and ASP.NET Core-based apps. This document highlights the key differences for migrating between the two stacks.

gRPC service implementation lifetime

In the ASP.NET Core stack, gRPC services, by default, are created with a scoped lifetime. In contrast, gRPC C-core by default binds to a service with a singleton lifetime.

A scoped lifetime allows the service implementation to resolve other services with scoped lifetimes. For example, a scoped lifetime can also resolve `DbContext` from the DI container through constructor injection. Using scoped lifetime:

- A new instance of the service implementation is constructed for each request.
- It isn't possible to share state between requests via instance members on the implementation type.
- The expectation is to store shared states in a singleton service in the DI container. The stored shared states are resolved in the constructor of the gRPC service implementation.

For more information on service lifetimes, see Dependency injection in ASP.NET Core.

Add a singleton service

To facilitate the transition from a gRPC C-core implementation to ASP.NET Core, it's possible to change the service lifetime of the service implementation from scoped to singleton. This involves adding an instance of the service implementation to the DI container:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();
    services.AddSingleton(new GreeterService());
}
```

However, a service implementation with a singleton lifetime is no longer able to resolve scoped services through constructor injection.

Configure gRPC services options

In C-core-based apps, settings such as `grpc.max_receive_message_length` and `grpc.max_send_message_length` are configured with `ChannelOption` when constructing the Server instance.

In ASP.NET Core, gRPC provides configuration through the `GrpcServiceOptions` type. For example, a gRPC service's the maximum incoming message size can be configured via `AddGrpc`. The following example changes the default `MaxReceiveMessageSize` of 4 MB to 16 MB:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.MaxReceiveMessageSize = 16 * 1024 * 1024; // 16 MB
    });
}
```

For more information on configuration, see gRPC for .NET configuration.

Logging

C-core-based apps rely on the `GrpcEnvironment` to configure the logger for debugging purposes. The ASP.NET Core stack provides this functionality through the Logging API. For example, a logger can be added to the gRPC service via constructor injection:

```
public class GreeterService : Greeter.GreeterBase
{
    public GreeterService(ILogger<GreeterService> logger)
    {
    }
}
```

HTTPS

C-core-based apps configure HTTPS through the Server.Ports property. A similar concept is used to configure servers in ASP.NET Core. For example, Kestrel uses endpoint configuration for this functionality.

gRPC Interceptors vs Middleware

ASP.NET Core middleware offers similar functionalities compared to interceptors in C-core-based gRPC apps. ASP.NET Core middleware and interceptors are conceptually similar. Both:

- Are used to construct a pipeline that handles a gRPC request.
- Allow work to be performed before or after the next component in the pipeline.
- Provide access to `HttpContext`:
  - In middleware the `HttpContext` is a parameter.
  - In interceptors the `HttpContext` can be accessed using the `ServerCallContext` parameter with the `ServerCallContext.GetHttpContext` extension method. Note that this feature is specific to interceptors running in ASP.NET Core.

gRPC Interceptor differences from ASP.NET Core Middleware:

- Interceptors:
  - Operate on the gRPC layer of abstraction using the ServerCallContext.
  - Provide access to:
    - The deserialized message sent to a call.
    - The message being returned from the call before it is serialized.
  - Can catch and handle exceptions thrown from gRPC services.
- Middleware:
  - Runs before gRPC interceptors.
  - Operates on the underlying HTTP/2 messages.
  - Can only access bytes from the request and response streams.

Additional resources

- Introduction to gRPC on .NET Core
- gRPC services with C#
- gRPC services with ASP.NET Core
- Create a .NET Core gRPC client and server in ASP.NET Core

gRPC for Windows Communication Foundation (WCF) developers

9/22/2020 • 3 minutes to read • Edit Online

This article provides a summary of why ASP.NET Core gRPC is a good fit for Windows Communication Foundation (WCF) developers who want to migrate to modern architectures and platforms.

Comparison to WCF

Although the implementation and approach are different for gRPC, the experience of developing and consuming services with gRPC should be intuitive for WCF developers. WCF and gRPC are RPC (remote procedure call) frameworks with the same goals:

- Make it possible to code as though the client and server are on the same platform.
- Provide a simplified portable networking API.

Both platforms share the requirement of declaring and implementing an interface, although the process for declaring the interface is different. The many types of RPC calls that gRPC supports map well to the bindings available to WCF services. For more information and examples, see Migrate a WCF solution to gRPC.

Benefits of gRPC

gRPC provides a better framework than other approaches for the following reasons.

Performance

gRPC uses HTTP/2. In contrast to HTTP/1.1, HTTP/2:

- Is a smaller, faster binary protocol.
- Is more efficient for computers to parse.
- Supports multiplexing requests over a single connection. Multiplexing enables multiple requests to be sent over one connection without requests blocking each other. In HTTP/1.1, the blocking is known as "head-of-line (HOL) blocking."

gRPC uses Protobuf, an efficient binary format, to serialize messages. Protobuf messages are:

- Fast to serialize and deserialize.
- Use less bandwidth than text-based formats.

gRPC is a good solution for mobile devices and networks with bandwidth restrictions.

Interoperability

There are gRPC tools and libraries for all major programming languages and platforms, including .NET, Java, Python, Go, C++, Node.js, Swift, Dart, Ruby, and PHP. Thanks to the Protobuf binary wire format and the efficient code generation for each platform, developers can build cross-platform, performant apps.

Usability and productivity

gRPC is a comprehensive RPC solution. It works consistently across multiple languages and platforms. It also provides excellent tooling, with much of the boilerplate code automatically generated. Like WCF, gRPC automatically generates messages and a strongly typed client. Developer time is freed up to focus on business logic.

Streaming

gRPC has full bidirectional streaming, which provides similar functionality to WCF's full duplex services. gRPC streaming can operate over regular internet connections, load balancers, and service meshes.

Deadlines, timeouts, and cancellation

gRPC allows clients to specify a maximum time for an RPC to finish. If the specified deadline is exceeded, the server can cancel the operation independently of the client. Deadlines and cancellations can be propagated through subsequent gRPC calls to help enforce resource usage limits. Clients can stop operations when a deadline is exceeded, or earlier if necessary. For example, clients can stop operations because of a user interaction.

Security

gRPC can use TLS and HTTP/2 to provide an end-to-end encrypted connection between the client and the server. Support for client certificate authentication further increases security and trust between client and server.

gRPC as a migration path for WCF to .NET Core and .NET 5

.NET Core and .NET 5 marks a shift in the way that Microsoft delivers remote communication solutions to developers who want to deliver services across a range of platforms. .NET Core and .NET 5 support calling WCF services, but won't offer server-side support for hosting WCF.

There are two recommended paths for modernizing WCF apps:

- gRPC is built on modern technologies and has emerged as the most popular choice across the developer community for RPC apps. Starting with .NET Core 3.0, modern .NET platforms have excellent support for gRPC. Migrating WCF services to use gRPC helps provide the RPC features, performance, an interoperability needed in modern apps.
- CoreWCF is a community effort to bring support for hosting WCF services to .NET Core and .NET 5. A preview release is available and the project is working towards being production ready. CoreWCF only supports a subset of WCF's features, and .NET Framework apps that migrate to use it will need code changes and testing to be successful. CoreWCF is a good choice if an app has to maintain compatibility with existing clients that call WCF services.

Get started

For detailed guidance on building gRPC services in ASP.NET Core for WCF developers, see ASP.NET Core gRPC for WCF Developers

Compare gRPC services with HTTP APIs

9/22/2020 • 5 minutes to read • Edit Online

By James Newton-King

This article explains how gRPC services compare to HTTP APIs with JSON (including ASP.NET Core web APIs). The technology used to provide an API for your app is an important choice, and gRPC offers unique benefits compared to HTTP APIs. This article discusses the strengths and weaknesses of gRPC and recommends scenarios for using gRPC over other technologies.

High-level comparison

The following table offers a high-level comparison of features between gRPC and HTTP APIs with JSON.

FEATURE	GRPC	HTTP APIS WITH JSON
Contract	Required (*.proto*)	Optional (OpenAPI)
Protocol	HTTP/2	HTTP
Payload	Protobuf (small, binary)	JSON (large, human readable)
Prescriptiveness	Strict specification	Loose. Any HTTP is valid.
Streaming	Client, server, bi-directional	Client, server
Browser support	No (requires grpc-web)	Yes
Security	Transport (TLS)	Transport (TLS)
Client code-generation	Yes	OpenAPI + third-party tooling

gRPC strengths

Performance

gRPC messages are serialized using Protobuf, an efficient binary message format. Protobuf serializes very quickly on the server and client. Protobuf serialization results in small message payloads, important in limited bandwidth scenarios like mobile apps.

gRPC is designed for HTTP/2, a major revision of HTTP that provides significant performance benefits over HTTP 1.x:

- Binary framing and compression. HTTP/2 protocol is compact and efficient both in sending and receiving.
- Multiplexing of multiple HTTP/2 calls over a single TCP connection. Multiplexing eliminates head-of-line blocking.

HTTP/2 is not exclusive to gRPC. Many request types, including HTTP APIs with JSON, can use HTTP/2 and benefit from its performance improvements.

Code generation

All gRPC frameworks provide first-class support for code generation. A core file to gRPC development is the .proto file, which defines the contract of gRPC services and messages. From this file gRPC frameworks will code generate a service base class, messages, and a complete client.

By sharing the *.proto* file between the server and client, messages and client code can be generated from end to end. Code generation of the client eliminates duplication of messages on the client and server, and creates a strongly-typed client for you. Not having to write a client saves significant development time in applications with many services.

Strict specification

A formal specification for HTTP API with JSON doesn't exist. Developers debate the best format of URLs, HTTP verbs, and response codes.

The gRPC specification is prescriptive about the format a gRPC service must follow. gRPC eliminates debate and saves developer time because gRPC is consistent across platforms and implementations.

Streaming

HTTP/2 provides a foundation for long-lived, real-time communication streams. gRPC provides first-class support for streaming through HTTP/2.

A gRPC service supports all streaming combinations:

- Unary (no streaming)
- Server to client streaming
- Client to server streaming
- Bi-directional streaming

Deadline/timeouts and cancellation

gRPC allows clients to specify how long they are willing to wait for an RPC to complete. The deadline is sent to the server, and the server can decide what action to take if it exceeds the deadline. For example, the server might cancel in-progress gRPC/HTTP/database requests on timeout.

Propagating the deadline and cancellation through child gRPC calls helps enforce resource usage limits.

gRPC recommended scenarios

gRPC is well suited to the following scenarios:

- **Microservices**: gRPC is designed for low latency and high throughput communication. gRPC is great for lightweight microservices where efficiency is critical.
- **Point-to-point real-time communication**: gRPC has excellent support for bi-directional streaming. gRPC services can push messages in real-time without polling.
- **Polyglot environments**: gRPC tooling supports all popular development languages, making gRPC a good choice for multi-language environments.
- **Network constrained environments**: gRPC messages are serialized with Protobuf, a lightweight message format. A gRPC message is always smaller than an equivalent JSON message.
- **Inter-process communication (IPC)**: IPC transports such as Unix domain sockets and named pipes can be used with gRPC to communicate between apps on the same machine. For more information, see Inter-process communication with gRPC.

gRPC weaknesses

Limited browser support

It's impossible to directly call a gRPC service from a browser today. gRPC heavily uses HTTP/2 features and no browser provides the level of control required over web requests to support a gRPC client. For example, browsers

do not allow a caller to require that HTTP/2 be used, or provide access to underlying HTTP/2 frames.

There are two common approaches to bring gRPC to browser apps:

- gRPC-Web is an additional technology from the gRPC team that provides gRPC support in the browser. gRPC-Web allows browser apps to benefit from the high-performance and low network usage of gRPC. Not all of gRPC's features are supported by gRPC-Web. Client and bi-directional streaming isn't supported, and there is limited support for server streaming.

  .NET Core has support for gRPC-Web. For more information, see Use gRPC in browser apps.

- RESTful JSON Web APIs can be automatically created from gRPC services by annotating the *.proto* file with HTTP metadata. This allows an app to support both gRPC and JSON web APIs, without duplicating effort of building separate services for both.

  .NET Core has experimental support for creating JSON web APIs from gRPC services. For more information, see Create JSON Web APIs from gRPC.

Not human readable

HTTP API requests are sent as text and can be read and created by humans.

gRPC messages are encoded with Protobuf by default. While Protobuf is efficient to send and receive, its binary format isn't human readable. Protobuf requires the message's interface description specified in the *.proto* file to properly deserialize. Additional tooling is required to analyze Protobuf payloads on the wire and to compose requests by hand.

Features such as server reflection and the gRPC command line tool exist to assist with binary Protobuf messages. Also, Protobuf messages support conversion to and from JSON. The built-in JSON conversion provides an efficient way to convert Protobuf messages to and from human readable form when debugging.

Alternative framework scenarios

Other frameworks are recommended over gRPC in the following scenarios:

- **Browser accessible APIs**: gRPC isn't fully supported in the browser. gRPC-Web can offer browser support, but it has limitations and introduces a server proxy.
- **Broadcast real-time communication**: gRPC supports real-time communication via streaming, but the concept of broadcasting a message out to registered connections doesn't exist. For example in a chat room scenario where new chat messages should be sent to all clients in the chat room, each gRPC call is required to individually stream new chat messages to the client. SignalR is a useful framework for this scenario. SignalR has the concept of persistent connections and built-in support for broadcasting messages.

Additional resources

- Create a .NET Core gRPC client and server in ASP.NET Core
- Introduction to gRPC on .NET Core
- gRPC services with C#
- Migrating gRPC services from C-core to ASP.NET Core

Troubleshoot gRPC on .NET Core

9/22/2020 • 4 minutes to read • Edit Online

By James Newton-King

This document discusses commonly encountered problems when developing gRPC apps on .NET.

Mismatch between client and service SSL/TLS configuration

The gRPC template and samples use Transport Layer Security (TLS) to secure gRPC services by default. gRPC clients need to use a secure connection to call secured gRPC services successfully.

You can verify the ASP.NET Core gRPC service is using TLS in the logs written on app start. The service will be listening on an HTTPS endpoint:

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

The .NET Core client must use `https` in the server address to make calls with a secured connection:

```
static async Task Main(string[] args)
{
    // The port number(5001) must match the port of the gRPC server.
    var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var client = new Greet.GreeterClient(channel);
}
```

All gRPC client implementations support TLS. gRPC clients from other languages typically require the channel configured with `SslCredentials`. `SslCredentials` specifies the certificate that the client will use, and it must be used instead of insecure credentials. For examples of configuring the different gRPC client implementations to use TLS, see gRPC Authentication.

Call a gRPC service with an untrusted/invalid certificate

The .NET gRPC client requires the service to have a trusted certificate. The following error message is returned when calling a gRPC service without a trusted certificate:

> Unhandled exception. System.Net.Http.HttpRequestException: The SSL connection could not be established, see inner exception. ---> System.Security.Authentication.AuthenticationException: The remote certificate is invalid according to the validation procedure.

You may see this error if you are testing your app locally and the ASP.NET Core HTTPS development certificate is not trusted. For instructions to fix this issue, see Trust the ASP.NET Core HTTPS development certificate on Windows and macOS.

If you are calling a gRPC service on another machine and are unable to trust the certificate then the gRPC client can be configured to ignore the invalid certificate. The following code uses HttpClientHandler.ServerCertificateCustomValidationCallback to allow calls without a trusted certificate:

```
var httpHandler = new HttpClientHandler();
// Return `true` to allow certificates that are untrusted/invalid
httpHandler.ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

var channel = GrpcChannel.ForAddress("https://localhost:5001",
    new GrpcChannelOptions { HttpHandler = httpHandler });
var client = new Greet.GreeterClient(channel);
```

WARNING

Untrusted certificates should only be used during app development. Production apps should always use valid certificates.

Call insecure gRPC services with .NET Core client

Additional configuration is required to call insecure gRPC services with the .NET Core client. The gRPC client must set the `System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport` switch to `true` and use `http` in the server address:

```
// This switch must be set before creating the GrpcChannel/HttpClient.
AppContext.SetSwitch(
    "System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);

// The port number(5000) must match the port of the gRPC server.
var channel = GrpcChannel.ForAddress("http://localhost:5000");
var client = new Greet.GreeterClient(channel);
```

Unable to start ASP.NET Core gRPC app on macOS

Kestrel doesn't support HTTP/2 with TLS on macOS and older Windows versions such as Windows 7. The ASP.NET Core gRPC template and samples use TLS by default. You'll see the following error message when you attempt to start the gRPC server:

> Unable to bind to https://localhost:5001 on the IPv4 loopback interface: 'HTTP/2 over TLS is not supported on macOS due to missing ALPN support.'.

To work around this issue, configure Kestrel and the gRPC client to use HTTP/2 *without* TLS. You should only do this during development. Not using TLS will result in gRPC messages being sent without encryption.

Kestrel must configure an HTTP/2 endpoint without TLS in *Program.cs*:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(options =>
            {
                // Setup a HTTP/2 endpoint without TLS.
                options.ListenLocalhost(5000, o => o.Protocols =
                    HttpProtocols.Http2);
            });
            webBuilder.UseStartup<Startup>();
        });
```

When an HTTP/2 endpoint is configured without TLS, the endpoint's ListenOptions.Protocols must be set to

`HttpProtocols.Http2`. `HttpProtocols.Http1AndHttp2` can't be used because TLS is required to negotiate HTTP/2. Without TLS, all connections to the endpoint default to HTTP/1.1, and gRPC calls fail.

The gRPC client must also be configured to not use TLS. For more information, see Call insecure gRPC services with .NET Core client.

> **WARNING**
> HTTP/2 without TLS should only be used during app development. Production apps should always use transport security. For more information, see Security considerations in gRPC for ASP.NET Core.

gRPC C# assets are not code generated from .proto files

gRPC code generation of concrete clients and service base classes requires protobuf files and tooling to be referenced from a project. You must include:

- *.proto* files you want to use in the `<Protobuf>` item group. Imported *.proto* files must be referenced by the project.
- Package reference to the gRPC tooling package Grpc.Tools.

For more information on generating gRPC C# assets, see gRPC services with C#.

An ASP.NET Core web app hosting gRPC services only needs the service base class generated:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

A gRPC client app making gRPC calls only needs the concrete client generated:

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

WPF projects unable to generate gRPC C# assets from .proto files

WPF projects have a known issue that prevents gRPC code generation from working correctly. Any gRPC types generated in a WPF project by referencing `Grpc.Tools` and *.proto* files will create compilation errors when used:

> error CS0246: The type or namespace name 'MyGrpcServices' could not be found (are you missing a using directive or an assembly reference?)

You can workaround this issue by:

1. Create a new .NET Core class library project.
2. In the new project, add references to enable C# code generation from **.proto* files:
   - Add a package reference to Grpc.Tools package.
   - Add **.proto* files to the `<Protobuf>` item group.
3. In the WPF application, add a reference to the new project.

The WPF application can use the gRPC generated types from the new class library project.

WARNING

ASP.NET Core gRPC is not currently supported on Azure App Service or IIS. The HTTP/2 implementation of Http.Sys does not support HTTP response trailing headers which gRPC relies on. For more information, see this GitHub issue.

Razor Pages unit tests in ASP.NET Core

9/22/2020 • 19 minutes to read • Edit Online

ASP.NET Core supports unit tests of Razor Pages apps. Tests of the data access layer (DAL) and page models help ensure:

- Parts of a Razor Pages app work independently and together as a unit during app construction.
- Classes and methods have limited scopes of responsibility.
- Additional documentation exists on how the app should behave.
- Regressions, which are errors brought about by updates to the code, are found during automated building and deployment.

This topic assumes that you have a basic understanding of Razor Pages apps and unit tests. If you're unfamiliar with Razor Pages apps or test concepts, see the following topics:

- Introduction to Razor Pages in ASP.NET Core
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Unit testing C# in .NET Core using dotnet test and xUnit

View or download sample code (how to download)

The sample project is composed of two apps:

APP	PROJECT FOLDER	DESCRIPTION
Message app	*src/RazorPagesTestSample*	Allows a user to add a message, delete one message, delete all messages, and analyze messages (find the average number of words per message).
Test app	*tests/RazorPagesTestSample.Tests*	Used to unit test the DAL and Index page model of the message app.

The tests can be run using the built-in test features of an IDE, such as Visual Studio or Visual Studio for Mac. If using Visual Studio Code or the command line, execute the following command at a command prompt in the *tests/RazorPagesTestSample.Tests* folder:

```
dotnet test
```

Message app organization

The message app is a Razor Pages message system with the following characteristics:

- The Index page of the app (*Pages/Index.cshtml* and *Pages/Index.cshtml.cs*) provides a UI and page model methods to control the addition, deletion, and analysis of messages (find the average number of words per message).
- A message is described by the `Message` class (*Data/Message.cs*) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.
- The app contains a DAL in its database context class, `AppDbContext` (*Data/AppDbContext.cs*). The DAL methods

are marked `virtual`, which allows mocking the methods for use in the tests.

- If the database is empty on app startup, the message store is initialized with three messages. These *seeded messages* are also used in tests.

†The EF topic, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the sample app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic in ASP.NET Core (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the *tests/RazorPagesTestSample.Tests* folder.

TEST APP FOLDER	DESCRIPTION
UnitTests	• *DataAccessLayerTest.cs* contains the unit tests for the DAL. • *IndexPageTests.cs* contains the unit tests for the Index page model.
Utilities	Contains the `TestDbContextOptions` method used to create new database context options for each DAL unit test so that the database is reset to its baseline condition for each test.

The test framework is xUnit. The object mocking framework is Moq.

Unit tests of the data access layer (DAL)

The message app has a DAL with four methods contained in the `AppDbContext` class (*src/RazorPagesTestSample/Data/AppDbContext.cs*). Each method has one or two unit tests in the test app.

DAL METHOD	FUNCTION
`GetMessagesAsync`	Obtains a `List<Message>` from the database sorted by the `Text` property.
`AddMessageAsync`	Adds a `Message` to the database.
`DeleteAllMessagesAsync`	Deletes all `Message` entries from the database.
`DeleteMessageAsync`	Deletes a single `Message` from the database by `Id`.

Unit tests of the DAL require DbContextOptions when creating a new `AppDbContext` for each test. One approach to creating the `DbContextOptions` for each test is to use a DbContextOptionsBuilder:

```
var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase("InMemoryDb");

using (var db = new AppDbContext(optionsBuilder.Options))
{
    // Use the db here in the unit test.
}
```

The problem with this approach is that each test receives the database in whatever state the previous test left it. This can be problematic when trying to write atomic unit tests that don't interfere with each other. To force the `AppDbContext` to use a new database context for each test, supply a `DbContextOptions` instance that's based on a new service provider. The test app shows how to do this using its `Utilities` class method `TestDbContextOptions` (*tests/RazorPagesTestSample.Tests/Utilities/Utilities.cs*):

```
public static DbContextOptions<AppDbContext> TestDbContextOptions()
{
    // Create a new service provider to create a new in-memory database.
    var serviceProvider = new ServiceCollection()
        .AddEntityFrameworkInMemoryDatabase()
        .BuildServiceProvider();

    // Create a new options instance using an in-memory database and
    // IServiceProvider that the context should resolve all of its
    // services from.
    var builder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb")
        .UseInternalServiceProvider(serviceProvider);

    return builder.Options;
}
```

Using the `DbContextOptions` in the DAL unit tests allows each test to run atomically with a fresh database instance:

```
using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
{
    // Use the db here in the unit test.
}
```

Each test method in the `DataAccessLayerTest` class (*UnitTests/DataAccessLayerTest.cs*) follows a similar Arrange-Act-Assert pattern:

1. Arrange: The database is configured for the test and/or the expected outcome is defined.
2. Act: The test is executed.
3. Assert: Assertions are made to determine if the test result is a success.

For example, the `DeleteMessageAsync` method is responsible for removing a single message identified by its `Id` (*src/RazorPagesTestSample/Data/AppDbContext.cs*):

```
public async virtual Task DeleteMessageAsync(int id)
{
    var message = await Messages.FindAsync(id);

    if (message != null)
    {
        Messages.Remove(message);
        await SaveChangesAsync();
    }
}
```

There are two tests for this method. One test checks that the method deletes a message when the message is present in the database. The other method tests that the database doesn't change if the message `Id` for deletion doesn't exist. The `DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound` method is shown below:

```
[Fact]
public async Task DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var seedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(seedMessages);
        await db.SaveChangesAsync();
        var recId = 1;
        var expectedMessages =
            seedMessages.Where(message => message.Id != recId).ToList();

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

First, the method performs the Arrange step, where preparation for the Act step takes place. The seeding messages are obtained and held in `seedMessages`. The seeding messages are saved into the database. The message with an `Id` of `1` is set for deletion. When the `DeleteMessageAsync` method is executed, the expected messages should have all of the messages except for the one with an `Id` of `1`. The `expectedMessages` variable represents this expected outcome.

```
// Arrange
var seedMessages = AppDbContext.GetSeedingMessages();
await db.AddRangeAsync(seedMessages);
await db.SaveChangesAsync();
var recId = 1;
var expectedMessages =
    seedMessages.Where(message => message.Id != recId).ToList();
```

The method acts: The `DeleteMessageAsync` method is executed passing in the `recId` of `1`:

```
// Act
await db.DeleteMessageAsync(recId);
```

Finally, the method obtains the `Messages` from the context and compares it to the `expectedMessages` asserting that the two are equal:

```
// Assert
var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

In order to compare that the two `List<Message>` are the same:

- The messages are ordered by `Id`.
- Message pairs are compared on the `Text` property.

A similar test method, `DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound` checks the result of attempting to delete a message that doesn't exist. In this case, the expected messages in the database should be equal to the actual messages after the `DeleteMessageAsync` method is executed. There should be no change to the database's content:

```
[Fact]
public async Task DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var expectedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(expectedMessages);
        await db.SaveChangesAsync();
        var recId = 4;

        // Act
        try
        {
            await db.DeleteMessageAsync(recId);
        }
        catch
        {
            // recId doesn't exist
        }

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

Unit tests of the page model methods

Another set of unit tests is responsible for tests of page model methods. In the message app, the Index page models are found in the `IndexModel` class in *src/RazorPagesTestSample/Pages/Index.cshtml.cs*.

PAGE MODEL METHOD	FUNCTION
`OnGetAsync`	Obtains the messages from the DAL for the UI using the `GetMessagesAsync` method.
`OnPostAddMessageAsync`	If the ModelState is valid, calls `AddMessageAsync` to add a message to the database.
`OnPostDeleteAllMessagesAsync`	Calls `DeleteAllMessagesAsync` to delete all of the messages in the database.
`OnPostDeleteMessageAsync`	Executes `DeleteMessageAsync` to delete a message with the `Id` specified.
`OnPostAnalyzeMessagesAsync`	If one or more messages are in the database, calculates the average number of words per message.

The page model methods are tested using seven tests in the `IndexPageTests` class (*tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs*). The tests use the familiar Arrange-Assert-Act pattern. These tests focus on:

- Determining if the methods follow the correct behavior when the ModelState is invalid.
- Confirming the methods produce the correct IActionResult.
- Checking that property value assignments are made correctly.

This group of tests often mock the methods of the DAL to produce expected data for the Act step where a page model method is executed. For example, the `GetMessagesAsync` method of the `AppDbContext` is mocked to produce output. When a page model method executes this method, the mock returns the result. The data doesn't come from the database. This creates predictable, reliable test conditions for using the DAL in the page model tests.

The `OnGetAsync_PopulatesThePageModel_WithAListOfMessages` test shows how the `GetMessagesAsync` method is mocked for the page model:

```
var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
var expectedMessages = AppDbContext.GetSeedingMessages();
mockAppDbContext.Setup(
    db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
var pageModel = new IndexModel(mockAppDbContext.Object);
```

When the `OnGetAsync` method is executed in the Act step, it calls the page model's `GetMessagesAsync` method.

Unit test Act step (*tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs*):

```
// Act
await pageModel.OnGetAsync();
```

`IndexPage` page model's `OnGetAsync` method (*src/RazorPagesTestSample/Pages/Index.cshtml.cs*):

```
public async Task OnGetAsync()
{
    Messages = await _db.GetMessagesAsync();
}
```

The `GetMessagesAsync` method in the DAL doesn't return the result for this method call. The mocked version of the method returns the result.

In the `Assert` step, the actual messages (`actualMessages`) are assigned from the `Messages` property of the page model. A type check is also performed when the messages are assigned. The expected and actual messages are compared by their `Text` properties. The test asserts that the two `List<Message>` instances contain the same messages.

```
// Assert
var actualMessages = Assert.IsAssignableFrom<List<Message>>(pageModel.Messages);
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

Other tests in this group create page model objects that include the DefaultHttpContext, the ModelStateDictionary, an ActionContext to establish the `PageContext`, a `ViewDataDictionary`, and a `PageContext`. These are useful in conducting tests. For example, the message app establishes a `ModelState` error with AddModelError to check that a valid PageResult is returned when `OnPostAddMessageAsync` is executed:

```
[Fact]
public async Task OnPostAddMessageAsync_ReturnsAPageResult_WhenModelStateIsInvalid()
{
    // Arrange
    var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb");
    var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
    var expectedMessages = AppDbContext.GetSeedingMessages();
    mockAppDbContext.Setup(db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
    var httpContext = new DefaultHttpContext();
    var modelState = new ModelStateDictionary();
    var actionContext = new ActionContext(httpContext, new RouteData(), new PageActionDescriptor(),
modelState);
    var modelMetadataProvider = new EmptyModelMetadataProvider();
    var viewData = new ViewDataDictionary(modelMetadataProvider, modelState);
    var tempData = new TempDataDictionary(httpContext, Mock.Of<ITempDataProvider>());
    var pageContext = new PageContext(actionContext)
    {
        ViewData = viewData
    };
    var pageModel = new IndexModel(mockAppDbContext.Object)
    {
        PageContext = pageContext,
        TempData = tempData,
        Url = new UrlHelper(actionContext)
    };
    pageModel.ModelState.AddModelError("Message.Text", "The Text field is required.");

    // Act
    var result = await pageModel.OnPostAddMessageAsync();

    // Assert
    Assert.IsType<PageResult>(result);
}
```

Additional resources

- Unit testing C# in .NET Core using dotnet test and xUnit
- Test controller logic in ASP.NET Core
- Unit Test Your Code (Visual Studio)
- Integration tests in ASP.NET Core
- xUnit.net
- Building a complete .NET Core solution on macOS using Visual Studio for Mac
- Getting started with xUnit.net: Using .NET Core with the .NET SDK command line
- Moq
- Moq Quickstart

ASP.NET Core supports unit tests of Razor Pages apps. Tests of the data access layer (DAL) and page models help ensure:

- Parts of a Razor Pages app work independently and together as a unit during app construction.
- Classes and methods have limited scopes of responsibility.
- Additional documentation exists on how the app should behave.
- Regressions, which are errors brought about by updates to the code, are found during automated building and deployment.

This topic assumes that you have a basic understanding of Razor Pages apps and unit tests. If you're unfamiliar with Razor Pages apps or test concepts, see the following topics:

- Introduction to Razor Pages in ASP.NET Core
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Unit testing C# in .NET Core using dotnet test and xUnit

View or download sample code (how to download)

The sample project is composed of two apps:

APP	PROJECT FOLDER	DESCRIPTION
Message app	*src/RazorPagesTestSample*	Allows a user to add a message, delete one message, delete all messages, and analyze messages (find the average number of words per message).
Test app	*tests/RazorPagesTestSample.Tests*	Used to unit test the DAL and Index page model of the message app.

The tests can be run using the built-in test features of an IDE, such as Visual Studio or Visual Studio for Mac. If using Visual Studio Code or the command line, execute the following command at a command prompt in the *tests/RazorPagesTestSample.Tests* folder:

```
dotnet test
```

Message app organization

The message app is a Razor Pages message system with the following characteristics:

- The Index page of the app (*Pages/Index.cshtml* and *Pages/Index.cshtml.cs*) provides a UI and page model methods to control the addition, deletion, and analysis of messages (find the average number of words per message).
- A message is described by the `Message` class (*Data/Message.cs*) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.
- The app contains a DAL in its database context class, `AppDbContext` (*Data/AppDbContext.cs*). The DAL methods are marked `virtual`, which allows mocking the methods for use in the tests.
- If the database is empty on app startup, the message store is initialized with three messages. These *seeded messages* are also used in tests.

†The EF topic, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the sample app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic in ASP.NET Core (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the *tests/RazorPagesTestSample.Tests* folder.

TEST APP FOLDER	DESCRIPTION
UnitTests	• *DataAccessLayerTest.cs* contains the unit tests for the DAL. • *IndexPageTests.cs* contains the unit tests for the Index page model.
Utilities	Contains the `TestDbContextOptions` method used to create new database context options for each DAL unit test so that the database is reset to its baseline condition for each test.

The test framework is xUnit. The object mocking framework is Moq.

Unit tests of the data access layer (DAL)

The message app has a DAL with four methods contained in the `AppDbContext` class (*src/RazorPagesTestSample/Data/AppDbContext.cs*). Each method has one or two unit tests in the test app.

DAL METHOD	FUNCTION
`GetMessagesAsync`	Obtains a `List<Message>` from the database sorted by the `Text` property.
`AddMessageAsync`	Adds a `Message` to the database.
`DeleteAllMessagesAsync`	Deletes all `Message` entries from the database.
`DeleteMessageAsync`	Deletes a single `Message` from the database by `Id`.

Unit tests of the DAL require DbContextOptions when creating a new `AppDbContext` for each test. One approach to creating the `DbContextOptions` for each test is to use a DbContextOptionsBuilder:

```
var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase("InMemoryDb");

using (var db = new AppDbContext(optionsBuilder.Options))
{
    // Use the db here in the unit test.
}
```

The problem with this approach is that each test receives the database in whatever state the previous test left it. This can be problematic when trying to write atomic unit tests that don't interfere with each other. To force the `AppDbContext` to use a new database context for each test, supply a `DbContextOptions` instance that's based on a new service provider. The test app shows how to do this using its `Utilities` class method `TestDbContextOptions` (*tests/RazorPagesTestSample.Tests/Utilities/Utilities.cs*):

```
public static DbContextOptions<AppDbContext> TestDbContextOptions()
{
    // Create a new service provider to create a new in-memory database.
    var serviceProvider = new ServiceCollection()
        .AddEntityFrameworkInMemoryDatabase()
        .BuildServiceProvider();

    // Create a new options instance using an in-memory database and
    // IServiceProvider that the context should resolve all of its
    // services from.
    var builder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb")
        .UseInternalServiceProvider(serviceProvider);

    return builder.Options;
}
```

Using the `DbContextOptions` in the DAL unit tests allows each test to run atomically with a fresh database instance:

```
using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
{
    // Use the db here in the unit test.
}
```

Each test method in the `DataAccessLayerTest` class (*UnitTests/DataAccessLayerTest.cs*) follows a similar Arrange-Act-Assert pattern:

1. Arrange: The database is configured for the test and/or the expected outcome is defined.
2. Act: The test is executed.
3. Assert: Assertions are made to determine if the test result is a success.

For example, the `DeleteMessageAsync` method is responsible for removing a single message identified by its `Id` (*src/RazorPagesTestSample/Data/AppDbContext.cs*):

```
public async virtual Task DeleteMessageAsync(int id)
{
    var message = await Messages.FindAsync(id);

    if (message != null)
    {
        Messages.Remove(message);
        await SaveChangesAsync();
    }
}
```

There are two tests for this method. One test checks that the method deletes a message when the message is present in the database. The other method tests that the database doesn't change if the message `Id` for deletion doesn't exist. The `DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound` method is shown below:

```
[Fact]
public async Task DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var seedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(seedMessages);
        await db.SaveChangesAsync();
        var recId = 1;
        var expectedMessages =
            seedMessages.Where(message => message.Id != recId).ToList();

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

First, the method performs the Arrange step, where preparation for the Act step takes place. The seeding messages are obtained and held in `seedMessages`. The seeding messages are saved into the database. The message with an `Id` of `1` is set for deletion. When the `DeleteMessageAsync` method is executed, the expected messages should have all of the messages except for the one with an `Id` of `1`. The `expectedMessages` variable represents this expected outcome.

```
// Arrange
var seedMessages = AppDbContext.GetSeedingMessages();
await db.AddRangeAsync(seedMessages);
await db.SaveChangesAsync();
var recId = 1;
var expectedMessages =
    seedMessages.Where(message => message.Id != recId).ToList();
```

The method acts: The `DeleteMessageAsync` method is executed passing in the `recId` of `1`:

```
// Act
await db.DeleteMessageAsync(recId);
```

Finally, the method obtains the `Messages` from the context and compares it to the `expectedMessages` asserting that the two are equal:

```
// Assert
var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

In order to compare that the two `List<Message>` are the same:

- The messages are ordered by `Id`.
- Message pairs are compared on the `Text` property.

A similar test method, `DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound` checks the result of

attempting to delete a message that doesn't exist. In this case, the expected messages in the database should be equal to the actual messages after the `DeleteMessageAsync` method is executed. There should be no change to the database's content:

```
[Fact]
public async Task DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var expectedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(expectedMessages);
        await db.SaveChangesAsync();
        var recId = 4;

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

Unit tests of the page model methods

Another set of unit tests is responsible for tests of page model methods. In the message app, the Index page models are found in the `IndexModel` class in *src/RazorPagesTestSample/Pages/Index.cshtml.cs*.

PAGE MODEL METHOD	FUNCTION
`OnGetAsync`	Obtains the messages from the DAL for the UI using the `GetMessagesAsync` method.
`OnPostAddMessageAsync`	If the ModelState is valid, calls `AddMessageAsync` to add a message to the database.
`OnPostDeleteAllMessagesAsync`	Calls `DeleteAllMessagesAsync` to delete all of the messages in the database.
`OnPostDeleteMessageAsync`	Executes `DeleteMessageAsync` to delete a message with the `Id` specified.
`OnPostAnalyzeMessagesAsync`	If one or more messages are in the database, calculates the average number of words per message.

The page model methods are tested using seven tests in the `IndexPageTests` class (*tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs*). The tests use the familiar Arrange-Assert-Act pattern. These tests focus on:

- Determining if the methods follow the correct behavior when the ModelState is invalid.
- Confirming the methods produce the correct IActionResult.
- Checking that property value assignments are made correctly.

This group of tests often mock the methods of the DAL to produce expected data for the Act step where a page

model method is executed. For example, the `GetMessagesAsync` method of the `AppDbContext` is mocked to produce output. When a page model method executes this method, the mock returns the result. The data doesn't come from the database. This creates predictable, reliable test conditions for using the DAL in the page model tests.

The `OnGetAsync_PopulatesThePageModel_WithAListOfMessages` test shows how the `GetMessagesAsync` method is mocked for the page model:

```
var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
var expectedMessages = AppDbContext.GetSeedingMessages();
mockAppDbContext.Setup(
    db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
var pageModel = new IndexModel(mockAppDbContext.Object);
```

When the `OnGetAsync` method is executed in the Act step, it calls the page model's `GetMessagesAsync` method.

Unit test Act step (*tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs*):

```
// Act
await pageModel.OnGetAsync();
```

`IndexPage` page model's `OnGetAsync` method (*src/RazorPagesTestSample/Pages/Index.cshtml.cs*):

```
public async Task OnGetAsync()
{
    Messages = await _db.GetMessagesAsync();
}
```

The `GetMessagesAsync` method in the DAL doesn't return the result for this method call. The mocked version of the method returns the result.

In the `Assert` step, the actual messages (`actualMessages`) are assigned from the `Messages` property of the page model. A type check is also performed when the messages are assigned. The expected and actual messages are compared by their `Text` properties. The test asserts that the two `List<Message>` instances contain the same messages.

```
// Assert
var actualMessages = Assert.IsAssignableFrom<List<Message>>(pageModel.Messages);
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

Other tests in this group create page model objects that include the DefaultHttpContext, the ModelStateDictionary, an ActionContext to establish the `PageContext`, a `ViewDataDictionary`, and a `PageContext`. These are useful in conducting tests. For example, the message app establishes a `ModelState` error with AddModelError to check that a valid PageResult is returned when `OnPostAddMessageAsync` is executed:

```
[Fact]
public async Task OnPostAddMessageAsync_ReturnsAPageResult_WhenModelStateIsInvalid()
{
    // Arrange
    var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb");
    var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
    var expectedMessages = AppDbContext.GetSeedingMessages();
    mockAppDbContext.Setup(db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
    var httpContext = new DefaultHttpContext();
    var modelState = new ModelStateDictionary();
    var actionContext = new ActionContext(httpContext, new RouteData(), new PageActionDescriptor(),
modelState);
    var modelMetadataProvider = new EmptyModelMetadataProvider();
    var viewData = new ViewDataDictionary(modelMetadataProvider, modelState);
    var tempData = new TempDataDictionary(httpContext, Mock.Of<ITempDataProvider>());
    var pageContext = new PageContext(actionContext)
    {
        ViewData = viewData
    };
    var pageModel = new IndexModel(mockAppDbContext.Object)
    {
        PageContext = pageContext,
        TempData = tempData,
        Url = new UrlHelper(actionContext)
    };
    pageModel.ModelState.AddModelError("Message.Text", "The Text field is required.");

    // Act
    var result = await pageModel.OnPostAddMessageAsync();

    // Assert
    Assert.IsType<PageResult>(result);
}
```

Additional resources

- Unit testing C# in .NET Core using dotnet test and xUnit
- Test controller logic in ASP.NET Core
- Unit Test Your Code (Visual Studio)
- Integration tests in ASP.NET Core
- xUnit.net
- Building a complete .NET Core solution on macOS using Visual Studio for Mac
- Getting started with xUnit.net: Using .NET Core with the .NET SDK command line
- Moq
- Moq Quickstart
- JustMockLite: A mocking framework for .NET developers. (*Not maintained or supported by Microsoft.*)

Unit test controller logic in ASP.NET Core

9/22/2020 • 25 minutes to read • Edit Online

By Steve Smith

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Unit testing controllers

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app.

View or download sample code (how to download)

The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new StormSessionViewModel()
        {
            Id = session.Id,
            DateCreated = session.DateCreated,
            Name = session.Name,
            IdeaCount = session.Ideas.Count
        });

        return View(model);
    }

    public class NewSessionModel
    {
        [Required]
        public string SessionName { get; set; }
    }

    [HttpPost]
    public async Task<IActionResult> Index(NewSessionModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        else
        {
            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });
        }

        return RedirectToAction(actionName: nameof(Index));
    }
}
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.

- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
  - The `Add` method on the repository is called.
  - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

```
[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> **NOTE**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (*Controllers/SessionController.cs*):

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return Ok(result);
}

[HttpPost("create")]
public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);
    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return Ok(session);
}
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).
- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState` :

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null` , so the mock repository is configured to

return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

In ASP.NET Core 2.1 or later, ActionResult<T> (ActionResult<TValue>) enables you to return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

```
[Fact]
public async Task ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>> CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id }, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

```
[Fact]
public async Task CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnValue.Ideas.LastOrDefault().Description);
}
```

Controllers play a central role in any ASP.NET Core MVC app. As such, you should have confidence that controllers behave as intended. Automated tests can detect errors before the app is deployed to a production environment.

View or download sample code (how to download)

Unit tests of controller logic

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app. The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new StormSessionViewModel()
        {
            Id = session.Id,
            DateCreated = session.DateCreated,
            Name = session.Name,
            IdeaCount = session.Ideas.Count
        });

        return View(model);
    }

    public class NewSessionModel
    {
        [Required]
        public string SessionName { get; set; }
    }

    [HttpPost]
    public async Task<IActionResult> Index(NewSessionModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        else
        {
            await _sessionRepository.AddAsync(new BrainstormSession()
            {
                DateCreated = DateTimeOffset.Now,
                Name = model.SessionName
            });
        }

        return RedirectToAction(actionName: nameof(Index));
    }
}
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.

- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
  - The `Add` method on the repository is called.
  - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

```
[Fact]
public async Task IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> **NOTE**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (*Controllers/SessionController.cs*):

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return Ok(result);
}

[HttpPost("create")]
public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);
    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return Ok(session);
}
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).
- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState` :

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null` , so the mock repository is configured to

return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

In ASP.NET Core 2.1 or later, ActionResult<T> (ActionResult<TValue>) enables you to return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

```
[Fact]
public async Task ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>> CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id }, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

```
[Fact]
public async Task CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription, returnValue.Ideas.LastOrDefault().Description);
}
```

Additional resources

- Integration tests in ASP.NET Core
- Create and run unit tests with Visual Studio
- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)
- JustMockLite: A mocking framework for .NET developers. (*Not maintained or supported by Microsoft.*)

Test ASP.NET Core middleware

9/22/2020 • 3 minutes to read • Edit Online

By Chris Ross

Middleware can be tested in isolation with TestServer. It allows you to:

- Instantiate an app pipeline containing only the components that you need to test.
- Send custom requests to verify middleware behavior.

Advantages:

- Requests are sent in-memory rather than being serialized over the network.
- This avoids additional concerns, such as port management and HTTPS certificates.
- Exceptions in the middleware can flow directly back to the calling test.
- It's possible to customize server data structures, such as HttpContext, directly in the test.

Set up the TestServer

In the test project, create a test:

- Build and start a host that uses TestServer.
- Add any required services that the middleware uses.
- Add the Microsoft.AspNetCore.TestHost NuGet package to the project:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.TestHost" Version="3.1.*" />
</ItemGroup>
```

- Configure the processing pipeline to use the middleware for the test.

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    ...
}
```

Send requests with HttpClient

Send a request using HttpClient:

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    ...
}
```

Assert the result. First, make an assertion the opposite of the result that you expect. An initial run with a false positive assertion confirms that the test fails when the middleware is performing correctly. Run the test and confirm that the test fails.

In the following example, the middleware should return a 404 status code (*Not Found*) when the root endpoint is requested. Make the first test run with `Assert.NotEqual( ... );`, which should fail:

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.NotEqual(HttpStatusCode.NotFound, response.StatusCode);
}
```

Change the assertion to test the middleware under normal operating conditions. The final test uses `Assert.Equal( ... );`. Run the test again to confirm that it passes.

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Send requests with HttpContext

A test app can also send a request using SendAsync(Action<HttpContext>, CancellationToken). In the following example, several checks are made when `https://example.com/A/Path/?and=query` is processed by the middleware:

```
[Fact]
public async Task TestMiddleware_ExpectedResponse()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var server = host.GetTestServer();
    server.BaseAddress = new Uri("https://example.com/A/Path/");

    var context = await server.SendAsync(c =>
    {
        c.Request.Method = HttpMethods.Post;
        c.Request.Path = "/and/file.txt";
        c.Request.QueryString = new QueryString("?and=query");
    });

    Assert.True(context.RequestAborted.CanBeCanceled);
    Assert.Equal(HttpProtocol.Http11, context.Request.Protocol);
    Assert.Equal("POST", context.Request.Method);
    Assert.Equal("https", context.Request.Scheme);
    Assert.Equal("example.com", context.Request.Host.Value);
    Assert.Equal("/A/Path", context.Request.PathBase.Value);
    Assert.Equal("/and/file.txt", context.Request.Path.Value);
    Assert.Equal("?and=query", context.Request.QueryString.Value);
    Assert.NotNull(context.Request.Body);
    Assert.NotNull(context.Request.Headers);
    Assert.NotNull(context.Response.Headers);
    Assert.NotNull(context.Response.Body);
    Assert.Equal(404, context.Response.StatusCode);
    Assert.Null(context.Features.Get<IHttpResponseFeature>().ReasonPhrase);
}
```

SendAsync permits direct configuration of an HttpContext object rather than using the HttpClient abstractions. Use SendAsync to manipulate structures only available on the server, such as HttpContext.Items or HttpContext.Features.

As with the earlier example that tested for a *404 - Not Found* response, check the opposite for each `Assert` statement in the preceding test. The check confirms that the test fails correctly when the middleware is operating normally. After you've confirmed that the false positive test works, set the final `Assert` statements for the expected conditions and values of the test. Run it again to confirm that the test passes.

TestServer limitations

TestServer:

- Was created to replicate server behaviors to test middleware.
- Does ***not*** try to replicate all HttpClient behaviors.
- Attempts to give the client access to as much control over the server as possible, and with as much visibility into what's happening on the server as possible. For example it may throw exceptions not normally thrown by

`HttpClient` in order to directly communicate server state.

- Doesn't set some transport specific headers by default as those are not usually relevant to middleware. For more information, see the next section.

Content-Length and Transfer-Encoding headers

TestServer does ***not*** set transport related request or response headers such as Content-Length or Transfer-Encoding. Applications should avoid depending on these headers because their usage varies by client, scenario, and protocol. If `Content-Length` and `Transfer-Encoding` are necessary to test a specific scenario, they can be specified in the test when composing the HttpRequestMessage or HttpContext. For more information, see the following GitHub issues:

- dotnet/aspnetcore#21677
- dotnet/aspnetcore#18463
- dotnet/aspnetcore#13273

Integration tests in ASP.NET Core

9/22/2020 • 38 minutes to read • Edit Online

By Javier Calvarro Nelson, Steve Smith, and Jos van der Til

Integration tests ensure that an app's components function correctly at a level that includes the app's supporting infrastructure, such as the database, file system, and network. ASP.NET Core supports integration tests using a unit test framework with a test web host and an in-memory test server.

This topic assumes a basic understanding of unit tests. If unfamiliar with test concepts, see the Unit Testing in .NET Core and .NET Standard topic and its linked content.

View or download sample code (how to download)

The sample app is a Razor Pages app and assumes a basic understanding of Razor Pages. If unfamiliar with Razor Pages, see the following topics:

- Introduction to Razor Pages
- Get started with Razor Pages
- Razor Pages unit tests

> **NOTE**
> For testing SPAs, we recommended a tool such as Selenium, which can automate a browser.

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test, choose the unit test.

TIP

Don't write integration tests for every possible permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes/mocks result in faster test execution.

NOTE

In discussions of integration tests, the tested project is frequently called the *System Under Test*, or "SUT" for short.

"SUT" is used throughout this topic to refer to the tested ASP.NET Core app.

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (*.deps*) from the SUT into the test project's *bin* directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

> **NOTE**
>
> When creating a test project for an app, separate the unit tests from the integration tests into different projects. This helps ensure that infrastructure testing components aren't accidentally included in the unit tests. Separation of unit and integration tests also allows control over which set of tests are run.

There's virtually no difference between the configuration for tests of Razor Pages apps and MVC apps. The only difference is in how the tests are named. In a Razor Pages app, tests of page endpoints are usually named after the page model class (for example, `IndexPageTests` to test component integration for the Index page). In an MVC app, tests are usually organized by controller classes and named after the controllers they test (for example, `HomeControllerTests` to test component integration for the Home controller).

Test app prerequisites

The test project must:

- Reference the Microsoft.AspNetCore.Mvc.Testing package.
- Specify the Web SDK in the project file (`<Project Sdk="Microsoft.NET.Sdk.Web">`).

These prerequisites can be seen in the sample app. Inspect the *tests/RazorPagesProject.Tests/RazorPagesProject.Tests.csproj* file. The sample app uses the xUnit test framework and the AngleSharp parser library, so the sample app also references:

- xunit
- xunit.runner.visualstudio
- AngleSharp

Entity Framework Core is also used in the tests. The app references:

- Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
- Microsoft.AspNetCore.Identity.EntityFrameworkCore
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.InMemory
- Microsoft.EntityFrameworkCore.Tools

SUT environment

If the SUT's environment isn't set, the environment defaults to Development.

Basic tests with the default WebApplicationFactory

WebApplicationFactory<TEntryPoint> is used to create a TestServer for the integration tests. `TEntryPoint` is the entry point class of the SUT, usually the `Startup` class.

Test classes implement a *class fixture* interface (IClassFixture) to indicate the class contains tests and provide shared object instances across the tests in the class.

The following test class, `BasicTests`, uses the `WebApplicationFactory` to bootstrap the SUT and provide an HttpClient to a test method, `Get_EndpointsReturnSuccessAndCorrectContentType`. The method checks if the response status code is successful (status codes in the range 200-299) and the `Content-Type` header is `text/html; charset=utf-8` for several app pages.

CreateClient creates an instance of `HttpClient` that automatically follows redirects and handles cookies.

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly WebApplicationFactory<RazorPagesProject.Startup> _factory;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    [InlineData("/Privacy")]
    [InlineData("/Contact")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string url)
    {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

By default, non-essential cookies aren't preserved across requests when the GDPR consent policy is enabled. To preserve non-essential cookies, such as those used by the TempData provider, mark them as essential in your tests. For instructions on marking a cookie as essential, see Essential cookies.

Customize WebApplicationFactory

Web host configuration can be created independently of the test classes by inheriting from `WebApplicationFactory` to create one or more custom factories:

1. Inherit from `WebApplicationFactory` and override ConfigureWebHost. The IWebHostBuilder allows the configuration of the service collection with ConfigureServices:

```
public class CustomWebApplicationFactory<TStartup>
    : WebApplicationFactory<TStartup> where TStartup: class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            var descriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));

            services.Remove(descriptor);

            services.AddDbContext<ApplicationDbContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryDbForTesting");
            });

            var sp = services.BuildServiceProvider();

            using (var scope = sp.CreateScope())
            {
                var scopedServices = scope.ServiceProvider;
                var db = scopedServices.GetRequiredService<ApplicationDbContext>();
                var logger = scopedServices
                    .GetRequiredService<ILogger<CustomWebApplicationFactory<TStartup>>>();

                db.Database.EnsureCreated();

                try
                {
                    Utilities.InitializeDbForTests(db);
                }
                catch (Exception ex)
                {
                    logger.LogError(ex, "An error occurred seeding the " +
                        "database with test messages. Error: {Message}", ex.Message);
                }
            }
        });
    }
}
```

Database seeding in the sample app is performed by the `InitializeDbForTests` method. The method is described in the Integration tests sample: Test app organization section.

The SUT's database context is registered in its `Startup.ConfigureServices` method. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Startup.ConfigureServices` code is executed. The execution order is a breaking change for the Generic Host with the release of ASP.NET Core 3.0. To use a different database for the tests than the app's database, the app's database context must be replaced in `builder.ConfigureServices`.

For SUTs that still use the Web Host, the test app's `builder.ConfigureServices` callback is executed *before* the SUT's `Startup.ConfigureServices` code. The test app's `builder.ConfigureTestServices` callback is executed *after*.

The sample app finds the service descriptor for the database context and uses the descriptor to remove the service registration. Next, the factory adds a new `ApplicationDbContext` that uses an in-memory database for the tests.

To connect to a different database than the in-memory database, change the `UseInMemoryDatabase` call to connect the context to a different database. To use a SQL Server test database:

- Reference the Microsoft.EntityFrameworkCore.SqlServer NuGet package in the project file.
- Call `UseSqlServer` with a connection string to the database.

```
services.AddDbContext<ApplicationDbContext>((options, context) =>
{
    context.UseSqlServer(
        Configuration.GetConnectionString("TestingDbConnectionString"));
});
```

2. Use the custom `CustomWebApplicationFactory` in test classes. The following example uses the factory in the `IndexPageTests` class:

```
public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<RazorPagesProject.Startup>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new WebApplicationFactoryClientOptions
            {
                AllowAutoRedirect = false
            });
    }
```

   The sample app's client is configured to prevent the `HttpClient` from following redirects. As explained later in the Mock authentication section, this permits tests to check the result of the app's first response. The first response is a redirect in many of these tests with a `Location` header.

3. A typical test uses the `HttpClient` and helper methods to process the request and the response:

```
[Fact]
public async Task Post_DeleteAllMessagesHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("button[id='deleteAllBtn']"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

Any POST request to the SUT must satisfy the antiforgery check that's automatically made by the app's data protection antiforgery system. In order to arrange for a test's POST request, the test app must:

1. Make a request for the page.
2. Parse the antiforgery cookie and request validation token from the response.
3. Make the POST request with the antiforgery cookie and request validation token in place.

The `SendAsync` helper extension methods (*Helpers/HttpClientExtensions.cs*) and the `GetDocumentAsync` helper method (*Helpers/HtmlHelpers.cs*) in the sample app use the AngleSharp parser to handle the antiforgery check with the following methods:

- `GetDocumentAsync`: Receives the HttpResponseMessage and returns an `IHtmlDocument`. `GetDocumentAsync` uses a factory that prepares a *virtual response* based on the original `HttpResponseMessage`. For more information, see the AngleSharp documentation.
- `SendAsync` extension methods for the `HttpClient` compose an HttpRequestMessage and call SendAsync(HttpRequestMessage) to submit requests to the SUT. Overloads for `SendAsync` accept the HTML form (`IHtmlFormElement`) and the following:
  - Submit button of the form (`IHtmlElement`)
  - Form values collection (`IEnumerable<KeyValuePair<string, string>>`)
  - Submit button (`IHtmlElement`) and form values (`IEnumerable<KeyValuePair<string, string>>`)

> **NOTE**
>
> AngleSharp is a third-party parsing library used for demonstration purposes in this topic and the sample app. AngleSharp isn't supported or required for integration testing of ASP.NET Core apps. Other parsers can be used, such as the Html Agility Pack (HAP). Another approach is to write code to handle the antiforgery system's request verification token and antiforgery cookie directly.

Customize the client with WithWebHostBuilder

When additional configuration is required within a test method, WithWebHostBuilder creates a new `WebApplicationFactory` with an IWebHostBuilder that is further customized by configuration.

The `Post_DeleteMessageHandler_ReturnsRedirectToRoot` test method of the sample app demonstrates the use of `WithWebHostBuilder`. This test performs a record delete in the database by triggering a form submission in the SUT.

Because another test in the `IndexPageTests` class performs an operation that deletes all of the records in the database and may run before the `Post_DeleteMessageHandler_ReturnsRedirectToRoot` method, the database is reseeded in this test method to ensure that a record is present for the SUT to delete. Selecting the first delete button of the `messages` form in the SUT is simulated in the request to the SUT:

```
[Fact]
public async Task Post_DeleteMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                var serviceProvider = services.BuildServiceProvider();

                using (var scope = serviceProvider.CreateScope())
                {
                    var scopedServices = scope.ServiceProvider;
                    var db = scopedServices
                        .GetRequiredService<ApplicationDbContext>();
                    var logger = scopedServices
                        .GetRequiredService<ILogger<IndexPageTests>>();

                    try
                    {
                        Utilities.ReinitializeDbForTests(db);
                    }
                    catch (Exception ex)
                    {
                        logger.LogError(ex, "An error occurred seeding " +
                            "the database with test messages. Error: {Message}",
                            ex.Message);
                    }
                }
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("form[id='messages']")
            .QuerySelector("div[class='panel-body']")
            .QuerySelector("button"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

Client options

The following table shows the default WebApplicationFactoryClientOptions available when creating `HttpClient` instances.

OPTION	DESCRIPTION	DEFAULT
AllowAutoRedirect	Gets or sets whether or not `HttpClient` instances should automatically follow redirect responses.	`true`

OPTION	DESCRIPTION	DEFAULT
BaseAddress	Gets or sets the base address of `HttpClient` instances.	`http://localhost`
HandleCookies	Gets or sets whether `HttpClient` instances should handle cookies.	`true`
MaxAutomaticRedirections	Gets or sets the maximum number of redirect responses that `HttpClient` instances should follow.	7

Create the `WebApplicationFactoryClientOptions` class and pass it to the CreateClient method (default values are shown in the code example):

```
// Default client option values are shown
var clientOptions = new WebApplicationFactoryClientOptions();
clientOptions.AllowAutoRedirect = true;
clientOptions.BaseAddress = new Uri("http://localhost");
clientOptions.HandleCookies = true;
clientOptions.MaxAutomaticRedirections = 7;

_client = _factory.CreateClient(clientOptions);
```

Inject mock services

Services can be overridden in a test with a call to ConfigureTestServices on the host builder. **To inject mock services, the SUT must have a `Startup` class with a `Startup.ConfigureServices` method.**

The sample SUT includes a scoped service that returns a quote. The quote is embedded in a hidden field on the Index page when the Index page is requested.

Services/IQuoteService.cs:

```
public interface IQuoteService
{
    Task<string> GenerateQuote();
}
```

Services/QuoteService.cs:

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Dr. Who: Planet of Evil
// https://www.bbc.co.uk/programmes/p00pyrx6
public class QuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Come on, Sarah. We've an appointment in London, " +
            "and we're already 30,000 years late.");
    }
}
```

Startup.cs:

```
services.AddScoped<IQuoteService, QuoteService>();
```

Pages/Index.cshtml.cs:

```
public class IndexModel : PageModel
{
    private readonly ApplicationDbContext _db;
    private readonly IQuoteService _quoteService;

    public IndexModel(ApplicationDbContext db, IQuoteService quoteService)
    {
        _db = db;
        _quoteService = quoteService;
    }

    [BindProperty]
    public Message Message { get; set; }

    public IList<Message> Messages { get; private set; }

    [TempData]
    public string MessageAnalysisResult { get; set; }

    public string Quote { get; private set; }

    public async Task OnGetAsync()
    {
        Messages = await _db.GetMessagesAsync();

        Quote = await _quoteService.GenerateQuote();
    }
```

Pages/Index.cs:

```
<input id="quote" type="hidden" value="@Model.Quote">
```

The following markup is generated when the SUT app is run:

```
<input id="quote" type="hidden" value="Come on, Sarah. We&#x27;ve an appointment in
    London, and we&#x27;re already 30,000 years late.">
```

To test the service and quote injection in an integration test, a mock service is injected into the SUT by the test. The mock service replaces the app's `QuoteService` with a service provided by the test app, called `TestQuoteService`:

IntegrationTests.IndexPageTests.cs:

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Pyramids of Mars
// https://www.bbc.co.uk/programmes/p00pys55
public class TestQuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Something's interfering with time, Mr. Scarman, " +
            "and time is my business.");
    }
}
```

`ConfigureTestServices` is called, and the scoped service is registered:

```
[Fact]
public async Task Get_QuoteService_ProvidesQuoteInPage()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddScoped<IQuoteService, TestQuoteService>();
            });
        })
        .CreateClient();

    //Act
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);
    var quoteElement = content.QuerySelector("#quote");

    // Assert
    Assert.Equal("Something's interfering with time, Mr. Scarman, " +
        "and time is my business.", quoteElement.Attributes["value"].Value);
}
```

The markup produced during the test's execution reflects the quote text supplied by `TestQuoteService`, thus the assertion passes:

```
<input id="quote" type="hidden" value="Something&#x27;s interfering with time,
    Mr. Scarman, and time is my business.">
```

Mock authentication

Tests in the `AuthTests` class check that a secure endpoint:

- Redirects an unauthenticated user to the app's Login page.
- Returns content for an authenticated user.

In the SUT, the `/SecurePage` page uses an AuthorizePage convention to apply an AuthorizeFilter to the page. For more information, see Razor Pages authorization conventions.

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/SecurePage");
});
```

In the `Get_SecurePageRedirectsAnUnauthenticatedUser` test, a WebApplicationFactoryClientOptions is set to disallow redirects by setting AllowAutoRedirect to `false`:

```
[Fact]
public async Task Get_SecurePageRedirectsAnUnauthenticatedUser()
{
    // Arrange
    var client = _factory.CreateClient(
        new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });

    // Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.StartsWith("http://localhost/Identity/Account/Login",
        response.Headers.Location.OriginalString);
}
```

By disallowing the client to follow the redirect, the following checks can be made:

- The status code returned by the SUT can be checked against the expected HttpStatusCode.Redirect result, not the final status code after the redirect to the Login page, which would be HttpStatusCode.OK.
- The `Location` header value in the response headers is checked to confirm that it starts with `http://localhost/Identity/Account/Login`, not the final Login page response, where the `Location` header wouldn't be present.

The test app can mock an AuthenticationHandler<TOptions> in ConfigureTestServices in order to test aspects of authentication and authorization. A minimal scenario returns an AuthenticateResult.Success:

```
public class TestAuthHandler : AuthenticationHandler<AuthenticationSchemeOptions>
{
    public TestAuthHandler(IOptionsMonitor<AuthenticationSchemeOptions> options,
        ILoggerFactory logger, UrlEncoder encoder, ISystemClock clock)
        : base(options, logger, encoder, clock)
    {
    }

    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        var claims = new[] { new Claim(ClaimTypes.Name, "Test user") };
        var identity = new ClaimsIdentity(claims, "Test");
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, "Test");

        var result = AuthenticateResult.Success(ticket);

        return Task.FromResult(result);
    }
}
```

The `TestAuthHandler` is called to authenticate a user when the authentication scheme is set to `Test` where `AddAuthentication` is registered for `ConfigureTestServices`. It's important for the `Test` scheme to match the scheme your app expects. Otherwise, authentication won't work.

```
[Fact]
public async Task Get_SecurePageIsReturnedForAnAuthenticatedUser()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddAuthentication("Test")
                    .AddScheme<AuthenticationSchemeOptions, TestAuthHandler>(
                        "Test", options => {});
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false,
        });

    client.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Test");

    //Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

For more information on `WebApplicationFactoryClientOptions`, see the Client options section.

Set the environment

By default, the SUT's host and app environment is configured to use the Development environment. To override the SUT's environment when using `IHostBuilder`:

- Set the `ASPNETCORE_ENVIRONMENT` environment variable (for example, `Staging`, `Production`, or other custom value, such as `Testing`).
- Override `CreateHostBuilder` in the test app to read environment variables prefixed with `ASPNETCORE`.

```
protected override IHostBuilder CreateHostBuilder() =>
    base.CreateHostBuilder()
        .ConfigureHostConfiguration(
            config => config.AddEnvironmentVariables("ASPNETCORE"));
```

If the SUT uses the Web Host (`IWebHostBuilder`), override `CreateWebHostBuilder`:

```
protected override IWebHostBuilder CreateWebHostBuilder() =>
    base.CreateWebHostBuilder().UseEnvironment("Testing");
```

How the test infrastructure infers the app content root path

The `WebApplicationFactory` constructor infers the app content root path by searching for a WebApplicationFactoryContentRootAttribute on the assembly containing the integration tests with a key equal to the `TEntryPoint` assembly `System.Reflection.Assembly.FullName`. In case an attribute with the correct key isn't found, `WebApplicationFactory` falls back to searching for a solution file (*.sln*) and appends the `TEntryPoint` assembly name to the solution directory. The app root directory (the content root path) is used to discover views and content files.

Disable shadow copying

Shadow copying causes the tests to execute in a different directory than the output directory. For tests to work properly, shadow copying must be disabled. The sample app uses xUnit and disables shadow copying for xUnit by including an *xunit.runner.json* file with the correct configuration setting. For more information, see Configuring xUnit with JSON.

Add the *xunit.runner.json* file to root of the test project with the following content:

```
{
  "shadowCopy": false
}
```

Disposal of objects

After the tests of the `IClassFixture` implementation are executed, TestServer and HttpClient are disposed when xUnit disposes of the WebApplicationFactory. If objects instantiated by the developer require disposal, dispose of them in the `IClassFixture` implementation. For more information, see Implementing a Dispose method.

Integration tests sample

The sample app is composed of two apps:

APP	PROJECT DIRECTORY	DESCRIPTION
Message app (the SUT)	*src/RazorPagesProject*	Allows a user to add, delete one, delete all, and analyze messages.
Test app	*tests/RazorPagesProject.Tests*	Used to integration test the SUT.

The tests can be run using the built-in test features of an IDE, such as Visual Studio. If using Visual Studio Code or the command line, execute the following command at a command prompt in the *tests/RazorPagesProject.Tests* directory:

```
dotnet test
```

Message app (SUT) organization

The SUT is a Razor Pages message system with the following characteristics:

- The Index page of the app (*Pages/Index.cshtml* and *Pages/Index.cshtml.cs*) provides a UI and page model methods to control the addition, deletion, and analysis of messages (average words per message).
- A message is described by the `Message` class (*Data/Message.cs*) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.
- The app contains a data access layer (DAL) in its database context class, `AppDbContext` (*Data/AppDbContext.cs*).
- If the database is empty on app startup, the message store is initialized with three messages.
- The app includes a `/SecurePage` that can only be accessed by an authenticated user.

†The EF topic, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the *tests/RazorPagesProject.Tests* directory.

TEST APP DIRECTORY	DESCRIPTION
AuthTests	Contains test methods for: • Accessing a secure page by an unauthenticated user. • Accessing a secure page by an authenticated user with a mock AuthenticationHandler<TOptions>. • Obtaining a GitHub user profile and checking the profile's user login.
BasicTests	Contains a test method for routing and content type.
IntegrationTests	Contains the integration tests for the Index page using custom `WebApplicationFactory` class.
Helpers/Utilities	• *Utilities.cs* contains the `InitializeDbForTests` method used to seed the database with test data. • *HtmlHelpers.cs* provides a method to return an AngleSharp `IHtmlDocument` for use by the test methods. • *HttpClientExtensions.cs* provide overloads for `SendAsync` to submit requests to the SUT.

The test framework is xUnit. Integration tests are conducted using the Microsoft.AspNetCore.TestHost, which includes the TestServer. Because the Microsoft.AspNetCore.Mvc.Testing package is used to configure the test host and test server, the `TestHost` and `TestServer` packages don't require direct package references in the test app's project file or developer configuration in the test app.

Seeding the database for testing

Integration tests usually require a small dataset in the database prior to the test execution. For example, a delete test calls for a database record deletion, so the database must have at least one record for the delete request to succeed.

The sample app seeds the database with three messages in *Utilities.cs* that tests can use when they execute:

```
public static void InitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.AddRange(GetSeedingMessages());
    db.SaveChanges();
}

public static void ReinitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.RemoveRange(db.Messages);
    InitializeDbForTests(db);
}

public static List<Message> GetSeedingMessages()
{
    return new List<Message>()
    {
        new Message(){ Text = "TEST RECORD: You're standing on my scarf." },
        new Message(){ Text = "TEST RECORD: Would you like a jelly baby?" },
        new Message(){ Text = "TEST RECORD: To the rational mind, " +
            "nothing is inexplicable; only unexplained." }
    };
}
```

The SUT's database context is registered in its `Startup.ConfigureServices` method. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Startup.ConfigureServices` code is executed. To use a different database for the tests, the app's database context must be replaced in `builder.ConfigureServices`. For more information, see the Customize WebApplicationFactory section.

For SUTs that still use the Web Host, the test app's `builder.ConfigureServices` callback is executed *before* the SUT's `Startup.ConfigureServices` code. The test app's `builder.ConfigureTestServices` callback is executed *after*.

Integration tests ensure that an app's components function correctly at a level that includes the app's supporting infrastructure, such as the database, file system, and network. ASP.NET Core supports integration tests using a unit test framework with a test web host and an in-memory test server.

This topic assumes a basic understanding of unit tests. If unfamiliar with test concepts, see the Unit Testing in .NET Core and .NET Standard topic and its linked content.

View or download sample code (how to download)

The sample app is a Razor Pages app and assumes a basic understanding of Razor Pages. If unfamiliar with Razor Pages, see the following topics:

- Introduction to Razor Pages
- Get started with Razor Pages
- Razor Pages unit tests

> **NOTE**
>
> For testing SPAs, we recommended a tool such as Selenium, which can automate a browser.

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test, choose the unit test.

> **TIP**
> Don't write integration tests for every possible permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes/mocks result in faster test execution.

> **NOTE**
> In discussions of integration tests, the tested project is frequently called the *System Under Test*, or "SUT" for short.
>
> *"SUT" is used throughout this topic to refer to the tested ASP.NET Core app.*

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For

example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (*.deps*) from the SUT into the test project's *bin* directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

> **NOTE**
> When creating a test project for an app, separate the unit tests from the integration tests into different projects. This helps ensure that infrastructure testing components aren't accidentally included in the unit tests. Separation of unit and integration tests also allows control over which set of tests are run.

There's virtually no difference between the configuration for tests of Razor Pages apps and MVC apps. The only difference is in how the tests are named. In a Razor Pages app, tests of page endpoints are usually named after the page model class (for example, `IndexPageTests` to test component integration for the Index page). In an MVC app, tests are usually organized by controller classes and named after the controllers they test (for example, `HomeControllerTests` to test component integration for the Home controller).

Test app prerequisites

The test project must:

- Reference the following packages:
  - Microsoft.AspNetCore.App
  - Microsoft.AspNetCore.Mvc.Testing
- Specify the Web SDK in the project file (`<Project Sdk="Microsoft.NET.Sdk.Web">`). The Web SDK is required when referencing the Microsoft.AspNetCore.App metapackage.

These prerequisites can be seen in the sample app. Inspect the *tests/RazorPagesProject.Tests/RazorPagesProject.Tests.csproj* file. The sample app uses the xUnit test framework and the AngleSharp parser library, so the sample app also references:

- xunit
- xunit.runner.visualstudio
- AngleSharp

SUT environment

If the SUT's environment isn't set, the environment defaults to Development.

Basic tests with the default WebApplicationFactory

WebApplicationFactory<TEntryPoint> is used to create a TestServer for the integration tests. `TEntryPoint` is the entry point class of the SUT, usually the `Startup` class.

Test classes implement a *class fixture* interface (IClassFixture) to indicate the class contains tests and provide shared object instances across the tests in the class.

The following test class, `BasicTests`, uses the `WebApplicationFactory` to bootstrap the SUT and provide an HttpClient to a test method, `Get_EndpointsReturnSuccessAndCorrectContentType`. The method checks if the response status code is successful (status codes in the range 200-299) and the `Content-Type` header is `text/html; charset=utf-8` for several app pages.

CreateClient creates an instance of `HttpClient` that automatically follows redirects and handles cookies.

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly WebApplicationFactory<RazorPagesProject.Startup> _factory;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    [InlineData("/Privacy")]
    [InlineData("/Contact")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string url)
    {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

By default, non-essential cookies aren't preserved across requests when the GDPR consent policy is enabled. To preserve non-essential cookies, such as those used by the TempData provider, mark them as essential in your tests. For instructions on marking a cookie as essential, see Essential cookies.

Customize WebApplicationFactory

Web host configuration can be created independently of the test classes by inheriting from `WebApplicationFactory` to create one or more custom factories:

1. Inherit from `WebApplicationFactory` and override ConfigureWebHost. The IWebHostBuilder allows the configuration of the service collection with ConfigureServices, which is executed before the app's `Startup.ConfigureServices`. The IWebHostBuilder allows overriding and modifying registered services in the service collection with ConfigureTestServices:

```
public class CustomWebApplicationFactory<TStartup>
    : WebApplicationFactory<TStartup> where TStartup: class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            // Create a new service provider.
            var serviceProvider = new ServiceCollection()
                .AddEntityFrameworkInMemoryDatabase()
                .BuildServiceProvider();

            // Add a database context (ApplicationDbContext) using an in-memory
            // database for testing.
            services.AddDbContext<ApplicationDbContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryDbForTesting");
                options.UseInternalServiceProvider(serviceProvider);
            });

            // Build the service provider.
            var sp = services.BuildServiceProvider();

            // Create a scope to obtain a reference to the database
            // context (ApplicationDbContext).
            using (var scope = sp.CreateScope())
            {
                var scopedServices = scope.ServiceProvider;
                var db = scopedServices.GetRequiredService<ApplicationDbContext>();
                var logger = scopedServices
                    .GetRequiredService<ILogger<CustomWebApplicationFactory<TStartup>>>();

                // Ensure the database is created.
                db.Database.EnsureCreated();

                try
                {
                    // Seed the database with test data.
                    Utilities.InitializeDbForTests(db);
                }
                catch (Exception ex)
                {
                    logger.LogError(ex, "An error occurred seeding the database. Error: {Message}",
ex.Message);
                }
            }
        });
    }
}
```

Database seeding in the sample app is performed by the `InitializeDbForTests` method. The method is described in the Integration tests sample: Test app organization section.

2. Use the custom `CustomWebApplicationFactory` in test classes. The following example uses the factory in the `IndexPageTests` class:

```
public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<RazorPagesProject.Startup>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new WebApplicationFactoryClientOptions
            {
                AllowAutoRedirect = false
            });
    }
```

The sample app's client is configured to prevent the `HttpClient` from following redirects. As explained later in the Mock authentication section, this permits tests to check the result of the app's first response. The first response is a redirect in many of these tests with a `Location` header.

3. A typical test uses the `HttpClient` and helper methods to process the request and the response:

```
[Fact]
public async Task Post_DeleteAllMessagesHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("button[id='deleteAllBtn']"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

Any POST request to the SUT must satisfy the antiforgery check that's automatically made by the app's data protection antiforgery system. In order to arrange for a test's POST request, the test app must:

1. Make a request for the page.
2. Parse the antiforgery cookie and request validation token from the response.
3. Make the POST request with the antiforgery cookie and request validation token in place.

The `SendAsync` helper extension methods (*Helpers/HttpClientExtensions.cs*) and the `GetDocumentAsync` helper method (*Helpers/HtmlHelpers.cs*) in the sample app use the AngleSharp parser to handle the antiforgery check with the following methods:

- `GetDocumentAsync`: Receives the HttpResponseMessage and returns an `IHtmlDocument`. `GetDocumentAsync` uses a factory that prepares a *virtual response* based on the original `HttpResponseMessage`. For more information, see the AngleSharp documentation.
- `SendAsync` extension methods for the `HttpClient` compose an HttpRequestMessage and call SendAsync(HttpRequestMessage) to submit requests to the SUT. Overloads for `SendAsync` accept the HTML form (`IHtmlFormElement`) and the following:
  - Submit button of the form (`IHtmlElement`)

- Form values collection (`IEnumerable<KeyValuePair<string, string>>`)
- Submit button (`IHtmlElement`) and form values (`IEnumerable<KeyValuePair<string, string>>`)

> **NOTE**
>
> AngleSharp is a third-party parsing library used for demonstration purposes in this topic and the sample app. AngleSharp isn't supported or required for integration testing of ASP.NET Core apps. Other parsers can be used, such as the Html Agility Pack (HAP). Another approach is to write code to handle the antiforgery system's request verification token and antiforgery cookie directly.

Customize the client with WithWebHostBuilder

When additional configuration is required within a test method, WithWebHostBuilder creates a new `WebApplicationFactory` with an IWebHostBuilder that is further customized by configuration.

The `Post_DeleteMessageHandler_ReturnsRedirectToRoot` test method of the sample app demonstrates the use of `WithWebHostBuilder`. This test performs a record delete in the database by triggering a form submission in the SUT.

Because another test in the `IndexPageTests` class performs an operation that deletes all of the records in the database and may run before the `Post_DeleteMessageHandler_ReturnsRedirectToRoot` method, the database is reseeded in this test method to ensure that a record is present for the SUT to delete. Selecting the first delete button of the `messages` form in the SUT is simulated in the request to the SUT:

```
[Fact]
public async Task Post_DeleteMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                var serviceProvider = services.BuildServiceProvider();

                using (var scope = serviceProvider.CreateScope())
                {
                    var scopedServices = scope.ServiceProvider;
                    var db = scopedServices
                        .GetRequiredService<ApplicationDbContext>();
                    var logger = scopedServices
                        .GetRequiredService<ILogger<IndexPageTests>>();

                    try
                    {
                        Utilities.ReinitializeDbForTests(db);
                    }
                    catch (Exception ex)
                    {
                        logger.LogError(ex, "An error occurred seeding " +
                            "the database with test messages. Error: {Message}" +
                            ex.Message);
                    }
                }
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("form[id='messages']")
            .QuerySelector("div[class='panel-body']")
            .QuerySelector("button"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

Client options

The following table shows the default WebApplicationFactoryClientOptions available when creating `HttpClient` instances.

OPTION	DESCRIPTION	DEFAULT
AllowAutoRedirect	Gets or sets whether or not `HttpClient` instances should automatically follow redirect responses.	`true`

OPTION	DESCRIPTION	DEFAULT
BaseAddress	Gets or sets the base address of `HttpClient` instances.	`http://localhost`
HandleCookies	Gets or sets whether `HttpClient` instances should handle cookies.	`true`
MaxAutomaticRedirections	Gets or sets the maximum number of redirect responses that `HttpClient` instances should follow.	7

Create the `WebApplicationFactoryClientOptions` class and pass it to the CreateClient method (default values are shown in the code example):

```
// Default client option values are shown
var clientOptions = new WebApplicationFactoryClientOptions();
clientOptions.AllowAutoRedirect = true;
clientOptions.BaseAddress = new Uri("http://localhost");
clientOptions.HandleCookies = true;
clientOptions.MaxAutomaticRedirections = 7;

_client = _factory.CreateClient(clientOptions);
```

Inject mock services

Services can be overridden in a test with a call to ConfigureTestServices on the host builder. **To inject mock services, the SUT must have a `Startup` class with a `Startup.ConfigureServices` method.**

The sample SUT includes a scoped service that returns a quote. The quote is embedded in a hidden field on the Index page when the Index page is requested.

Services/IQuoteService.cs:

```
public interface IQuoteService
{
    Task<string> GenerateQuote();
}
```

Services/QuoteService.cs:

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Dr. Who: Planet of Evil
// https://www.bbc.co.uk/programmes/p00pyrx6
public class QuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Come on, Sarah. We've an appointment in London, " +
            "and we're already 30,000 years late.");
    }
}
```

Startup.cs:

```
services.AddScoped<IQuoteService, QuoteService>();
```

Pages/Index.cshtml.cs:

```
public class IndexModel : PageModel
{
    private readonly ApplicationDbContext _db;
    private readonly IQuoteService _quoteService;

    public IndexModel(ApplicationDbContext db, IQuoteService quoteService)
    {
        _db = db;
        _quoteService = quoteService;
    }

    [BindProperty]
    public Message Message { get; set; }

    public IList<Message> Messages { get; private set; }

    [TempData]
    public string MessageAnalysisResult { get; set; }

    public string Quote { get; private set; }

    public async Task OnGetAsync()
    {
        Messages = await _db.GetMessagesAsync();

        Quote = await _quoteService.GenerateQuote();
    }
```

Pages/Index.cs:

```
<input id="quote" type="hidden" value="@Model.Quote">
```

The following markup is generated when the SUT app is run:

```
<input id="quote" type="hidden" value="Come on, Sarah. We&#x27;ve an appointment in
    London, and we&#x27;re already 30,000 years late.">
```

To test the service and quote injection in an integration test, a mock service is injected into the SUT by the test. The mock service replaces the app's `QuoteService` with a service provided by the test app, called `TestQuoteService`:

IntegrationTests.IndexPageTests.cs:

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Pyramids of Mars
// https://www.bbc.co.uk/programmes/p00pys55
public class TestQuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Something's interfering with time, Mr. Scarman, " +
            "and time is my business.");
    }
}
```

`ConfigureTestServices` is called, and the scoped service is registered:

```
[Fact]
public async Task Get_QuoteService_ProvidesQuoteInPage()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddScoped<IQuoteService, TestQuoteService>();
            });
        })
        .CreateClient();

    //Act
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);
    var quoteElement = content.QuerySelector("#quote");

    // Assert
    Assert.Equal("Something's interfering with time, Mr. Scarman, " +
        "and time is my business.", quoteElement.Attributes["value"].Value);
}
```

The markup produced during the test's execution reflects the quote text supplied by `TestQuoteService`, thus the assertion passes:

```
<input id="quote" type="hidden" value="Something&#x27;s interfering with time,
    Mr. Scarman, and time is my business.">
```

Mock authentication

Tests in the `AuthTests` class check that a secure endpoint:

- Redirects an unauthenticated user to the app's Login page.
- Returns content for an authenticated user.

In the SUT, the `/SecurePage` page uses an AuthorizePage convention to apply an AuthorizeFilter to the page. For more information, see Razor Pages authorization conventions.

```
services.AddMvc()
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/SecurePage");
    });
```

In the `Get_SecurePageRedirectsAnUnauthenticatedUser` test, a WebApplicationFactoryClientOptions is set to disallow redirects by setting AllowAutoRedirect to `false`:

```
[Fact]
public async Task Get_SecurePageRedirectsAnUnauthenticatedUser()
{
    // Arrange
    var client = _factory.CreateClient(
        new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });

    // Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.StartsWith("http://localhost/Identity/Account/Login",
        response.Headers.Location.OriginalString);
}
```

By disallowing the client to follow the redirect, the following checks can be made:

- The status code returned by the SUT can be checked against the expected HttpStatusCode.Redirect result, not the final status code after the redirect to the Login page, which would be HttpStatusCode.OK.
- The `Location` header value in the response headers is checked to confirm that it starts with `http://localhost/Identity/Account/Login`, not the final Login page response, where the `Location` header wouldn't be present.

The test app can mock an AuthenticationHandler<TOptions> in ConfigureTestServices in order to test aspects of authentication and authorization. A minimal scenario returns an AuthenticateResult.Success:

```
public class TestAuthHandler : AuthenticationHandler<AuthenticationSchemeOptions>
{
    public TestAuthHandler(IOptionsMonitor<AuthenticationSchemeOptions> options,
        ILoggerFactory logger, UrlEncoder encoder, ISystemClock clock)
        : base(options, logger, encoder, clock)
    {
    }

    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        var claims = new[] { new Claim(ClaimTypes.Name, "Test user") };
        var identity = new ClaimsIdentity(claims, "Test");
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, "Test");

        var result = AuthenticateResult.Success(ticket);

        return Task.FromResult(result);
    }
}
```

The `TestAuthHandler` is called to authenticate a user when the authentication scheme is set to `Test` where `AddAuthentication` is registered for `ConfigureTestServices`:

```
[Fact]
public async Task Get_SecurePageIsReturnedForAnAuthenticatedUser()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddAuthentication("Test")
                    .AddScheme<AuthenticationSchemeOptions, TestAuthHandler>(
                        "Test", options => {});
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false,
        });

    client.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Test");

    //Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

For more information on `WebApplicationFactoryClientOptions`, see the Client options section.

Set the environment

By default, the SUT's host and app environment is configured to use the Development environment. To override the SUT's environment:

- Set the `ASPNETCORE_ENVIRONMENT` environment variable (for example, `Staging`, `Production`, or other custom value, such as `Testing`).
- Override `CreateWebHostBuilder` in the test app to read the `ASPNETCORE_ENVIRONMENT` environment variable.

```
public class CustomWebApplicationFactory<TStartup>
    : WebApplicationFactory<TStartup> where TStartup: class
{
    protected override IWebHostBuilder CreateWebHostBuilder()
    {
        return base.CreateWebHostBuilder()
            .UseEnvironment(
                Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT"));
    }

    ...
}
```

The environment can also be set directly on the host builder in a custom WebApplicationFactory<TEntryPoint>:

```
public class CustomWebApplicationFactory<TStartup>
    : WebApplicationFactory<TStartup> where TStartup: class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.UseEnvironment(
            Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT"));

        ...
    }

    ...
```

How the test infrastructure infers the app content root path

The `WebApplicationFactory` constructor infers the app content root path by searching for a WebApplicationFactoryContentRootAttribute on the assembly containing the integration tests with a key equal to the `TEntryPoint` assembly `System.Reflection.Assembly.FullName`. In case an attribute with the correct key isn't found, `WebApplicationFactory` falls back to searching for a solution file (*.sln*) and appends the `TEntryPoint` assembly name to the solution directory. The app root directory (the content root path) is used to discover views and content files.

Disable shadow copying

Shadow copying causes the tests to execute in a different directory than the output directory. For tests to work properly, shadow copying must be disabled. The sample app uses xUnit and disables shadow copying for xUnit by including an *xunit.runner.json* file with the correct configuration setting. For more information, see Configuring xUnit with JSON.

Add the *xunit.runner.json* file to root of the test project with the following content:

```
{
  "shadowCopy": false
}
```

If using Visual Studio, set the file's **Copy to Output Directory** property to **Copy always**. If not using Visual Studio, add a `Content` target to the test app's project file:

```
<ItemGroup>
  <Content Update="xunit.runner.json">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </Content>
</ItemGroup>
```

Disposal of objects

After the tests of the `IClassFixture` implementation are executed, TestServer and HttpClient are disposed when xUnit disposes of the WebApplicationFactory. If objects instantiated by the developer require disposal, dispose of them in the `IClassFixture` implementation. For more information, see Implementing a Dispose method.

Integration tests sample

The sample app is composed of two apps:

APP	PROJECT DIRECTORY	DESCRIPTION
Message app (the SUT)	*src/RazorPagesProject*	Allows a user to add, delete one, delete all, and analyze messages.
Test app	*tests/RazorPagesProject.Tests*	Used to integration test the SUT.

The tests can be run using the built-in test features of an IDE, such as Visual Studio. If using Visual Studio Code or the command line, execute the following command at a command prompt in the *tests/RazorPagesProject.Tests* directory:

```
dotnet test
```

Message app (SUT) organization

The SUT is a Razor Pages message system with the following characteristics:

- The Index page of the app (*Pages/Index.cshtml* and *Pages/Index.cshtml.cs*) provides a UI and page model methods to control the addition, deletion, and analysis of messages (average words per message).
- A message is described by the `Message` class (*Data/Message.cs*) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.
- The app contains a data access layer (DAL) in its database context class, `AppDbContext` (*Data/AppDbContext.cs*).
- If the database is empty on app startup, the message store is initialized with three messages.
- The app includes a `/SecurePage` that can only be accessed by an authenticated user.

†The EF topic, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the *tests/RazorPagesProject.Tests* directory.

TEST APP DIRECTORY	DESCRIPTION
AuthTests	Contains test methods for: • Accessing a secure page by an unauthenticated user. • Accessing a secure page by an authenticated user with a mock AuthenticationHandler<TOptions>. • Obtaining a GitHub user profile and checking the profile's user login.
BasicTests	Contains a test method for routing and content type.
IntegrationTests	Contains the integration tests for the Index page using custom `WebApplicationFactory` class.

TEST APP DIRECTORY	DESCRIPTION
Helpers/Utilities	• *Utilities.cs* contains the `InitializeDbForTests` method used to seed the database with test data. • *HtmlHelpers.cs* provides a method to return an AngleSharp `IHtmlDocument` for use by the test methods. • *HttpClientExtensions.cs* provide overloads for `SendAsync` to submit requests to the SUT.

The test framework is xUnit. Integration tests are conducted using the Microsoft.AspNetCore.TestHost, which includes the TestServer. Because the Microsoft.AspNetCore.Mvc.Testing package is used to configure the test host and test server, the `TestHost` and `TestServer` packages don't require direct package references in the test app's project file or developer configuration in the test app.

Seeding the database for testing

Integration tests usually require a small dataset in the database prior to the test execution. For example, a delete test calls for a database record deletion, so the database must have at least one record for the delete request to succeed.

The sample app seeds the database with three messages in *Utilities.cs* that tests can use when they execute:

```
public static void InitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.AddRange(GetSeedingMessages());
    db.SaveChanges();
}

public static void ReinitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.RemoveRange(db.Messages);
    InitializeDbForTests(db);
}

public static List<Message> GetSeedingMessages()
{
    return new List<Message>()
    {
        new Message(){ Text = "TEST RECORD: You're standing on my scarf." },
        new Message(){ Text = "TEST RECORD: Would you like a jelly baby?" },
        new Message(){ Text = "TEST RECORD: To the rational mind, " +
            "nothing is inexplicable; only unexplained." }
    };
}
```

Additional resources

- Unit tests
- Razor Pages unit tests in ASP.NET Core
- ASP.NET Core Middleware
- Test controller logic in ASP.NET Core

ASP.NET Core load/stress testing

9/22/2020 • 2 minutes to read • Edit Online

Load testing and stress testing are important to ensure a web app is performant and scalable. Their goals are different even though they often share similar tests.

Load tests: Test whether the app can handle a specified load of users for a certain scenario while still satisfying the response goal. The app is run under normal conditions.

Stress tests: Test app stability when running under extreme conditions, often for a long period of time. The tests place high user load, either spikes or gradually increasing load, on the app, or they limit the app's computing resources.

Stress tests determine if an app under stress can recover from failure and gracefully return to expected behavior. Under stress, the app isn't run under normal conditions.

Visual Studio 2019 announced plans to deprecate the load testing. The corresponding Azure DevOps cloud-based load testing service has been closed.

Third-party tools

The following list contains third-party web performance tools with various feature sets:

- Apache JMeter
- ApacheBench (ab)
- Gatling
- k6
- Locust
- West Wind WebSurge
- Netling
- Vegeta

Troubleshoot and debug ASP.NET Core projects

9/22/2020 • 4 minutes to read • Edit Online

By Rick Anderson

The following links provide troubleshooting guidance:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- NDC Conference (London, 2018): Diagnosing issues in ASP.NET Core Applications
- ASP.NET Blog: Troubleshooting ASP.NET Core Performance Problems

.NET Core SDK warnings

Both the 32-bit and 64-bit versions of the .NET Core SDK are installed

In the **New Project** dialog for ASP.NET Core, you may see the following warning:

> Both 32-bit and 64-bit versions of the .NET Core SDK are installed. Only templates from the 64-bit versions installed at 'C:\Program Files\dotnet\sdk\' are displayed.

This warning appears when both 32-bit (x86) and 64-bit (x64) versions of the .NET Core SDK are installed. Common reasons both versions may be installed include:

- You originally downloaded the .NET Core SDK installer using a 32-bit machine but then copied it across and installed it on a 64-bit machine.
- The 32-bit .NET Core SDK was installed by another application.
- The wrong version was downloaded and installed.

Uninstall the 32-bit .NET Core SDK to prevent this warning. Uninstall from **Control Panel** > **Programs and Features** > **Uninstall or change a program**. If you understand why the warning occurs and its implications, you can ignore the warning.

The .NET Core SDK is installed in multiple locations

In the **New Project** dialog for ASP.NET Core, you may see the following warning:

> The .NET Core SDK is installed in multiple locations. Only templates from the SDKs installed at 'C:\Program Files\dotnet\sdk\' are displayed.

You see this message when you have at least one installation of the .NET Core SDK in a directory outside of *C:\Program Files\dotnet\sdk*. Usually this happens when the .NET Core SDK has been deployed on a machine using copy/paste instead of the MSI installer.

Uninstall all 32-bit .NET Core SDKs and runtimes to prevent this warning. Uninstall from **Control Panel** > **Programs and Features** > **Uninstall or change a program**. If you understand why the warning occurs and its implications, you can ignore the warning.

No .NET Core SDKs were detected

- In the Visual Studio **New Project** dialog for ASP.NET Core, you may see the following warning:

  > No .NET Core SDKs were detected, ensure they are included in the environment variable `PATH`.

- When executing a `dotnet` command, the warning appears as:

> It was not possible to find any installed dotnet SDKs.

These warnings appear when the environment variable `PATH` doesn't point to any .NET Core SDKs on the machine. To resolve this problem:

- Install the .NET Core SDK. Obtain the latest installer from .NET Downloads.
- Verify that the `PATH` environment variable points to the location where the SDK is installed (`C:\Program Files\dotnet\` for 64-bit/x64 or `C:\Program Files (x86)\dotnet\` for 32-bit/x86). The SDK installer normally sets the `PATH`. Always install the same bitness SDKs and runtimes on the same machine.

Missing SDK after installing the .NET Core Hosting Bundle

Installing the .NET Core Hosting Bundle modifies the `PATH` when it installs the .NET Core runtime to point to the 32-bit (x86) version of .NET Core (`C:\Program Files (x86)\dotnet\`). This can result in missing SDKs when the 32-bit (x86) .NET Core `dotnet` command is used (No .NET Core SDKs were detected). To resolve this problem, move `C:\Program Files\dotnet\` to a position before `C:\Program Files (x86)\dotnet\` on the `PATH`.

Obtain data from an app

If an app is capable of responding to requests, you can obtain the following data from the app using middleware:

- Request: Method, scheme, host, pathbase, path, query string, headers
- Connection: Remote IP address, remote port, local IP address, local port, client certificate
- Identity: Name, display name
- Configuration settings
- Environment variables

Place the following middleware code at the beginning of the `Startup.Configure` method's request processing pipeline. The environment is checked before the middleware is run to ensure that the code is only executed in the Development environment.

To obtain the environment, use either of the following approaches:

- Inject the `IHostingEnvironment` into the `Startup.Configure` method and check the environment with the local variable. The following sample code demonstrates this approach.
- Assign the environment to a property in the `Startup` class. Check the environment using the property (for example, `if (Environment.IsDevelopment())`).

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IConfiguration config)
{
    if (env.IsDevelopment())
    {
        app.Run(async (context) =>
        {
            var sb = new StringBuilder();
            var nl = System.Environment.NewLine;
            var rule = string.Concat(nl, new string('-', 40), nl);
            var authSchemeProvider = app.ApplicationServices
                .GetRequiredService<IAuthenticationSchemeProvider>();

            sb.Append($"Request{rule}");
            sb.Append($"{DateTimeOffset.Now}{nl}");
            sb.Append($"{context.Request.Method} {context.Request.Path}{nl}");
            sb.Append($"Scheme: {context.Request.Scheme}{nl}");
```

```
            sb.Append($"Host: {context.Request.Headers["Host"]}{nl}");
            sb.Append($"PathBase: {context.Request.PathBase.Value}{nl}");
            sb.Append($"Path: {context.Request.Path.Value}{nl}");
            sb.Append($"Query: {context.Request.QueryString.Value}{nl}{nl}");

            sb.Append($"Connection{rule}");
            sb.Append($"RemoteIp: {context.Connection.RemoteIpAddress}{nl}");
            sb.Append($"RemotePort: {context.Connection.RemotePort}{nl}");
            sb.Append($"LocalIp: {context.Connection.LocalIpAddress}{nl}");
            sb.Append($"LocalPort: {context.Connection.LocalPort}{nl}");
            sb.Append($"ClientCert: {context.Connection.ClientCertificate}{nl}{nl}");

            sb.Append($"Identity{rule}");
            sb.Append($"User: {context.User.Identity.Name}{nl}");
            var scheme = await authSchemeProvider
                .GetSchemeAsync(IISDefaults.AuthenticationScheme);
            sb.Append($"DisplayName: {scheme?.DisplayName}{nl}{nl}");

            sb.Append($"Headers{rule}");
            foreach (var header in context.Request.Headers)
            {
                sb.Append($"{header.Key}: {header.Value}{nl}");
            }
            sb.Append(nl);

            sb.Append($"Websockets{rule}");
            if (context.Features.Get<IHttpUpgradeFeature>() != null)
            {
                sb.Append($"Status: Enabled{nl}{nl}");
            }
            else
            {
                sb.Append($"Status: Disabled{nl}{nl}");
            }

            sb.Append($"Configuration{rule}");
            foreach (var pair in config.AsEnumerable())
            {
                sb.Append($"{pair.Path}: {pair.Value}{nl}");
            }
            sb.Append(nl);

            sb.Append($"Environment Variables{rule}");
            var vars = System.Environment.GetEnvironmentVariables();
            foreach (var key in vars.Keys.Cast<string>().OrderBy(key => key,
                StringComparer.OrdinalIgnoreCase))
            {
                var value = vars[key];
                sb.Append($"{key}: {value}{nl}");
            }

            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync(sb.ToString());
        });
    }
}
```

Debug ASP.NET Core apps

The following links provide information on debugging ASP.NET Core apps.

- Debugging ASP Core on Linux
- Debugging .NET Core on Unix over SSH
- Quickstart: Debug ASP.NET with the Visual Studio debugger
- See this GitHub issue for more debugging information.

Logging in .NET Core and ASP.NET Core

9/22/2020 • 57 minutes to read • Edit Online

By Kirk Larkin, Juergen Gutsch and Rick Anderson

.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. This article shows how to use the logging API with built-in providers.

Most of the code examples shown in this article are from ASP.NET Core apps. The logging-specific parts of these code snippets apply to any .NET Core app that uses the Generic Host. The ASP.NET Core web app templates use the Generic Host.

View or download sample code (how to download)

Logging providers

Logging providers store logs, except for the `Console` provider which displays logs. For example, the Azure Application Insights provider stores logs in Azure Application Insights. Multiple providers can be enabled.

The default ASP.NET Core web app templates:

- Use the Generic Host.
- Call CreateDefaultBuilder, which adds the following logging providers:
  - Console
  - Debug
  - EventSource
  - EventLog: Windows only

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The preceding code shows the `Program` class created with the ASP.NET Core web app templates. The next several sections provide samples based on the ASP.NET Core web app templates, which use the Generic Host. Non-host console apps are discussed later in this document.

To override the default set of logging providers added by `Host.CreateDefaultBuilder`, call `ClearProviders` and add the required logging providers. For example, the following code:

- Calls ClearProviders to remove all the ILoggerProvider instances from the builder.
- Adds the Console logging provider.

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

For additional providers, see:

- Built-in logging providers
- Third-party logging providers.

Create logs

To create logs, use an ILogger<TCategoryName> object from dependency injection (DI).

The following example:

- Creates a logger, `ILogger<AboutModel>`, which uses a log *category* of the fully qualified name of the type `AboutModel`. The log category is a string that is associated with each log.
- Calls LogInformation to log at the `Information` level. The Log *level* indicates the severity of the logged event.

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
    public string Message { get; set; }

    public void OnGet()
    {
        Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
        _logger.LogInformation(Message);
    }
}
```

Levels and categories are explained in more detail later in this document.

For information on Blazor, see Create logs in Blazor and Blazor WebAssembly in this document.

Create logs in Main and Startup shows how to create logs in `Main` and `Startup`.

Configure logging

Logging configuration is commonly provided by the `Logging` section of *appsettings*.`{Environment}` *.json* files. The following *appsettings.Development.json* file is generated by the ASP.NET Core web app templates:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

In the preceding JSON:

- The `"Default"`, `"Microsoft"`, and `"Microsoft.Hosting.Lifetime"` categories are specified.
- The `"Microsoft"` category applies to all categories that start with `"Microsoft"`. For example, this setting applies to the `"Microsoft.AspNetCore.Routing.EndpointMiddleware"` category.
- The `"Microsoft"` category logs at log level `Warning` and higher.
- The `"Microsoft.Hosting.Lifetime"` category is more specific than the `"Microsoft"` category, so the `"Microsoft.Hosting.Lifetime"` category logs at log level "Information" and higher.
- A specific log provider is not specified, so `LogLevel` applies to all the enabled logging providers except for the Windows EventLog.

The `Logging` property can have LogLevel and log provider properties. The `LogLevel` specifies the minimum level to log for selected categories. In the preceding JSON, `Information` and `Warning` log levels are specified. `LogLevel` indicates the severity of the log and ranges from 0 to 6:

`Trace` = 0, `Debug` = 1, `Information` = 2, `Warning` = 3, `Error` = 4, `Critical` = 5, and `None` = 6.

When a `LogLevel` is specified, logging is enabled for messages at the specified level and higher. In the preceding JSON, the `Default` category is logged for `Information` and higher. For example, `Information`, `Warning`, `Error`, and `Critical` messages are logged. If no `LogLevel` is specified, logging defaults to the `Information` level. For more information, see Log levels.

A provider property can specify a `LogLevel` property. `LogLevel` under a provider specifies levels to log for that provider, and overrides the non-provider log settings. Consider the following *appsettings.json* file:

```
{
  "Logging": {
    "LogLevel": { // All providers, LogLevel applies to all the enabled providers.
      "Default": "Error", // Default logging, Error and higher.
      "Microsoft": "Warning" // All Microsoft* categories, Warning and higher.
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information", // Overrides preceding LogLevel:Default setting.
        "Microsoft.Hosting": "Trace" // Debug:Microsoft.Hosting category.
      }
    },
    "EventSource": { // EventSource provider
      "LogLevel": {
        "Default": "Warning" // All categories of EventSource provider.
      }
    }
  }
}
```

Settings in `Logging.{providername}.LogLevel` override settings in `Logging.LogLevel`. In the preceding JSON, the `Debug` provider's default log level is set to `Information`:

`Logging:Debug:LogLevel:Default:Information`

The preceding setting specifies the `Information` log level for every `Logging:Debug:` category except `Microsoft.Hosting`. When a specific category is listed, the specific category overrides the default category. In the preceding JSON, the `Logging:Debug:LogLevel` categories `"Microsoft.Hosting"` and `"Default"` override the settings in `Logging:LogLevel`

The minimum log level can be specified for any of:

- Specific providers: For example, `Logging:EventSource:LogLevel:Default:Information`
- Specific categories: For example, `Logging:LogLevel:Microsoft:Warning`
- All providers and all categories: `Logging:LogLevel:Default:Warning`

Any logs below the minimum level are ***not***:

- Passed to the provider.
- Logged or displayed.

To suppress all logs, specify LogLevel.None. `LogLevel.None` has a value of 6, which is higher than `LogLevel.Critical` (5).

If a provider supports log scopes, `IncludeScopes` indicates whether they're enabled. For more information, see log scopes

The following *appsettings.json* file contains all the providers enabled by default:

```
{
  "Logging": {
    "LogLevel": { // No provider, LogLevel applies to all the enabled providers.
      "Default": "Error",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Warning"
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information" // Overrides preceding LogLevel:Default setting.
      }
    },
    "Console": {
      "IncludeScopes": true,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "EventSource": {
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "EventLog": {
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "AzureAppServicesFile": {
      "IncludeScopes": true,
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "AzureAppServicesBlob": {
      "IncludeScopes": true,
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "ApplicationInsights": {
      "LogLevel": {
        "Default": "Information"
      }
    }
  }
}
```

In the preceding sample:

- The categories and levels are not suggested values. The sample is provided to show all the default providers.
- Settings in `Logging.{providername}.LogLevel` override settings in `Logging.LogLevel`. For example, the level in `Debug.LogLevel.Default` overrides the level in `LogLevel.Default`.
- Each default provider *alias* is used. Each provider defines an *alias* that can be used in configuration in place of the fully qualified type name. The built-in providers aliases are:
  - Console
  - Debug
  - EventSource

- EventLog
- AzureAppServicesFile
- AzureAppServicesBlob
- ApplicationInsights

Set log level by command line, environment variables, and other configuration

Log level can be set by any of the configuration providers.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

The following commands:

- Set the environment key `Logging:LogLevel:Microsoft` to a value of `Information` on Windows.
- Test the settings when using an app created with the ASP.NET Core web application templates. The `dotnet run` command must be run in the project directory after using `set`.

```
set Logging__LogLevel__Microsoft=Information
dotnet run
```

The preceding environment setting:

- Is only set in processes launched from the command window they were set in.
- Isn't read by browsers launched with Visual Studio.

The following setx command also sets the environment key and value on Windows. Unlike `set`, `setx` settings are persisted. The `/M` switch sets the variable in the system environment. If `/M` isn't used, a user environment variable is set.

```
setx Logging__LogLevel__Microsoft=Information /M
```

On Azure App Service, select **New application setting** on the **Settings** > **Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

For more information, see Azure Apps: Override app configuration using the Azure Portal.

For more information on setting ASP.NET Core configuration values using environment variables, see environment variables. For information on using other configuration sources, including the command line, Azure Key Vault, Azure App Configuration, other file formats, and more, see Configuration in ASP.NET Core.

How filtering rules are applied

When an ILogger<TCategoryName> object is created, the ILoggerFactory object selects a single rule per provider to apply to that logger. All messages written by an `ILogger` instance are filtered based on the selected rules. The most specific rule for each provider and category pair is selected from the

available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If no match is found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If no match is found, select all rules that don't specify a category.
- If multiple rules are selected, take the **last** one.
- If no rules are selected, use `MinimumLevel`.

Logging output from dotnet run and Visual Studio

Logs created with the default logging providers are displayed:

- In Visual Studio
  - In the Debug output window when debugging.
  - In the ASP.NET Core Web Server window.
- In the console window when the app is run with `dotnet run`.

Logs that begin with "Microsoft" categories are from ASP.NET Core framework code. ASP.NET Core and application code use the same logging API and providers.

Log category

When an `ILogger` object is created, a *category* is specified. That category is included with each log message created by that instance of `ILogger`. The category string is arbitrary, but the convention is to use the class name. For example, in a controller the name might be `"TodoApi.Controllers.TodoController"`. The ASP.NET Core web apps use `ILogger<T>` to automatically get an `ILogger` instance that uses the fully qualified type name of `T` as the category:

```
public class PrivacyModel : PageModel
{
    private readonly ILogger<PrivacyModel> _logger;

    public PrivacyModel(ILogger<PrivacyModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.PrivacyModel called.");
    }
}
```

To explicitly specify the category, call `ILoggerFactory.CreateLogger`:

```
public class ContactModel : PageModel
{
    private readonly ILogger _logger;

    public ContactModel(ILoggerFactory logger)
    {
        _logger = logger.CreateLogger("MyCategory");
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.ContactModel called.");
    }
```

Calling `CreateLogger` with a fixed name can be useful when used in multiple methods so the events can be organized by category.

`ILogger<T>` is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

The following table lists the LogLevel values, the convenience `Log{LogLevel}` extension method, and the suggested usage:

LOGLEVEL	VALUE	METHOD	DESCRIPTION
Trace	0	LogTrace	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should ***not*** be enabled in production.
Debug	1	LogDebug	For debugging and development. Use with caution in production due to the high volume.
Information	2	LogInformation	Tracks the general flow of the app. May have long-term value.
Warning	3	LogWarning	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	LogError	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.

LOGLEVEL	VALUE	METHOD	DESCRIPTION
Critical	5	LogCritical	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category should not write any messages.

In the previous table, the `LogLevel` is listed from lowest to highest severity.

The Log method's first parameter, LogLevel, indicates the severity of the log. Rather than calling `Log(LogLevel, ...)`, most developers call the Log{LogLevel} extension methods. The `Log{LogLevel}` extension methods call the Log method and specify the LogLevel. For example, the following two logging calls are functionally equivalent and produce the same log:

```
[HttpGet]
public IActionResult Test1(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);

    _logger.Log(LogLevel.Information, MyLogEvents.TestItem, routeInfo);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    return ControllerContext.MyDisplayRouteInfo();
}
```

`MyLogEvents.TestItem` is the event ID. `MyLogEvents` is part of the sample app and is displayed in the Log event ID section.

MyDisplayRouteInfo and ToCtxString are provided by the Rick.Docs.Samples.RouteInfo NuGet package. The methods display `Controller` route information.

The following code creates `Information` and `Warning` logs:

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

In the preceding code, the first `Log{LogLevel}` parameter, `MyLogEvents.GetItem`, is the Log event ID. The second parameter is a message template with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in the message template section later in this document.

Call the appropriate `Log{LogLevel}` method to control how much log output is written to a particular storage medium. For example:

- In production:
  - Logging at the `Trace` or `Information` levels produces a high-volume of detailed log messages. To control costs and not exceed data storage limits, log `Trace` and `Information` level messages to a high-volume, low-cost data store. Consider limiting `Trace` and `Information` to specific categories.
  - Logging at `Warning` through `Critical` levels should produce few log messages.
    - Costs and storage limits usually aren't a concern.
    - Few logs allow more flexibility in data store choices.
- In development:
  - Set to `Warning`.
  - Add `Trace` or `Information` messages when troubleshooting. To limit output, set `Trace` or `Information` only for the categories under investigation.

ASP.NET Core writes logs for framework events. For example, consider the log output for:

- A Razor Pages app created with the ASP.NET Core templates.
- Logging set to `Logging:Console:LogLevel:Microsoft:Information`
- Navigation to the Privacy page:

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 GET https://localhost:5001/Privacy
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint '/Privacy'
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
      Route matched with {page = "/Privacy"}. Executing page /Privacy
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
      Executing handler method DefaultRP.Pages.PrivacyModel.OnGet - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
      Executed handler method OnGet, returned result .
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[103]
      Executing an implicit handler method - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[104]
      Executed an implicit handler method, returned result
Microsoft.AspNetCore.Mvc.RazorPages.PageResult.
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
      Executed page /Privacy in 74.5188ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint '/Privacy'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 149.3023ms 200 text/html; charset=utf-8
```

The following JSON sets `Logging:Console:LogLevel:Microsoft:Information`:

```
{
  "Logging": {      // Default, all providers.
    "LogLevel": {
      "Microsoft": "Warning"
    },
    "Console": { // Console provider.
      "LogLevel": {
        "Microsoft": "Information"
      }
    }
  }
}
```

Log event ID

Each log can specify an *event ID*. The sample app uses the `MyLogEvents` class to define event IDs:

```
public class MyLogEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems     = 1001;
    public const int GetItem       = 1002;
    public const int InsertItem    = 1003;
    public const int UpdateItem    = 1004;
    public const int DeleteItem    = 1005;

    public const int TestItem      = 3000;

    public const int GetItemNotFound    = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

An event ID associates a set of events. For example, all logs related to displaying a list of items on a page might be 1001.

The logging provider may store the event ID in an ID field, in the logging message, or not at all. The Debug provider doesn't show event IDs. The console provider shows event IDs in brackets after the category:

```
info: TodoApi.Controllers.TodoItemsController[1002]
      Getting item 1
warn: TodoApi.Controllers.TodoItemsController[4000]
      Get(1) NOT FOUND
```

Some logging providers store the event ID in a field, which allows for filtering on the ID.

Log message template

Each log API uses a message template. The message template can contain placeholders for which arguments are provided. Use names for the placeholders, not numbers.

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

The order of placeholders, not their names, determines which parameters are used to provide their values. In the following code, the parameter names are out of sequence in the message template:

```
string p1 = "param1";
string p2 = "param2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

The preceding code creates a log message with the parameter values in sequence:

```
Parameter values: param1, param2
```

This approach allows logging providers to implement semantic or structured logging. The arguments themselves are passed to the logging system, not just the formatted message template. This enables logging providers to store the parameter values as fields. For example, consider the following logger method:

```
_logger.LogInformation("Getting item {Id} at {RequestTime}", id, DateTime.Now);
```

For example, when logging to Azure Table Storage:

- Each Azure Table entity can have `ID` and `RequestTime` properties.
- Tables with properties simplify queries on logged data. For example, a query can find all logs within a particular `RequestTime` range without having to parse the time out of the text message.

Log exceptions

The logger methods have overloads that take an exception parameter:

```
[HttpGet("{id}")]
public IActionResult TestExp(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    try
    {
        if (id == 3)
        {
            throw new Exception("Test exception");
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, ex, "TestExp({Id})", id);
        return NotFound();
    }

    return ControllerContext.MyDisplayRouteInfo();
}
```

MyDisplayRouteInfo and ToCtxString are provided by the Rick.Docs.Samples.RouteInfo NuGet package. The methods display `Controller` route information.

Exception logging is provider-specific.

Default log level

If the default log level is not set, the default log level value is `Information`.

For example, consider the following web app:

- Created with the ASP.NET web app templates.
- *appsettings.json* and *appsettings.Development.json* deleted or renamed.

With the preceding setup, navigating to the privacy or home page produces many `Trace`, `Debug`, and `Information` messages with `Microsoft` in the category name.

The following code sets the default log level when the default log level is not set in configuration:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning))
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Generally, log levels should be specified in configuration and not code.

Filter function

A filter function is invoked for all providers and categories that don't have rules assigned to them by configuration or code:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
            {
                logging.AddFilter((provider, category, logLevel) =>
                {
                    if (provider.Contains("ConsoleLoggerProvider")
                        && category.Contains("Controller")
                        && logLevel >= LogLevel.Information)
                    {
                        return true;
                    }
                    else if (provider.Contains("ConsoleLoggerProvider")
                        && category.Contains("Microsoft")
                        && logLevel >= LogLevel.Information)
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                });
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The preceding code displays console logs when the category contains `Controller` or `Microsoft` and the log level is `Information` or higher.

Generally, log levels should be specified in configuration and not code.

ASP.NET Core and EF Core categories

The following table contains some categories used by ASP.NET Core and Entity Framework Core, with notes about the logs:

CATEGORY	NOTES
Microsoft.AspNetCore	General ASP.NET Core diagnostics.
Microsoft.AspNetCore.DataProtection	Which keys were considered, found, and used.
Microsoft.AspNetCore.HostFiltering	Hosts allowed.
Microsoft.AspNetCore.Hosting	How long HTTP requests took to complete and what time they started. Which hosting startup assemblies were loaded.

CATEGORY	NOTES
Microsoft.AspNetCore.Mvc	MVC and Razor diagnostics. Model binding, filter execution, view compilation, action selection.
Microsoft.AspNetCore.Routing	Route matching information.
Microsoft.AspNetCore.Server	Connection start, stop, and keep alive responses. HTTPS certificate information.
Microsoft.AspNetCore.StaticFiles	Files served.
Microsoft.EntityFrameworkCore	General Entity Framework Core diagnostics. Database activity and configuration, change detection, migrations.

To view more categories in the console window, set **appsettings.Development.json** to the following:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Trace",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Log scopes

A *scope* can group a set of logical operations. This grouping can be used to attach the same data to each log that's created as part of a set. For example, every log created as part of processing a transaction can include the transaction ID.

A scope:

- Is an IDisposable type that's returned by the BeginScope method.
- Lasts until it's disposed.

The following providers support scopes:

- `Console`
- AzureAppServicesFile and AzureAppServicesBlob

Use a scope by wrapping logger calls in a `using` block:

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    TodoItem todoItem;

    using (_logger.BeginScope("using block message"))
    {
        _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

        todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            _logger.LogWarning(MyLogEvents.GetItemNotFound,
                "Get({Id}) NOT FOUND", id);
            return NotFound();
        }
    }

    return ItemToDTO(todoItem);
}
```

The following JSON enables scopes for the console provider:

```
{
  "Logging": {
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "IncludeScopes": true, // Required to use Scopes.
      "LogLevel": {
        "Microsoft": "Warning",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

The following code enables scopes for the console provider:

```
public class Scopes
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging((hostingContext, logging) =>
            {
                logging.ClearProviders();
                logging.AddConsole(options => options.IncludeScopes = true);
                logging.AddDebug();
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Generally, logging should be specified in configuration and not code.

Built-in logging providers

ASP.NET Core includes the following logging providers as part of the shared framework:

- Console
- Debug
- EventSource
- EventLog

The following logging providers are shipped by Microsoft, but not as part of the shared framework. They must be installed as additional nuget.

- AzureAppServicesFile and AzureAppServicesBlob
- ApplicationInsights

For information on `stdout` and debug logging with the ASP.NET Core Module, see Troubleshoot ASP.NET Core on Azure App Service and IIS and ASP.NET Core Module.

Console

The `Console` provider logs output to the console. For more information on viewing `Console` logs in development, see Logging output from dotnet run and Visual Studio.

Debug

The `Debug` provider writes log output by using the System.Diagnostics.Debug class. Calls to `System.Diagnostics.Debug.WriteLine` write to the `Debug` provider.

On Linux, the `Debug` provider log location is distribution-dependent and may be one of the following:

- */var/log/message*
- */var/log/syslog*

Event Source

The `EventSource` provider writes to a cross-platform event source with the name `Microsoft-Extensions-Logging`. On Windows, the provider uses ETW.

dotnet trace tooling

The dotnet-trace tool is a cross-platform CLI global tool that enables the collection of .NET Core traces of a running process. The tool collects Microsoft.Extensions.Logging.EventSource provider data using a LoggingEventSource.

See dotnet-trace for installation instructions.

Use the dotnet trace tooling to collect a trace from an app:

1. Run the app with the `dotnet run` command.
2. Determine the process identifier (PID) of the .NET Core app:
   - On Windows, use one of the following approaches:
     - Task Manager (Ctrl+Alt+Del)
     - tasklist command
     - Get-Process Powershell command
   - On Linux, use the pidof command.

   Find the PID for the process that has the same name as the app's assembly.
3. Execute the `dotnet trace` command.

   General command syntax:

   ```
   dotnet trace collect -p {PID}
       --providers Microsoft-Extensions-Logging:{Keyword}:{Provider Level}
           :FilterSpecs=\"
               {Logger Category 1}:{Category Level 1};
               {Logger Category 2}:{Category Level 2};
               ...
               {Logger Category N}:{Category Level N}\"
   ```

   When using a PowerShell command shell, enclose the `--providers` value in single quotes (`'`):

   ```
   dotnet trace collect -p {PID}
       --providers 'Microsoft-Extensions-Logging:{Keyword}:{Provider Level}
           :FilterSpecs=\"
               {Logger Category 1}:{Category Level 1};
               {Logger Category 2}:{Category Level 2};
               ...
               {Logger Category N}:{Category Level N}\"'
   ```

   On non-Windows platforms, add the `-f speedscope` option to change the format of the output trace file to `speedscope`.

   The following table defines the Keyword:

KEYWORD	DESCRIPTION
1	Log meta events about the `LoggingEventSource`. Doesn't log events from `ILogger`.
2	Turns on the `Message` event when `ILogger.Log()` is called. Provides information in a programmatic (not formatted) way.

KEYWORD	DESCRIPTION
4	Turns on the `FormatMessage` event when `ILogger.Log()` is called. Provides the formatted string version of the information.
8	Turns on the `MessageJson` event when `ILogger.Log()` is called. Provides a JSON representation of the arguments.

The following table lists the provider levels:

PROVIDER LEVEL	DESCRIPTION
0	`LogAlways`
1	`Critical`
2	`Error`
3	`Warning`
4	`Informational`
5	`Verbose`

The parsing for a category level can be either a string or a number:

CATEGORY NAMED VALUE	NUMERIC VALUE
`Trace`	0
`Debug`	1
`Information`	2
`Warning`	3
`Error`	4
`Critical`	5

The provider level and category level:

- Are in reverse order.
- The string constants aren't all identical.

If no `FilterSpecs` are specified then the `EventSourceLogger` implementation attempts to convert the provider level to a category level and applies it to all categories.

PROVIDER LEVEL	CATEGORY LEVEL
`Verbose` (5)	`Debug` (1)

PROVIDER LEVEL	CATEGORY LEVEL
`Informational` (4)	`Information` (2)
`Warning` (3)	`Warning` (3)
`Error` (2)	`Error` (4)
`Critical` (1)	`Critical` (5)

If `FilterSpecs` are provided, any category that is included in the list uses the category level encoded there, all other categories are filtered out.

The following examples assume:

- An app is running and calling `logger.LogDebug("12345")`.
- The process ID (PID) has been set via `set PID=12345`, where `12345` is the actual PID.

Consider the following command:

```
dotnet trace collect -p %PID% --providers Microsoft-Extensions-Logging:4:5
```

The preceding command:

- Captures debug messages.
- Doesn't apply a `FilterSpecs`.
- Specifies level 5 which maps category Debug.

Consider the following command:

```
dotnet trace collect -p %PID%  --providers Microsoft-Extensions-
Logging:4:5:\"FilterSpecs=*:5\"
```

The preceding command:

- Doesn't capture debug messages because the category level 5 is `Critical`.
- Provides a `FilterSpecs`.

The following command captures debug messages because category level 1 specifies `Debug`.

```
dotnet trace collect -p %PID%  --providers Microsoft-Extensions-
Logging:4:5:\"FilterSpecs=*:1\"
```

The following command captures debug messages because category specifies `Debug`.

```
dotnet trace collect -p %PID%  --providers Microsoft-Extensions-
Logging:4:5:\"FilterSpecs=*:Debug\"
```

`FilterSpecs` entries for `{Logger Category}` and `{Category Level}` represent additional log filtering conditions. Separate `FilterSpecs` entries with the `;` semicolon character.

Example using a Windows command shell:

```
dotnet trace collect -p %PID% --providers Microsoft-Extensions-
Logging:4:2:FilterSpecs=\"Microsoft.AspNetCore.Hosting*:4\"
```

The preceding command activates:

- The Event Source logger to produce formatted strings (`4`) for errors (`2`).
- `Microsoft.AspNetCore.Hosting` logging at the `Informational` logging level (`4`).

4. Stop the dotnet trace tooling by pressing the Enter key or Ctrl+C.

   The trace is saved with the name *trace.nettrace* in the folder where the `dotnet trace` command is executed.

5. Open the trace with Perfview. Open the *trace.nettrace* file and explore the trace events.

If the app doesn't build the host with `CreateDefaultBuilder`, add the Event Source provider to the app's logging configuration.

For more information, see:

- Trace for performance analysis utility (dotnet-trace) (.NET Core documentation)
- Trace for performance analysis utility (dotnet-trace) (dotnet/diagnostics GitHub repository documentation)
- LoggingEventSource Class (.NET API Browser)
- EventLevel
- LoggingEventSource reference source (3.0): To obtain reference source for a different version, change the branch to `release/{Version}`, where `{Version}` is the version of ASP.NET Core desired.
- Perfview: Useful for viewing Event Source traces.

Perfview

Use the PerfView utility to collect and view logs. There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET Core.

To configure PerfView for collecting events logged by this provider, add the string `*Microsoft-Extensions-Logging` to the **Additional Providers** list. Don't miss the `*` at the start of the string.

Windows EventLog

The `EventLog` provider sends log output to the Windows Event Log. Unlike the other providers, the `EventLog` provider does ***not*** inherit the default non-provider settings. If `EventLog` log settings aren't specified, they default to LogLevel.Warning.

To log events lower than LogLevel.Warning, explicitly set the log level. The following example sets the Event Log default log level to LogLevel.Information:

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

AddEventLog overloads can pass in EventLogSettings. If `null` or not specified, the following default settings are used:

- `LogName`: "Application"

- `SourceName`: ".NET Runtime"
- `MachineName`: The local machine name is used.

The following code changes the `SourceName` from the default value of `".NET Runtime"` to `MyLogs`:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
            {
                logging.AddEventLog(eventLogSettings =>
                {
                    eventLogSettings.SourceName = "MyLogs";
                });
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Azure App Service

The Microsoft.Extensions.Logging.AzureAppServices provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account.

The provider package isn't included in the shared framework. To use the provider, add the provider package to the project.

To configure provider settings, use AzureFileLoggerOptions and AzureBlobLoggerOptions, as shown in the following example:

```
public class Scopes
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureLogging(logging => logging.AddAzureWebAppDiagnostics())
                .ConfigureServices(serviceCollection => serviceCollection
                    .Configure<AzureFileLoggerOptions>(options =>
                    {
                        options.FileName = "azure-diagnostics-";
                        options.FileSizeLimit = 50 * 1024;
                        options.RetainedFileCountLimit = 5;
                    })
                    .Configure<AzureBlobLoggerOptions>(options =>
                    {
                        options.BlobName = "log.txt";
                    }))
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

When deployed to Azure App Service, the app uses the settings in the App Service logs section of the **App Service** page of the Azure portal. When the following settings are updated, the changes take effect immediately without requiring a restart or redeployment of the app.

- **Application Logging (Filesystem)**
- **Application Logging (Blob)**

The default location for log files is in the *D:\home\LogFiles\Application* folder, and the default file name is *diagnostics-yyyymmdd.txt*. The default file size limit is 10 MB, and the default maximum number of files retained is 2. The default blob name is *{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt*.

This provider only logs when the project runs in the Azure environment.

Azure log streaming

Azure log streaming supports viewing log activity in real time from:

- The app server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the **App Service logs** page from the app's portal page.
- Set **Application Logging (Filesystem)** to **On**.
- Choose the log **Level**. This setting only applies to Azure log streaming.

Navigate to the **Log Stream** page to view logs. The logged messages are logged with the `ILogger` interface.

Azure Application Insights

The Microsoft.Extensions.Logging.ApplicationInsights provider package writes logs to Azure Application Insights. Application Insights is a service that monitors a web app and provides tools for querying and analyzing the telemetry data. If you use this provider, you can query and analyze your logs by using the Application Insights tools.

The logging provider is included as a dependency of Microsoft.ApplicationInsights.AspNetCore, which is the package that provides all available telemetry for ASP.NET Core. If you use this package, you don't have to install the provider package.

The Microsoft.ApplicationInsights.Web package is for ASP.NET 4.x, not ASP.NET Core.

For more information, see the following resources:

- Application Insights overview
- Application Insights for ASP.NET Core applications - Start here if you want to implement the full range of Application Insights telemetry along with logging.
- ApplicationInsightsLoggerProvider for .NET Core ILogger logs - Start here if you want to implement the logging provider without the rest of Application Insights telemetry.
- Application Insights logging adapters.
- Install, configure, and initialize the Application Insights SDK - Interactive tutorial on the Microsoft Learn site.

Third-party logging providers

Third-party logging frameworks that work with ASP.NET Core:

- elmah.io (GitHub repo)
- Gelf (GitHub repo)
- JSNLog (GitHub repo)
- KissLog.net (GitHub repo)
- Log4Net (GitHub repo)
- Loggr (GitHub repo)
- NLog (GitHub repo)
- PLogger (GitHub repo)
- Sentry (GitHub repo)
- Serilog (GitHub repo)
- Stackdriver (Github repo)

Some third-party frameworks can perform semantic logging, also known as structured logging.

Using a third-party framework is similar to using one of the built-in providers:

1. Add a NuGet package to your project.
2. Call an `ILoggerFactory` extension method provided by the logging framework.

For more information, see each provider's documentation. Third-party logging providers aren't supported by Microsoft.

Non-host console app

For an example of how to use the Generic Host in a non-web console app, see the *Program.cs* file of the Background Tasks sample app (Background tasks with hosted services in ASP.NET Core).

Logging code for apps without Generic Host differs in the way providers are added and loggers are created.

Logging providers

In a non-host console app, call the provider's `Add{provider name}` extension method while creating a `LoggerFactory`:

```
class Program
{
    static void Main(string[] args)
    {
        using var loggerFactory = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter("Microsoft", LogLevel.Warning)
                .AddFilter("System", LogLevel.Warning)
                .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
                .AddConsole()
                .AddEventLog();
        });
        ILogger logger = loggerFactory.CreateLogger<Program>();
        logger.LogInformation("Example log message");
    }
}
```

Create logs

To create logs, use an ILogger<TCategoryName> object. Use the `LoggerFactory` to create an `ILogger`.

The following example creates a logger with `LoggingConsoleApp.Program` as the category.

```
class Program
{
    static void Main(string[] args)
    {
        using var loggerFactory = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter("Microsoft", LogLevel.Warning)
                .AddFilter("System", LogLevel.Warning)
                .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
                .AddConsole()
                .AddEventLog();
        });
        ILogger logger = loggerFactory.CreateLogger<Program>();
        logger.LogInformation("Example log message");
    }
}
```

In the following example, the logger is used to create logs with `Information` as the level. The Log *level* indicates the severity of the logged event.

```
class Program
{
    static void Main(string[] args)
    {
        using var loggerFactory = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter("Microsoft", LogLevel.Warning)
                .AddFilter("System", LogLevel.Warning)
                .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
                .AddConsole()
                .AddEventLog();
        });
        ILogger logger = loggerFactory.CreateLogger<Program>();
        logger.LogInformation("Example log message");
    }
}
```

Levels and categories are explained in more detail in this document.

Log during host construction

Logging during host construction isn't directly supported. However, a separate logger can be used. In the following example, a Serilog logger is used to log in `CreateHostBuilder`. `AddSerilog` uses the static configuration specified in `Log.Logger`:

```
using System;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args)
    {
        var builtConfig = new ConfigurationBuilder()
            .AddJsonFile("appsettings.json")
            .AddCommandLine(args)
            .Build();

        Log.Logger = new LoggerConfiguration()
            .WriteTo.Console()
            .WriteTo.File(builtConfig["Logging:FilePath"])
            .CreateLogger();

        try
        {
            return Host.CreateDefaultBuilder(args)
                .ConfigureServices((context, services) =>
                {
                    services.AddRazorPages();
                })
                .ConfigureAppConfiguration((hostingContext, config) =>
                {
                    config.AddConfiguration(builtConfig);
                })
                .ConfigureLogging(logging =>
                {
                    logging.AddSerilog();
                })
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Host builder error");

            throw;
        }
        finally
        {
            Log.CloseAndFlush();
        }
    }
}
```

Configure a service that depends on ILogger

Constructor injection of a logger into `Startup` works in earlier versions of ASP.NET Core because a separate DI container is created for the Web Host. For information about why only one container is created for the Generic Host, see the breaking change announcement.

To configure a service that depends on `ILogger<T>`, use constructor injection or provide a factory method. The factory method approach is recommended only if there is no other option. For example, consider a service that needs an `ILogger<T>` instance provided by DI:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddRazorPages();

    services.AddSingleton<IMyService>((container) =>
    {
        var logger = container.GetRequiredService<ILogger<MyService>>();
        return new MyService() { Logger = logger };
    });
}
```

The preceding highlighted code is a Func that runs the first time the DI container needs to construct an instance of `MyService`. You can access any of the registered services in this way.

Create logs in Main

The following code logs in `Main` by getting an `ILogger` instance from DI after building the host:

```
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Host created.");

    host.Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

Create logs in Startup

The following code writes logs in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
                      ILogger<Startup> logger)
{
    if (env.IsDevelopment())
    {
        logger.LogInformation("In Development.");
        app.UseDeveloperExceptionPage();
    }
    else
    {
        logger.LogInformation("Not Development.");
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapRazorPages();
    });
}
```

Writing logs before completion of the DI container setup in the `Startup.ConfigureServices` method is not supported:

- Logger injection into the `Startup` constructor is not supported.
- Logger injection into the `Startup.ConfigureServices` method signature is not supported

The reason for this restriction is that logging depends on DI and on configuration, which in turns depends on DI. The DI container isn't set up until `ConfigureServices` finishes.

For information on configuring a service that depends on `ILogger<T>` or why constructor injection of a logger into `Startup` worked in earlier versions, see Configure a service that depends on ILogger

No asynchronous logger methods

Logging should be so fast that it isn't worth the performance cost of asynchronous code. If a logging data store is slow, don't write to it directly. Consider writing the log messages to a fast store initially, then moving them to the slow store later. For example, when logging to SQL Server, don't do so directly in a `Log` method, since the `Log` methods are synchronous. Instead, synchronously add log messages to an in-memory queue and have a background worker pull the messages out of the queue to do the asynchronous work of pushing data to SQL Server. For more information, see this GitHub issue.

Change log levels in a running app

The Logging API doesn't include a scenario to change log levels while an app is running. However, some configuration providers are capable of reloading configuration, which takes immediate effect on logging configuration. For example, the File Configuration Provider, reloads logging configuration by default. If configuration is changed in code while an app is running, the app can call IConfigurationRoot.Reload to update the app's logging configuration.

ILogger and ILoggerFactory

The ILogger<TCategoryName> and ILoggerFactory interfaces and implementations are included in the .NET Core SDK. They are also available in the following NuGet packages:

- The interfaces are in Microsoft.Extensions.Logging.Abstractions.
- The default implementations are in Microsoft.Extensions.Logging.

Apply log filter rules in code

The preferred approach for setting log filter rules is by using Configuration.

The following example shows how to register filter rules in code:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
               logging.AddFilter("System", LogLevel.Debug)
                  .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)
                  .AddFilter<ConsoleLoggerProvider>("Microsoft", LogLevel.Trace))
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

`logging.AddFilter("System", LogLevel.Debug)` specifies the `System` category and log level `Debug`. The filter is applied to all providers because a specific provider was not configured.

`AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)` specifies:

- The `Debug` logging provider.
- Log level `Information` and higher.
- All categories starting with `"Microsoft"`.

Create a custom logger

To add a custom logger, add an ILoggerProvider with ILoggerFactory:

```
public void Configure(
    IApplicationBuilder app,
    IWebHostEnvironment env,
    ILoggerFactory loggerFactory)
{
    loggerFactory.AddProvider(new CustomLoggerProvider(new CustomLoggerConfiguration()));
```

The `ILoggerProvider` creates one or more `ILogger` instances. The `ILogger` instances are used by the framework to log the information.

Sample custom logger configuration

The sample:

- Is designed to be a very basic sample that sets the color of the log console by event ID and log level. Loggers generally don't change by event ID and are not specific to log level.
- Creates different color console entries per log level and event ID using the following configuration type:

```
public class ColorConsoleLoggerConfiguration
{
    public LogLevel LogLevel { get; set; } = LogLevel.Warning;
    public int EventId { get; set; } = 0;
    public ConsoleColor Color { get; set; } = ConsoleColor.Yellow;
}
```

The preceding code sets the default level to `Warning` and the color to `Yellow`. If the `EventId` is set to 0, we will log all events.

Create the custom logger

The `ILogger` implementation category name is typically the logging source. For example, the type where the logger is created:

```
public class ColorConsoleLogger : ILogger
{
    private readonly string _name;
    private readonly ColorConsoleLoggerConfiguration _config;

    public ColorConsoleLogger(string name, ColorConsoleLoggerConfiguration config)
    {
        _name = name;
        _config = config;
    }

    public IDisposable BeginScope<TState>(TState state)
    {
        return null;
    }

    public bool IsEnabled(LogLevel logLevel)
    {
        return logLevel == _config.LogLevel;
    }

    public void Log<TState>(LogLevel logLevel, EventId eventId, TState state,
                        Exception exception, Func<TState, Exception, string> formatter)
    {
        if (!IsEnabled(logLevel))
        {
            return;
        }

        if (_config.EventId == 0 || _config.EventId == eventId.Id)
        {
            var color = Console.ForegroundColor;
            Console.ForegroundColor = _config.Color;
            Console.WriteLine($"{logLevel} - {eventId.Id} " +
                              $"- {_name} - {formatter(state, exception)}");
            Console.ForegroundColor = color;
        }
    }
}
```

The preceding code:

- Creates a logger instance per category name.

- Checks `logLevel == _config.LogLevel` in `IsEnabled`, so each `logLevel` has a unique logger. Generally, loggers should also be enabled for all higher log levels:

```
public bool IsEnabled(LogLevel logLevel)
{
    return logLevel >= _config.LogLevel;
}
```

Create the custom LoggerProvider

The `LoggerProvider` is the class that creates the logger instances. Maybe it is not needed to create a logger instance per category, but this makes sense for some Loggers, like NLog or log4net. Doing this you are also able to choose different logging output targets per category if needed:

```
public class ColorConsoleLoggerProvider : ILoggerProvider
{
    private readonly ColorConsoleLoggerConfiguration _config;
    private readonly ConcurrentDictionary<string, ColorConsoleLogger> _loggers = new
ConcurrentDictionary<string, ColorConsoleLogger>();

    public ColorConsoleLoggerProvider(ColorConsoleLoggerConfiguration config)
    {
        _config = config;
    }

    public ILogger CreateLogger(string categoryName)
    {
        return _loggers.GetOrAdd(categoryName, name => new ColorConsoleLogger(name, _config));
    }

    public void Dispose()
    {
        _loggers.Clear();
    }
}
```

In the preceding code, CreateLogger creates a single instance of the `ColorConsoleLogger` per category name and stores it in the `ConcurrentDictionary<TKey,TValue>`;

Usage and registration of the custom logger

Register the logger in the `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
                      ILoggerFactory loggerFactory)
{
    // Default registration.
    loggerFactory.AddProvider(new ColorConsoleLoggerProvider(
                              new ColorConsoleLoggerConfiguration
    {
        LogLevel = LogLevel.Error,
        Color = ConsoleColor.Red
    }));

    // Custom registration with default values.
    loggerFactory.AddColorConsoleLogger();

    // Custom registration with a new configuration instance.
    loggerFactory.AddColorConsoleLogger(new ColorConsoleLoggerConfiguration
    {
        LogLevel = LogLevel.Debug,
        Color = ConsoleColor.Gray
    });

    // Custom registration with a configuration object.
    loggerFactory.AddColorConsoleLogger(c =>
    {
        c.LogLevel = LogLevel.Information;
        c.Color = ConsoleColor.Blue;
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

For the preceding code, provide at least one extension method for the `ILoggerFactory`:

```
public static class ColorConsoleLoggerExtensions
{
    public static ILoggerFactory AddColorConsoleLogger(
                                      this ILoggerFactory loggerFactory,
                                      ColorConsoleLoggerConfiguration config)
    {
        loggerFactory.AddProvider(new ColorConsoleLoggerProvider(config));
        return loggerFactory;
    }
    public static ILoggerFactory AddColorConsoleLogger(
                                      this ILoggerFactory loggerFactory)
    {
        var config = new ColorConsoleLoggerConfiguration();
        return loggerFactory.AddColorConsoleLogger(config);
    }
    public static ILoggerFactory AddColorConsoleLogger(
                                    this ILoggerFactory loggerFactory,
                                    Action<ColorConsoleLoggerConfiguration> configure)
    {
        var config = new ColorConsoleLoggerConfiguration();
        configure(config);
        return loggerFactory.AddColorConsoleLogger(config);
    }
}
```

Additional resources

- High-performance logging with LoggerMessage in ASP.NET Core
- Logging bugs should be created in the github.com/dotnet/runtime/ repo.
- ASP.NET Core Blazor logging

By Tom Dykstra and Steve Smith

.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. This article shows how to use the logging API with built-in providers.

View or download sample code (how to download)

Add providers

A logging provider displays or stores logs. For example, the Console provider displays logs on the console, and the Azure Application Insights provider stores them in Azure Application Insights. Logs can be sent to multiple destinations by adding multiple providers.

To add a provider, call the provider's `Add{provider name}` extension method in *Program.cs*:

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var env = hostingContext.HostingEnvironment;
            config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                  .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
                      optional: true, reloadOnChange: true);
            config.AddEnvironmentVariables();
        })
        .ConfigureLogging((hostingContext, logging) =>
        {
            // Requires `using Microsoft.Extensions.Logging;`
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
            logging.AddEventSourceLogger();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

The preceding code requires references to `Microsoft.Extensions.Logging` and `Microsoft.Extensions.Configuration`.

The default project template calls CreateDefaultBuilder, which adds the following logging providers:

- Console
- Debug
- EventSource (starting in ASP.NET Core 2.2)

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

If you use `CreateDefaultBuilder`, you can replace the default providers with your own choices. Call ClearProviders, and add the providers you want.

```
public static void Main(string[] args)
{
    var host = CreateWebHostBuilder(args).Build();

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        });
```

Learn more about built-in logging providers and third-party logging providers later in the article.

Create logs

To create logs, use an ILogger<TCategoryName> object. In a web app or hosted service, get an `ILogger` from dependency injection (DI). In non-host console apps, use the `LoggerFactory` to create an `ILogger`.

The following ASP.NET Core example creates a logger with `TodoApiSample.Pages.AboutModel` as the category. The log *category* is a string that is associated with each log. The `ILogger<T>` instance provided by DI creates logs that have the fully qualified name of type `T` as the category.

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }
```

In the following ASP.NET Core and console app examples, the logger is used to create logs with `Information` as the level. The Log *level* indicates the severity of the logged event.

```
public void OnGet()
{
    Message = $"About page visited at {DateTime.UtcNow.ToLongTimeString()}";
    _logger.LogInformation("Message displayed: {Message}", Message);
}
```

Levels and categories are explained in more detail later in this article.

Create logs in Startup

To write logs in the `Startup` class, include an `ILogger` parameter in the constructor signature:

```
public class Startup
{
    private readonly ILogger _logger;

    public Startup(IConfiguration configuration, ILogger<Startup> logger)
    {
        Configuration = configuration;
        _logger = logger;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

        // Add our repository type
        services.AddSingleton<ITodoRepository, TodoRepository>();
        _logger.LogInformation("Added TodoRepository to services");
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            _logger.LogInformation("In Development environment");
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseMvc();
    }
}
```

Create logs in the Program class

To write logs in the `Program` class, get an `ILogger` instance from DI:

```
public static void Main(string[] args)
{
    var host = CreateWebHostBuilder(args).Build();

    var todoRepository = host.Services.GetRequiredService<ITodoRepository>();
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Feed the dog" });
    todoRepository.Add(new Core.Model.TodoItem() { Name = "Walk the dog" });

    var logger = host.Services.GetRequiredService<ILogger<Program>>();
    logger.LogInformation("Seeded the database.");

    host.Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        });
```

Logging during host construction isn't directly supported. However, a separate logger can be used. In the following example, a Serilog logger is used to log in `CreateWebHostBuilder`. `AddSerilog` uses the static configuration specified in `Log.Logger`:

```
using System;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args)
    {
        var builtConfig = new ConfigurationBuilder()
            .AddJsonFile("appsettings.json")
            .AddCommandLine(args)
            .Build();

        Log.Logger = new LoggerConfiguration()
            .WriteTo.Console()
            .WriteTo.File(builtConfig["Logging:FilePath"])
            .CreateLogger();

        try
        {
            return WebHost.CreateDefaultBuilder(args)
                .ConfigureServices((context, services) =>
                {
                    services.AddMvc();
                })
                .ConfigureAppConfiguration((hostingContext, config) =>
                {
                    config.AddConfiguration(builtConfig);
                })
                .ConfigureLogging(logging =>
                {
                    logging.AddSerilog();
                })
                .UseStartup<Startup>();
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Host builder error");

            throw;
        }
        finally
        {
            Log.CloseAndFlush();
        }
    }
}
```

No asynchronous logger methods

Logging should be so fast that it isn't worth the performance cost of asynchronous code. If your logging data store is slow, don't write to it directly. Consider writing the log messages to a fast store initially, then move them to the slow store later. For example, if you're logging to SQL Server, you don't want to do that directly in a `Log` method, since the `Log` methods are synchronous. Instead, synchronously add log messages to an in-memory queue and have a background worker pull the messages out of the queue to do the asynchronous work of pushing data to SQL Server. For more information, see this GitHub issue.

Configuration

Logging provider configuration is provided by one or more configuration providers:

- File formats (INI, JSON, and XML).
- Command-line arguments.
- Environment variables.
- In-memory .NET objects.
- The unencrypted Secret Manager storage.
- An encrypted user store, such as Azure Key Vault.
- Custom providers (installed or created).

For example, logging configuration is commonly provided by the `Logging` section of app settings files. The following example shows the contents of a typical *appsettings.Development.json* file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    },
    "Console":
    {
      "IncludeScopes": true
    }
  }
}
```

The `Logging` property can have `LogLevel` and log provider properties (Console is shown).

The `LogLevel` property under `Logging` specifies the minimum level to log for selected categories. In the example, `System` and `Microsoft` categories log at `Information` level, and all others log at `Debug` level.

Other properties under `Logging` specify logging providers. The example is for the Console provider. If a provider supports log scopes, `IncludeScopes` indicates whether they're enabled. A provider property (such as `Console` in the example) may also specify a `LogLevel` property. `LogLevel` under a provider specifies levels to log for that provider.

If levels are specified in `Logging.{providername}.LogLevel`, they override anything set in `Logging.LogLevel`. For example, consider the following JSON:

```
{
  "Logging": {      // Default, all providers.
    "LogLevel": {
      "Microsoft": "Warning"
    },
    "Console": { // Console provider.
      "LogLevel": {
        "Microsoft": "Information"
      }
    }
  }
}
```

In the preceding JSON, the `Console` provider settings overrides the preceding (default) log level.

The Logging API doesn't include a scenario to change log levels while an app is running. However, some configuration providers are capable of reloading configuration, which takes immediate effect on logging configuration. For example, the File Configuration Provider, which is added by `CreateDefaultBuilder` to read settings files, reloads logging configuration by default. If configuration is changed in code while an app is running, the app can call IConfigurationRoot.Reload to update the app's logging configuration.

For information on implementing configuration providers, see Configuration in ASP.NET Core.

Sample logging output

With the sample code shown in the preceding section, logs appear in the console when the app is run from the command line. Here's an example of console output:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/api/todo/0
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments
(0) - ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 42.9286ms
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 148.889ms 404
```

The preceding logs were generated by making an HTTP Get request to the sample app at `http://localhost:5000/api/todo/0`.

Here's an example of the same logs as they appear in the Debug window when you run the sample app in Visual Studio:

```
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request starting HTTP/1.1 GET
http://localhost:53104/api/todo/0
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executing action method
TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments (0) - ModelState is Valid
TodoApi.Controllers.TodoController:Information: Getting item 0
TodoApi.Controllers.TodoController:Warning: GetById(0) NOT FOUND
Microsoft.AspNetCore.Mvc.StatusCodeResult:Information: Executing HttpStatusCodeResult, setting
HTTP status code 404
Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker:Information: Executed action
TodoApi.Controllers.TodoController.GetById (TodoApi) in 152.5657ms
Microsoft.AspNetCore.Hosting.Internal.WebHost:Information: Request finished in 316.3195ms 404
```

The logs that are created by the `ILogger` calls shown in the preceding section begin with "TodoApi". The logs that begin with "Microsoft" categories are from ASP.NET Core framework code. ASP.NET Core and application code are using the same logging API and providers.

The remainder of this article explains some details and options for logging.

NuGet packages

The `ILogger` and `ILoggerFactory` interfaces are in Microsoft.Extensions.Logging.Abstractions, and default implementations for them are in Microsoft.Extensions.Logging.

Log category

When an `ILogger` object is created, a *category* is specified for it. That category is included with each log message created by that instance of `ILogger`. The category may be any string, but the convention is to use the class name, such as "TodoApi.Controllers.TodoController".

Use `ILogger<T>` to get an `ILogger` instance that uses the fully qualified type name of `T` as the category:

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILogger<TodoController> logger)
    {
        _todoRepository = todoRepository;
        _logger = logger;
    }
```

To explicitly specify the category, call `ILoggerFactory.CreateLogger`:

```
public class TodoController : Controller
{
    private readonly ITodoRepository _todoRepository;
    private readonly ILogger _logger;

    public TodoController(ITodoRepository todoRepository,
        ILoggerFactory logger)
    {
        _todoRepository = todoRepository;
        _logger = logger.CreateLogger("TodoApiSample.Controllers.TodoController");
    }
```

`ILogger<T>` is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

Every log specifies a LogLevel value. The log level indicates the severity or importance. For example, you might write an `Information` log when a method ends normally and a `Warning` log when a method returns a *404 Not Found* status code.

The following code creates `Information` and `Warning` logs:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

In the preceding code, the `MyLogEvents.GetItem` and `MyLogEvents.GetItemNotFound` parameters are the Log event ID. The second parameter is a message template with placeholders for argument values

provided by the remaining method parameters. The method parameters are explained in the Log message template section in this article.

Log methods that include the level in the method name (for example, `LogInformation` and `LogWarning`) are extension methods for ILogger. These methods call a `Log` method that takes a `LogLevel` parameter. You can call the `Log` method directly rather than one of these extension methods, but the syntax is relatively complicated. For more information, see ILogger and the logger extensions source code.

ASP.NET Core defines the following log levels, ordered here from lowest to highest severity.

- Trace = 0

  For information that's typically valuable only for debugging. These messages may contain sensitive application data and so shouldn't be enabled in a production environment. *Disabled by default.*

- Debug = 1

  For information that may be useful in development and debugging. Example: `Entering method Configure with flag set to true.` Enable `Debug` level logs in production only when troubleshooting, due to the high volume of logs.

- Information = 2

  For tracking the general flow of the app. These logs typically have some long-term value. Example: `Request received for path /api/todo`

- Warning = 3

  For abnormal or unexpected events in the app flow. These may include errors or other conditions that don't cause the app to stop but might need to be investigated. Handled exceptions are a common place to use the `Warning` log level. Example: `FileNotFoundException for file quotes.txt.`

- Error = 4

  For errors and exceptions that cannot be handled. These messages indicate a failure in the current activity or operation (such as the current HTTP request), not an app-wide failure. Example log message: `Cannot insert record due to duplicate key violation.`

- Critical = 5

  For failures that require immediate attention. Examples: data loss scenarios, out of disk space.

Use the log level to control how much log output is written to a particular storage medium or display window. For example:

- In production:
  - Logging at the `Trace` through `Information` levels produces a high-volume of detailed log messages. To control costs and not exceed data storage limits, log `Trace` through `Information` level messages to a high-volume, low-cost data store.
  - Logging at `Warning` through `Critical` levels typically produces fewer, smaller log messages. Therefore, costs and storage limits usually aren't a concern, which results in greater flexibility of data store choice.
- During development:
  - Log `Warning` through `Critical` messages to the console.
  - Add `Trace` through `Information` messages when troubleshooting.

The Log filtering section later in this article explains how to control which log levels a provider handles.

ASP.NET Core writes logs for framework events. The log examples earlier in this article excluded logs below `Information` level, so no `Debug` or `Trace` level logs were created. Here's an example of console logs produced by running the sample app configured to show `Debug` logs:

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:62555/api/todo/0
dbug: Microsoft.AspNetCore.Routing.Tree.TreeRouter[1]
      Request successfully matched the route with name 'GetTodo' and template 'api/Todo/{id}'.
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Update (TodoApi)' with id '089d59b6-92ec-472d-
b552-cc613dfd625d' did not match the constraint
'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ActionSelector[2]
      Action 'TodoApi.Controllers.TodoController.Delete (TodoApi)' with id 'f3476abe-4bd9-4ad3-
9261-3ead09607366' did not match the constraint
'Microsoft.AspNetCore.Mvc.Internal.HttpMethodActionConstraint'
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action TodoApi.Controllers.TodoController.GetById (TodoApi)
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method TodoApi.Controllers.TodoController.GetById (TodoApi) with arguments
(0) - ModelState is Valid
info: TodoApi.Controllers.TodoController[1002]
      Getting item 0
warn: TodoApi.Controllers.TodoController[4000]
      GetById(0) NOT FOUND
dbug: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action method TodoApi.Controllers.TodoController.GetById (TodoApi), returned result
Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
      Executing HttpStatusCodeResult, setting HTTP status code 404
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[2]
      Executed action TodoApi.Controllers.TodoController.GetById (TodoApi) in 0.8788ms
dbug: Microsoft.AspNetCore.Server.Kestrel[9]
      Connection id "0HL6L7NEFF2QD" completed keep alive response.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]
      Request finished in 2.7286ms 404
```

Log event ID

Each log can specify an *event ID*. The sample app does this by using a locally defined `LoggingEvents` class:

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

```
public class LoggingEvents
{
    public const int GenerateItems = 1000;
    public const int ListItems = 1001;
    public const int GetItem = 1002;
    public const int InsertItem = 1003;
    public const int UpdateItem = 1004;
    public const int DeleteItem = 1005;

    public const int GetItemNotFound = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

An event ID associates a set of events. For example, all logs related to displaying a list of items on a page might be 1001.

The logging provider may store the event ID in an ID field, in the logging message, or not at all. The Debug provider doesn't show event IDs. The console provider shows event IDs in brackets after the category:

```
info: TodoApi.Controllers.TodoController[1002]
      Getting item invalidid
warn: TodoApi.Controllers.TodoController[4000]
      GetById(invalidid) NOT FOUND
```

Log message template

Each log specifies a message template. The message template can contain placeholders for which arguments are provided. Use names for the placeholders, not numbers.

```
public IActionResult GetById(string id)
{
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
    var item = _todoRepository.Find(id);
    if (item == null)
    {
        _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
        return NotFound();
    }
    return new ObjectResult(item);
}
```

The order of placeholders, not their names, determines which parameters are used to provide their values. In the following code, notice that the parameter names are out of sequence in the message template:

```
string p1 = "parm1";
string p2 = "parm2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

This code creates a log message with the parameter values in sequence:

```
Parameter values: parm1, parm2
```

The logging framework works this way so that logging providers can implement semantic logging, also known as structured logging. The arguments themselves are passed to the logging system, not

just the formatted message template. This information enables logging providers to store the parameter values as fields. For example, suppose logger method calls look like this:

```
_logger.LogInformation("Getting item {Id} at {RequestTime}", id, DateTime.Now);
```

If you're sending the logs to Azure Table Storage, each Azure Table entity can have `ID` and `RequestTime` properties, which simplifies queries on log data. A query can find all logs within a particular `RequestTime` range without parsing the time out of the text message.

Logging exceptions

The logger methods have overloads that let you pass in an exception, as in the following example:

```
catch (Exception ex)
{
    _logger.LogWarning(LoggingEvents.GetItemNotFound, ex, "GetById({Id}) NOT FOUND", id);
    return NotFound();
}
return new ObjectResult(item);
```

Different providers handle the exception information in different ways. Here's an example of Debug provider output from the code shown above.

```
TodoApiSample.Controllers.TodoController: Warning: GetById(55) NOT FOUND

System.Exception: Item not found exception.
   at TodoApiSample.Controllers.TodoController.GetById(String id) in
C:\TodoApiSample\Controllers\TodoController.cs:line 226
```

Log filtering

You can specify a minimum log level for a specific provider and category or for all providers or all categories. Any logs below the minimum level aren't passed to that provider, so they don't get displayed or stored.

To suppress all logs, specify `LogLevel.None` as the minimum log level. The integer value of `LogLevel.None` is 6, which is higher than `LogLevel.Critical` (5).

Create filter rules in configuration

The project template code calls `CreateDefaultBuilder` to set up logging for the Console, Debug, and EventSource (ASP.NET Core 2.2 or later) providers. The `CreateDefaultBuilder` method sets up logging to look for configuration in a `Logging` section, as explained earlier in this article.

The configuration data specifies minimum log levels by provider and category, as in the following example:

```
{
  "Logging": {
    "Debug": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "IncludeScopes": false,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

This JSON creates six filter rules: one for the Debug provider, four for the Console provider, and one for all providers. A single rule is chosen for each provider when an `ILogger` object is created.

Filter rules in code

The following example shows how to register filter rules in code:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging =>
        logging.AddFilter("System", LogLevel.Debug)
               .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Trace));
```

The second `AddFilter` specifies the Debug provider by using its type name. The first `AddFilter` applies to all providers because it doesn't specify a provider type.

How filtering rules are applied

The configuration data and the `AddFilter` code shown in the preceding examples create the rules shown in the following table. The first six come from the configuration example and the last two come from the code example.

NUMBER	PROVIDER	CATEGORIES THAT BEGIN WITH ...	MINIMUM LOG LEVEL
1	Debug	All categories	Information
2	Console	Microsoft.AspNetCore.Mvc.Razor.Internal	Warning
3	Console	Microsoft.AspNetCore.Mvc.Razor.Razor	Debug
4	Console	Microsoft.AspNetCore.Mvc.Razor	Error
5	Console	All categories	Information

NUMBER	PROVIDER	CATEGORIES THAT BEGIN WITH ...	MINIMUM LOG LEVEL
6	All providers	All categories	Debug
7	All providers	System	Debug
8	Debug	Microsoft	Trace

When an `ILogger` object is created, the `ILoggerFactory` object selects a single rule per provider to apply to that logger. All messages written by an `ILogger` instance are filtered based on the selected rules. The most specific rule possible for each provider and category pair is selected from the available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If no match is found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If no match is found, select all rules that don't specify a category.
- If multiple rules are selected, take the **last** one.
- If no rules are selected, use `MinimumLevel`.

With the preceding list of rules, suppose you create an `ILogger` object for category "Microsoft.AspNetCore.Mvc.Razor.RazorViewEngine":

- For the Debug provider, rules 1, 6, and 8 apply. Rule 8 is most specific, so that's the one selected.
- For the Console provider, rules 3, 4, 5, and 6 apply. Rule 3 is most specific.

The resulting `ILogger` instance sends logs of `Trace` level and above to the Debug provider. Logs of `Debug` level and above are sent to the Console provider.

Provider aliases

Each provider defines an *alias* that can be used in configuration in place of the fully qualified type name. For the built-in providers, use the following aliases:

- Console
- Debug
- EventSource
- EventLog
- TraceSource
- AzureAppServicesFile
- AzureAppServicesBlob
- ApplicationInsights

Default minimum level

There's a minimum level setting that takes effect only if no rules from configuration or code apply for a given provider and category. The following example shows how to set the minimum level:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning));
```

If you don't explicitly set the minimum level, the default value is `Information`, which means that `Trace` and `Debug` logs are ignored.

Filter functions

A filter function is invoked for all providers and categories that don't have rules assigned to them by configuration or code. Code in the function has access to the provider type, category, and log level. For example:

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging(logBuilder =>
    {
        logBuilder.AddFilter((provider, category, logLevel) =>
        {
            if (provider == "Microsoft.Extensions.Logging.Console.ConsoleLoggerProvider" &&
                category == "TodoApiSample.Controllers.TodoController")
            {
                return false;
            }
            return true;
        });
    });
```

System categories and levels

Here are some categories used by ASP.NET Core and Entity Framework Core, with notes about what logs to expect from them:

CATEGORY	NOTES
Microsoft.AspNetCore	General ASP.NET Core diagnostics.
Microsoft.AspNetCore.DataProtection	Which keys were considered, found, and used.
Microsoft.AspNetCore.HostFiltering	Hosts allowed.
Microsoft.AspNetCore.Hosting	How long HTTP requests took to complete and what time they started. Which hosting startup assemblies were loaded.
Microsoft.AspNetCore.Mvc	MVC and Razor diagnostics. Model binding, filter execution, view compilation, action selection.
Microsoft.AspNetCore.Routing	Route matching information.
Microsoft.AspNetCore.Server	Connection start, stop, and keep alive responses. HTTPS certificate information.
Microsoft.AspNetCore.StaticFiles	Files served.
Microsoft.EntityFrameworkCore	General Entity Framework Core diagnostics. Database activity and configuration, change detection, migrations.

Log scopes

A *scope* can group a set of logical operations. This grouping can be used to attach the same data to each log that's created as part of a set. For example, every log created as part of processing a transaction can include the transaction ID.

A scope is an `IDisposable` type that's returned by the BeginScope method and lasts until it's disposed. Use a scope by wrapping logger calls in a `using` block:

```
public IActionResult GetById(string id)
{
    TodoItem item;
    using (_logger.BeginScope("Message attached to logs created in the using block"))
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);
        item = _todoRepository.Find(id);
        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
            return NotFound();
        }
    }
    return new ObjectResult(item);
}
```

The following code enables scopes for the console provider:

Program.cs:

```
.ConfigureLogging((hostingContext, logging) =>
{
    logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
    logging.AddConsole(options => options.IncludeScopes = true);
    logging.AddDebug();
})
```

> **NOTE**
>
> Configuring the `IncludeScopes` console logger option is required to enable scope-based logging.
>
> For information on configuration, see the Configuration section.

Each log message includes the scoped information:

```
info: TodoApiSample.Controllers.TodoController[1002]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 =>
TodoApiSample.Controllers.TodoController.GetById (TodoApi) => Message attached to logs created in
the using block
      Getting item 0
warn: TodoApiSample.Controllers.TodoController[4000]
      => RequestId:0HKV9C49II9CK RequestPath:/api/todo/0 =>
TodoApiSample.Controllers.TodoController.GetById (TodoApi) => Message attached to logs created in
the using block
      GetById(0) NOT FOUND
```

Built-in logging providers

ASP.NET Core ships the following providers:

- Console

- Debug
- EventSource
- EventLog
- TraceSource
- AzureAppServicesFile
- AzureAppServicesBlob
- ApplicationInsights

For information on stdout and debug logging with the ASP.NET Core Module, see Troubleshoot ASP.NET Core on Azure App Service and IIS and ASP.NET Core Module.

Console provider

The Microsoft.Extensions.Logging.Console provider package sends log output to the console.

```
logging.AddConsole();
```

To see console logging output, open a command prompt in the project folder and run the following command:

```
dotnet run
```

Debug provider

The Microsoft.Extensions.Logging.Debug provider package writes log output by using the System.Diagnostics.Debug class (`Debug.WriteLine` method calls).

On Linux, this provider writes logs to */var/log/message*.

```
logging.AddDebug();
```

Event Source provider

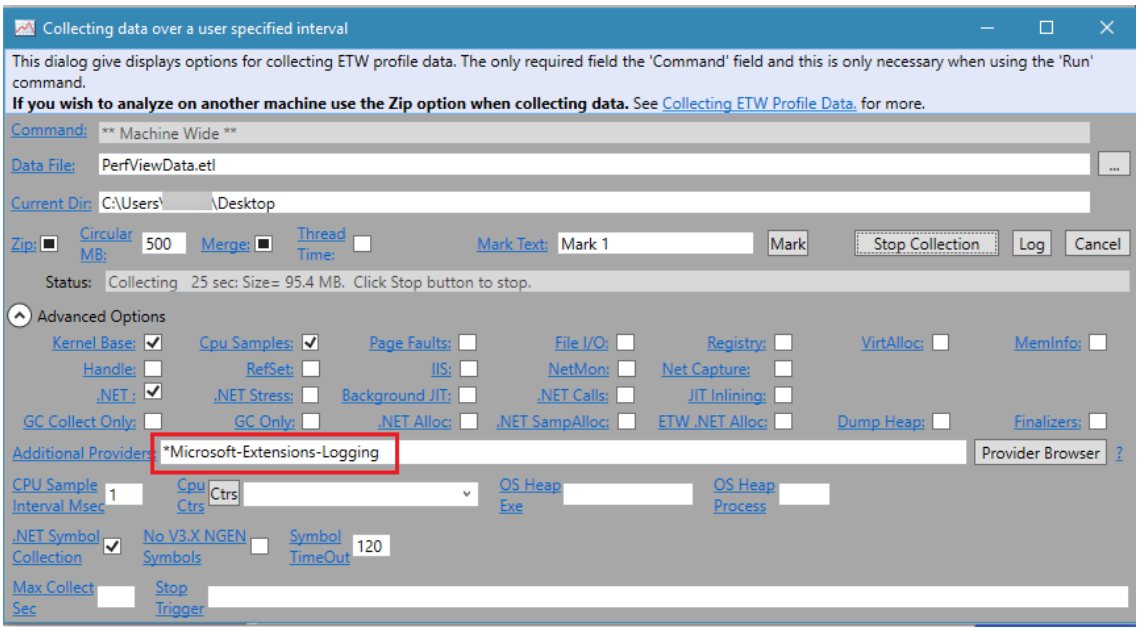
The Microsoft.Extensions.Logging.EventSource provider package writes to an Event Source cross-platform with the name `Microsoft-Extensions-Logging`. On Windows, the provider uses ETW.

```
logging.AddEventSourceLogger();
```

The Event Source provider is added automatically when `CreateDefaultBuilder` is called to build the host.

Use the PerfView utility to collect and view logs. There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET Core.

To configure PerfView for collecting events logged by this provider, add the string `*Microsoft-Extensions-Logging` to the **Additional Providers** list. (Don't miss the asterisk at the start of the string.)



Windows EventLog provider

The Microsoft.Extensions.Logging.EventLog provider package sends log output to the Windows Event Log.

```
logging.AddEventLog();
```

AddEventLog overloads let you pass in EventLogSettings. If `null` or not specified, the following default settings are used:

- `LogName`: "Application"
- `SourceName`: ".NET Runtime"
- `MachineName`: The local machine name is used.

Events are logged for Warning level and higher. The following example sets the Event Log default log level to LogLevel.Information:

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

TraceSource provider

The Microsoft.Extensions.Logging.TraceSource provider package uses the TraceSource libraries and providers.

```
logging.AddTraceSource(sourceSwitchName);
```

AddTraceSource overloads let you pass in a source switch and a trace listener.

To use this provider, an app has to run on the .NET Framework (rather than .NET Core). The provider can route messages to a variety of listeners, such as the TextWriterTraceListener used in the sample app.

Azure App Service provider

The Microsoft.Extensions.Logging.AzureAppServices provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account.

```
logging.AddAzureWebAppDiagnostics();
```

The provider package isn't included in the Microsoft.AspNetCore.App metapackage. When targeting .NET Framework or referencing the `Microsoft.AspNetCore.App` metapackage, add the provider package to the project.

An AddAzureWebAppDiagnostics overload lets you pass in AzureAppServicesDiagnosticsSettings. The settings object can override default settings, such as the logging output template, blob name, and file size limit. (*Output template* is a message template that's applied to all logs in addition to what's provided with an `ILogger` method call.)

When you deploy to an App Service app, the application honors the settings in the App Service logs section of the **App Service** page of the Azure portal. When the following settings are updated, the changes take effect immediately without requiring a restart or redeployment of the app.

- **Application Logging (Filesystem)**
- **Application Logging (Blob)**

The default location for log files is in the *D:\home\LogFiles\Application* folder, and the default file name is *diagnostics-yyyymmdd.txt*. The default file size limit is 10 MB, and the default maximum number of files retained is 2. The default blob name is *{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt*.

The provider only works when the project runs in the Azure environment. It has no effect when the project is run locally—it doesn't write to local files or local development storage for blobs.

Azure log streaming

Azure log streaming lets you view log activity in real time from:

- The app server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the **App Service logs** page from your app's portal page.
- Set **Application Logging (Filesystem)** to **On**.
- Choose the log **Level**. This setting only applies to Azure log streaming, not other logging providers in the app.

Navigate to the **Log Stream** page to view app messages. They're logged by the app through the `ILogger` interface.

Azure Application Insights trace logging

The Microsoft.Extensions.Logging.ApplicationInsights provider package writes logs to Azure Application Insights. Application Insights is a service that monitors a web app and provides tools for querying and analyzing the telemetry data. If you use this provider, you can query and analyze your logs by using the Application Insights tools.

The provider package isn't included in the shared framework. To use the provider, add the provider package to the project. The logging provider is included as a dependency of Microsoft.ApplicationInsights.AspNetCore, which is the package that provides all available telemetry for ASP.NET Core. If you use this package, you don't have to install the provider package.

Don't use the Microsoft.ApplicationInsights.Web package—that's for ASP.NET 4.x.

For more information, see the following resources:

- Application Insights overview
- Application Insights for ASP.NET Core applications - Start here if you want to implement the full range of Application Insights telemetry along with logging.
- ApplicationInsightsLoggerProvider for .NET Core ILogger logs - Start here if you want to implement the logging provider without the rest of Application Insights telemetry.
- Application Insights logging adapters.
- Install, configure, and initialize the Application Insights SDK - Interactive tutorial on the Microsoft Learn site.

Third-party logging providers

Third-party logging frameworks that work with ASP.NET Core:

- elmah.io (GitHub repo)
- Gelf (GitHub repo)
- JSNLog (GitHub repo)
- KissLog.net (GitHub repo)
- Log4Net (GitHub repo)
- Loggr (GitHub repo)
- NLog (GitHub repo)
- Sentry (GitHub repo)
- Serilog (GitHub repo)
- Stackdriver (Github repo)

Some third-party frameworks can perform semantic logging, also known as structured logging.

Using a third-party framework is similar to using one of the built-in providers:

1. Add a NuGet package to your project.
2. Call an `ILoggerFactory` or `ILoggingBuilder` extension method provided by the logging framework.

For more information, see each provider's documentation. Third-party logging providers aren't supported by Microsoft.

Additional resources

- High-performance logging with LoggerMessage in ASP.NET Core

Troubleshoot ASP.NET Core on Azure App Service and IIS

9/22/2020 • 59 minutes to read • Edit Online

By Justin Kotalik

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.

- When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error,

either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file deployment model. Use **one** of the following approaches in the app's project file to fix

this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`
- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.
4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:
   - Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic

logging. Redeploy the app.

- Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
  a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
  b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
  c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.

2. Run the EnableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.

2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.

3. Restore and rebuild the project.

4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
   - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.
3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module fails to find the .NET Core CLR and find the in-process request handler (*aspnetcorev2_inprocess.dll*). Check that:

- The app targets either the Microsoft.AspNetCore.Server.IIS NuGet package or the Microsoft.AspNetCore.App metapackage.
- The version of the ASP.NET Core shared framework that the app targets is installed on the target machine.

500.0 Out-Of-Process Handler Load Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module fails to find the out-of-process hosting request handler. Make sure the *aspnetcorev2_outofprocess.dll* is present in a subfolder next to *aspnetcorev2.dll*.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout

log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.

2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Under **SELECT PROBLEM CATEGORY**, select the **Web App Down** button.
3. Under **Suggested Solutions** > **Enable Stdout Log Redirection**, select the button to **Open Kudu Console to edit Web.Config**.
4. In the Kudu **Diagnostic Console**, open the folders to the path **site** > **wwwroot**. Scroll down to reveal the *web.config* file at the bottom of the list.
5. Click the pencil icon next to the *web.config* file.
6. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
7. Select **Save** to save the updated *web.config* file.
8. Make a request to the app.
9. Return to the Azure portal. Select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
10. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
11. Select the **LogFiles** folder.
12. Inspect the **Modified** column and select the pencil icon to edit the stdout log with the latest modification date.
13. When the log file opens, the error is displayed.

Disable stdout logging when troubleshooting is complete:

1. In the Kudu **Diagnostic Console**, return to the path **site** > **wwwroot** to reveal the *web.config* file. Open the **web.config** file again by selecting the pencil icon.
2. Set **stdoutLogEnabled** to `false`.
3. Select **Save** to save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:
   - Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
   - Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
     a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
     b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
     c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.
2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.

4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.

2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.
2. Run the EnableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 Process Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
   - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.
3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.

2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`
- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Under **SELECT PROBLEM CATEGORY**, select the **Web App Down** button.
3. Under **Suggested Solutions** > **Enable Stdout Log Redirection**, select the button to **Open Kudu Console to edit Web.Config**.
4. In the Kudu **Diagnostic Console**, open the folders to the path **site** > **wwwroot**. Scroll down to reveal the *web.config* file at the bottom of the list.
5. Click the pencil icon next to the *web.config* file.
6. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
7. Select **Save** to save the updated *web.config* file.
8. Make a request to the app.
9. Return to the Azure portal. Select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
10. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
11. Select the **LogFiles** folder.
12. Inspect the **Modified** column and select the pencil icon to edit the stdout log with the latest modification date.
13. When the log file opens, the error is displayed.

Disable stdout logging when troubleshooting is complete:

1. In the Kudu **Diagnostic Console**, return to the path **site** > **wwwroot** to reveal the *web.config* file. Open the **web.config** file again by selecting the pencil icon.
2. Set **stdoutLogEnabled** to `false`.
3. Select **Save** to save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.

2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.

3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.
2. Run the EnableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview

- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common errors reference for Azure App Service and IIS with ASP.NET Core

9/22/2020 • 22 minutes to read • Edit Online

This topic describes common errors and provides troubleshooting advice for specific errors when hosting ASP.NET Core apps on Azure Apps Service and IIS.

For general troubleshooting guidance, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Collect the following information:

- Browser behavior (status code and error message)
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS
    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.
- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely
- **ASP.NET Core Module stdout Log:** The log file is created but empty.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED --**OR**-- Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied --**OR**-- 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS

website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. **--OR--** This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Download Archives and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.
- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure --**OR**-- HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

This topic describes common errors and provides troubleshooting advice for specific errors when hosting ASP.NET Core apps on Azure Apps Service and IIS.

For general troubleshooting guidance, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Collect the following information:

- Browser behavior (status code and error message)
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS
    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.
- ASP.NET Core Module stdout and debug log entries

- Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
- IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site

extension, restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED --**OR**-- Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.

- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied --**OR**-- 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. --**OR**-- This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-

compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0.
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Download Archives and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"dotnet" .{ASSEMBLY}.dll', ErrorCode = '0x80004005 : 80008081.
- **ASP.NET Core Module stdout Log:** The application to execute does not exist: 'PATH{ASSEMBLY}.dll'

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`)

for a self-contained deployment (SCD).

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Warning: Could not create stdoutLogFile \?{PATH}\path_doesnt_exist\stdout_{PROCESS ID}_{TIMESTAMP}.log, ErrorCode = -2147024893.
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' created process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ' but either crashed or did not respond or did not listen on the given port '{PORT}', ErrorCode = '{ERROR CODE}'
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8

9/22/2020 • 50 minutes to read • Edit Online

By Tom Dykstra and Rick Anderson

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.
- Visual Studio
- Visual Studio Code
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK or later

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

The Visual Studio Code instructions use SQLite, a cross-platform database engine.

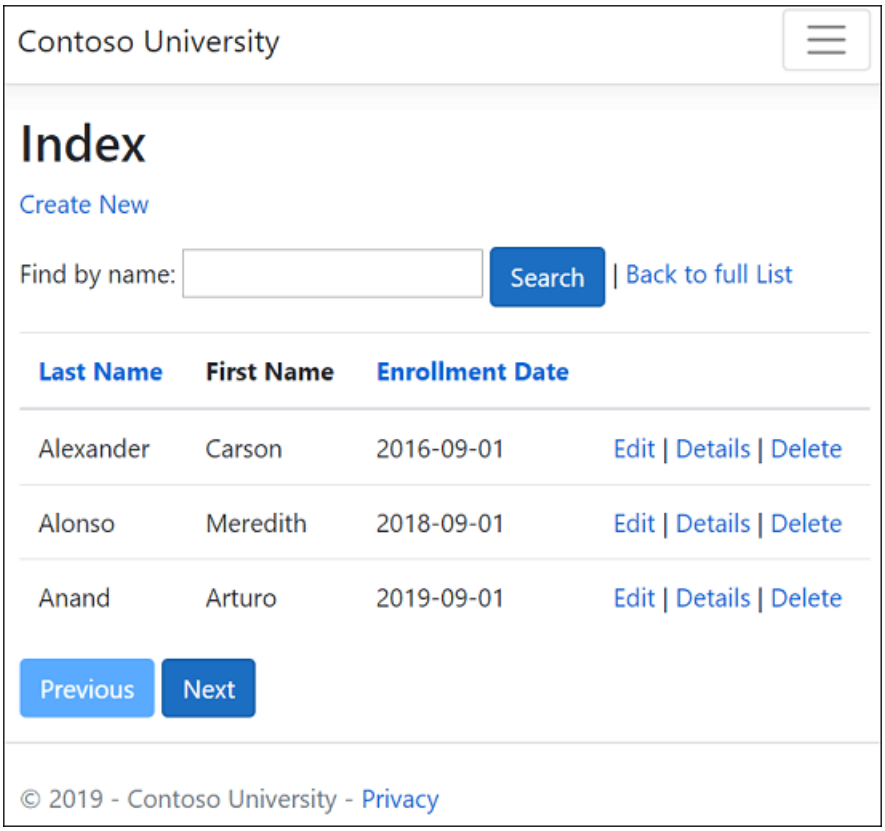
If you choose to use SQLite, download and install a third-party tool for managing and viewing a SQLite database, such as DB Browser for SQLite.

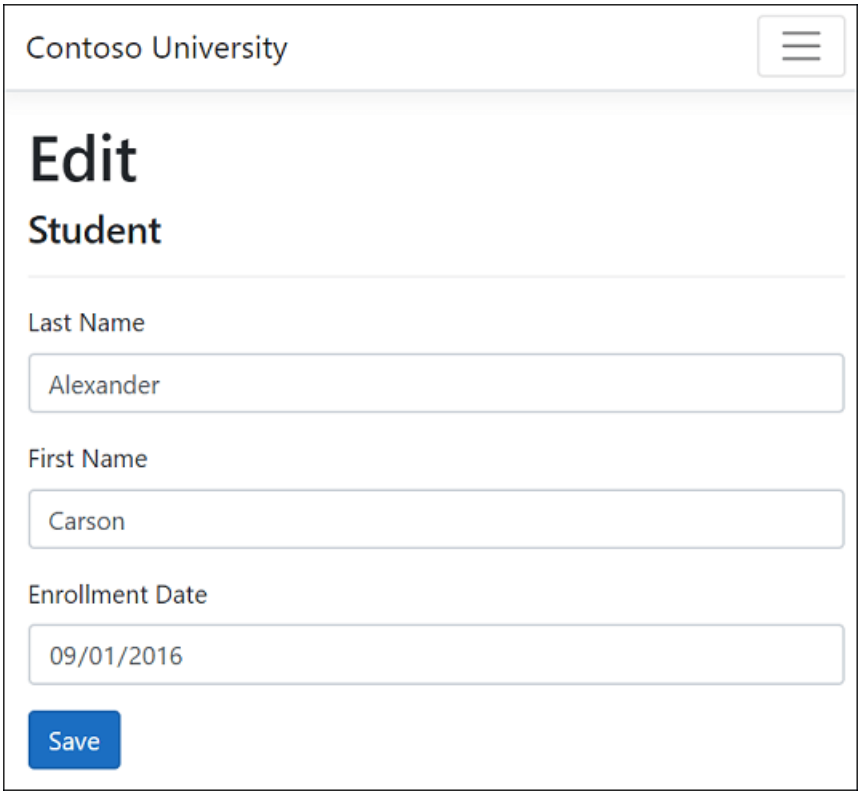
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core, not how to customize the UI.

Follow the link at the top of the page to get the source code for the completed project. The *cu30* folder has the code for the ASP.NET Core 3.0 version of the tutorial. Files that reflect the state of the code for tutorials 1-7 can be found in the *cu30snapshots* folder.

- Visual Studio
- Visual Studio Code

To run the app after downloading the completed project:

- Build the project.
- In Package Manager Console (PMC) run the following command:

```
Update-Database
```

- Run the project to seed the database.

Create the web app project

- Visual Studio
- Visual Studio Code

- From the Visual Studio **File** menu, select **New** > **Project**.
- Select **ASP.NET Core Web Application**.
- Name the project *ContosoUniversity*. It's important to use this exact name including capitalization, so the namespaces match when code is copied and pasted.
- Select **.NET Core** and **ASP.NET Core 3.0** in the dropdowns, and then select **Web Application**.

Set up the site style

Set up the site header, footer, and menu by updating *Pages/Shared/_Layout.cshtml*:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Delete the **Home** and **Privacy** menu entries, and add entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

The changes are highlighted.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom
box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET Core with text about this app:

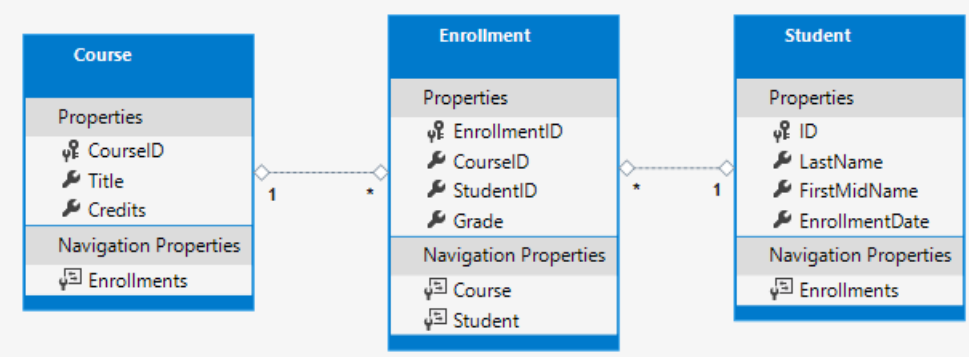
```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in a series of tutorials.
                </p>
                <p>
                    <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
                    <a href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
                </p>
            </div>
        </div>
    </div>
</div>
```

Run the app to verify that the home page appears.

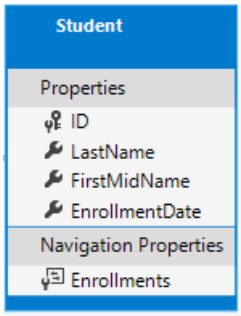
The data model

The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity



- Create a *Models* folder in the project folder.
- Create *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

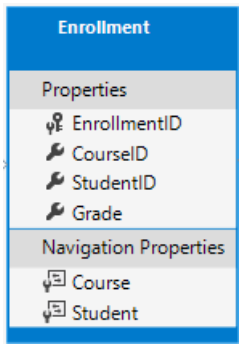
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its StudentID column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have StudentID = 1. StudentID is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. You can use other collection types, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



Create *Models/Enrollment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

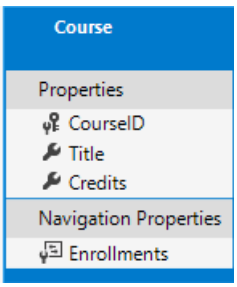
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the project to validate that there are no compiler errors.

Scaffold Student pages

In this section, you use the ASP.NET Core scaffolding tool to generate:

- An EF Core *context* class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the `Microsoft.EntityFrameworkCore.DbContext` class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

- Visual Studio
- Visual Studio Code

- Create a *Students* folder in the *Pages* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the **+** (plus) sign.
  - Change the data context name from *ContosoUniversity.Models.ContosoUniversityContext* to *ContosoUniversity.Data.SchoolContext*.
  - Select **Add**.

The following packages are automatically installed:

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.Extensions.Logging.Debug`
- `Microsoft.EntityFrameworkCore.Tools`

If you have a problem with the preceding step, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - *Create.cshtml* and *Create.cshtml.cs*
  - *Delete.cshtml* and *Delete.cshtml.cs*
  - *Details.cshtml* and *Details.cshtml.cs*
  - *Edit.cshtml* and *Edit.cshtml.cs*
  - *Index.cshtml* and *Index.cshtml.cs*
- Creates *Data/SchoolContext.cs*.
- Adds the context to dependency injection in *Startup.cs*.
- Adds a database connection string to *appsettings.json*.

Database connection string

- Visual Studio
- Visual Studio Code

The connection string specifies SQL Server LocalDB.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext6;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update *SchoolContext.cs* with the following code:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext (DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

The highlighted code creates a DbSet<TEntity> property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

Since an entity set contains multiple entities, the DBSet properties should be plural names. Since the scaffolding tool created a `Student` DBSet, this step changes it to plural `Students`.

To make the Razor Pages code match the new DBSet name, make a global change across the whole project of `_context.Student` to `_context.Students`. There are 8 occurrences.

Build the project to verify there are no compiler errors.

Startup.cs

ASP.NET Core is built with dependency injection. Services (such as the EF Core database context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- Visual Studio
- Visual Studio Code

- In `ConfigureServices`, the highlighted lines were added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Create the database

Update *Program.cs* to create the database if it doesn't exist:

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.

- `EnsureCreated` creates a database with the new schema.

This workflow works well early in development when the schema is rapidly evolving, as long as you don't need to preserve data. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, you delete the database that was created by `EnsureCreated` and use migrations instead. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create *Data/DbInitializer.cs* with the following code:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2019-09-01")},
                new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-
09-01")},
                new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-
01")},
                new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-
09-01")},
                new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-01")}
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var courses = new Course[]
```

```
            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
            };

            context.Courses.AddRange(courses);
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In *Program.cs*, replace the `EnsureCreated` call with a `DbInitializer.Initialize` call:

  ```
  // context.Database.EnsureCreated();
  DbInitializer.Initialize(context);
  ```

- Visual Studio
- Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

- Restart the app.
- Select the Students page to see the seeded data.

View the database

- Visual Studio
- Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.

- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name you provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task<T>` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

Next steps

NEXT TUTORIAL

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.
- Visual Studio
- Visual Studio Code
- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK or later

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

The Visual Studio Code instructions use SQLite, a cross-platform database engine.

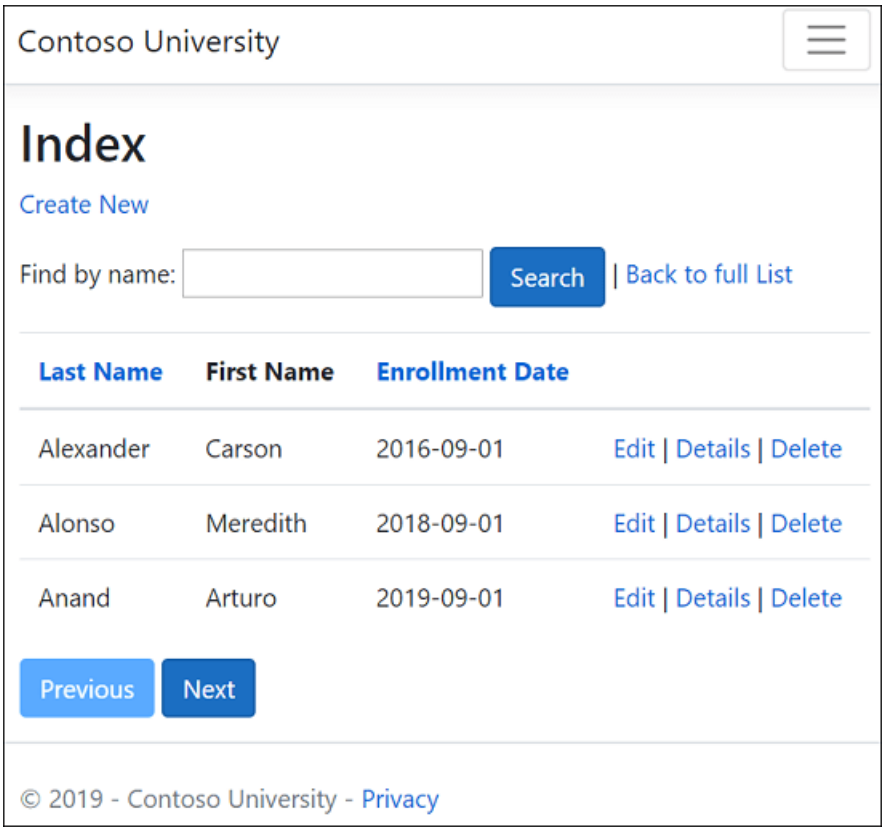
If you choose to use SQLite, download and install a third-party tool for managing and viewing a SQLite database, such as DB Browser for SQLite.

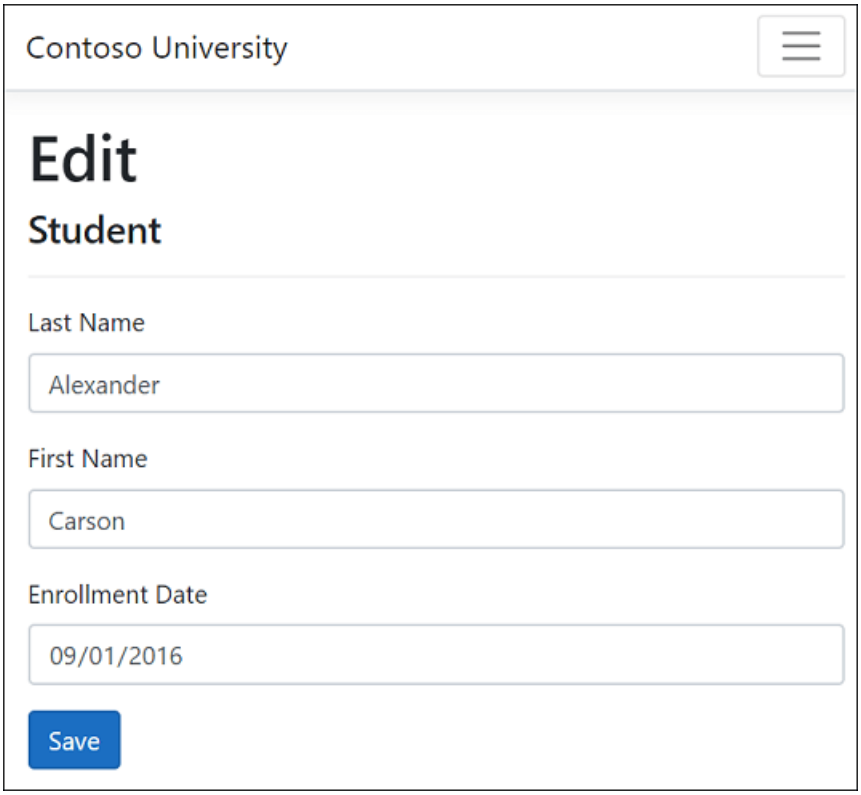
Troubleshooting

If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core, not how to customize the UI.

Follow the link at the top of the page to get the source code for the completed project. The *cu30* folder has the code for the ASP.NET Core 3.0 version of the tutorial. Files that reflect the state of the code for tutorials 1-7 can be found in the *cu30snapshots* folder.

- Visual Studio
- Visual Studio Code

To run the app after downloading the completed project:

- Build the project.
- In Package Manager Console (PMC) run the following command:

```
Update-Database
```

- Run the project to seed the database.

Create the web app project

- Visual Studio
- Visual Studio Code

- From the Visual Studio **File** menu, select **New** > **Project**.
- Select **ASP.NET Core Web Application**.
- Name the project *ContosoUniversity*. It's important to use this exact name including capitalization, so the namespaces match when code is copied and pasted.
- Select **.NET Core** and **ASP.NET Core 3.0** in the dropdowns, and then select **Web Application**.

Set up the site style

Set up the site header, footer, and menu by updating *Pages/Shared/_Layout.cshtml*:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Delete the **Home** and **Privacy** menu entries, and add entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

The changes are highlighted.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom
box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Contoso University - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET Core with text about this app:

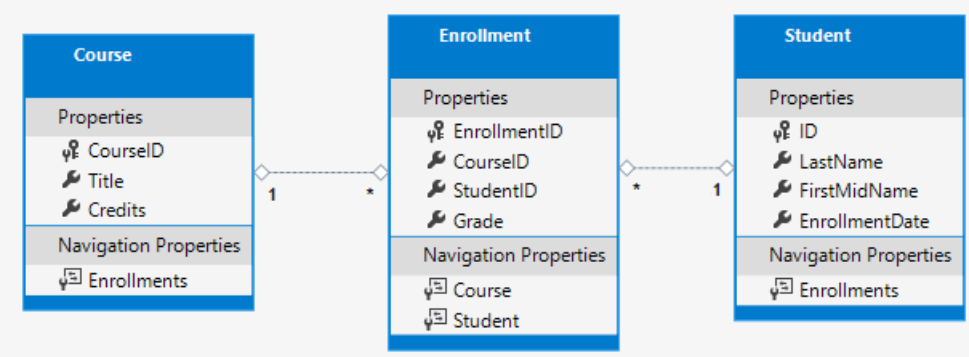
```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in a series of tutorials.
                </p>
                <p>
                    <a href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro" class="stretched-
link">See the tutorial</a>
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
                    <a href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples" class="stretched-link">See project source code</a>
                </p>
            </div>
        </div>
    </div>
</div>
```

Run the app to verify that the home page appears.

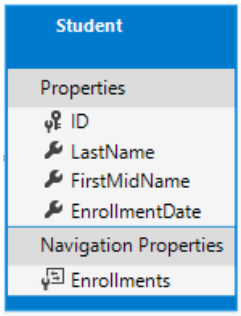
The data model

The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity



- Create a *Models* folder in the project folder.
- Create *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

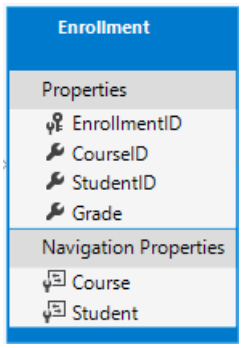
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its StudentID column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have StudentID = 1. StudentID is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. You can use other collection types, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



Create *Models/Enrollment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

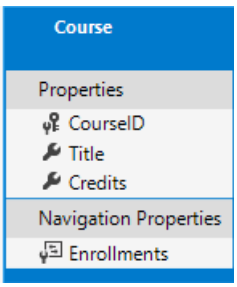
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the project to validate that there are no compiler errors.

Scaffold Student pages

In this section, you use the ASP.NET Core scaffolding tool to generate:

- An EF Core *context* class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the `Microsoft.EntityFrameworkCore.DbContext` class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

- Visual Studio
- Visual Studio Code

- Create a *Students* folder in the *Pages* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the **+** (plus) sign.
  - Change the data context name from *ContosoUniversity.Models.ContosoUniversityContext* to *ContosoUniversity.Data.SchoolContext*.
  - Select **Add**.

The following packages are automatically installed:

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`
- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.Extensions.Logging.Debug`
- `Microsoft.EntityFrameworkCore.Tools`

If you have a problem with the preceding step, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - *Create.cshtml* and *Create.cshtml.cs*
  - *Delete.cshtml* and *Delete.cshtml.cs*
  - *Details.cshtml* and *Details.cshtml.cs*
  - *Edit.cshtml* and *Edit.cshtml.cs*
  - *Index.cshtml* and *Index.cshtml.cs*
- Creates *Data/SchoolContext.cs*.
- Adds the context to dependency injection in *Startup.cs*.
- Adds a database connection string to *appsettings.json*.

Database connection string

- Visual Studio
- Visual Studio Code

The connection string specifies SQL Server LocalDB.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext6;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf* files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update *SchoolContext.cs* with the following code:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext (DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

The highlighted code creates a DbSet<TEntity> property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

Since an entity set contains multiple entities, the DBSet properties should be plural names. Since the scaffolding tool created a `Student` DBSet, this step changes it to plural `Students`.

To make the Razor Pages code match the new DBSet name, make a global change across the whole project of `_context.Student` to `_context.Students`. There are 8 occurrences.

Build the project to verify there are no compiler errors.

Startup.cs

ASP.NET Core is built with dependency injection. Services (such as the EF Core database context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

- Visual Studio
- Visual Studio Code

- In `ConfigureServices`, the highlighted lines were added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddDbContext<SchoolContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Create the database

Update *Program.cs* to create the database if it doesn't exist:

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.

- `EnsureCreated` creates a database with the new schema.

This workflow works well early in development when the schema is rapidly evolving, as long as you don't need to preserve data. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, you delete the database that was created by `EnsureCreated` and use migrations instead. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create *Data/DbInitializer.cs* with the following code:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2019-09-01")},
                new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2018-
09-01")},
                new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2017-09-01")},
                new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017-09-
01")},
                new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2016-09-01")},
                new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2018-
09-01")},
                new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2019-09-01")}
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var courses = new Course[]
```

```
            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
            };

            context.Courses.AddRange(courses);
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In *Program.cs*, replace the `EnsureCreated` call with a `DbInitializer.Initialize` call:

  ```
  // context.Database.EnsureCreated();
  DbInitializer.Initialize(context);
  ```

- Visual Studio
- Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

- Restart the app.
- Select the Students page to see the seeded data.

View the database

- Visual Studio
- Visual Studio Code

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.

- In SSOX, select **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**. The database name is generated from the context name you provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task<T>` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

Next steps

NEXT TUTORIAL

The Contoso University sample web app demonstrates how to create an ASP.NET Core Razor Pages app using Entity Framework (EF) Core.

The sample app is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This page is the first in a series of tutorials that explain how to build the Contoso University sample app.

Download or view the completed app. Download instructions.

Prerequisites

- Visual Studio
- Visual Studio Code

Visual Studio 2019 with the following workloads:

- **ASP.NET and web development**
- **.NET Core cross-platform development**

.NET Core 2.1 SDK or later

Familiarity with Razor Pages. New programmers should complete Get started with Razor Pages before starting this series.

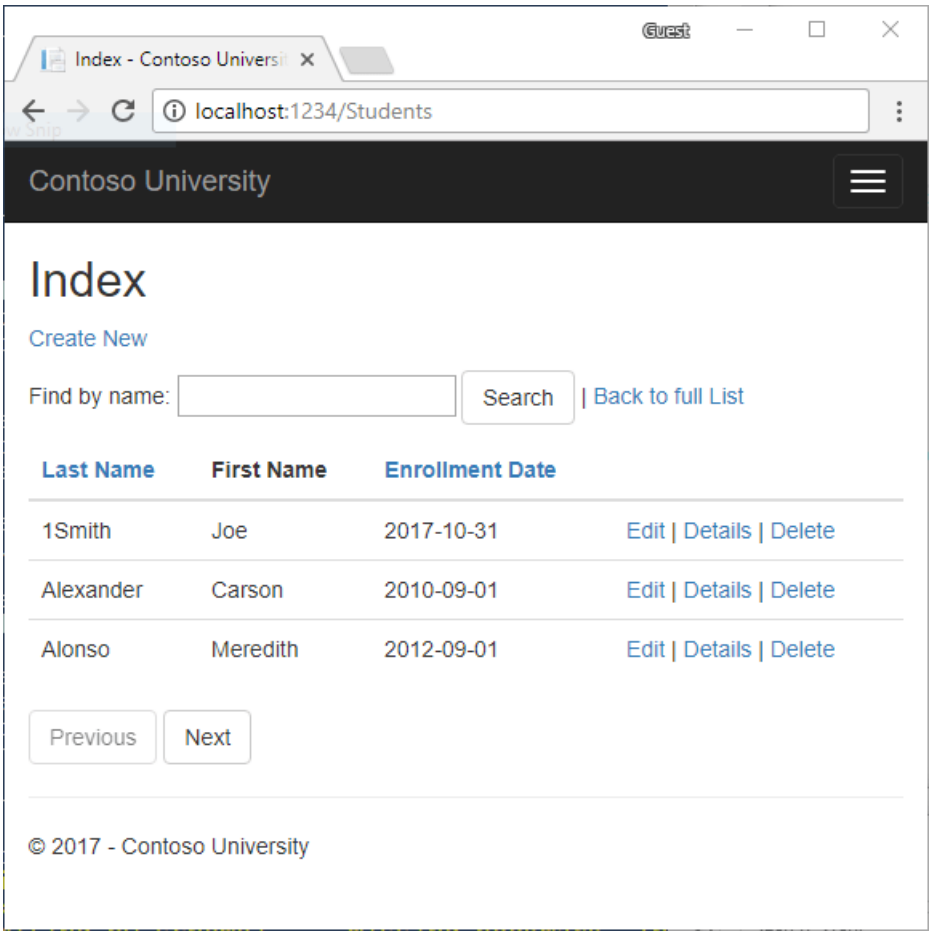
Troubleshooting

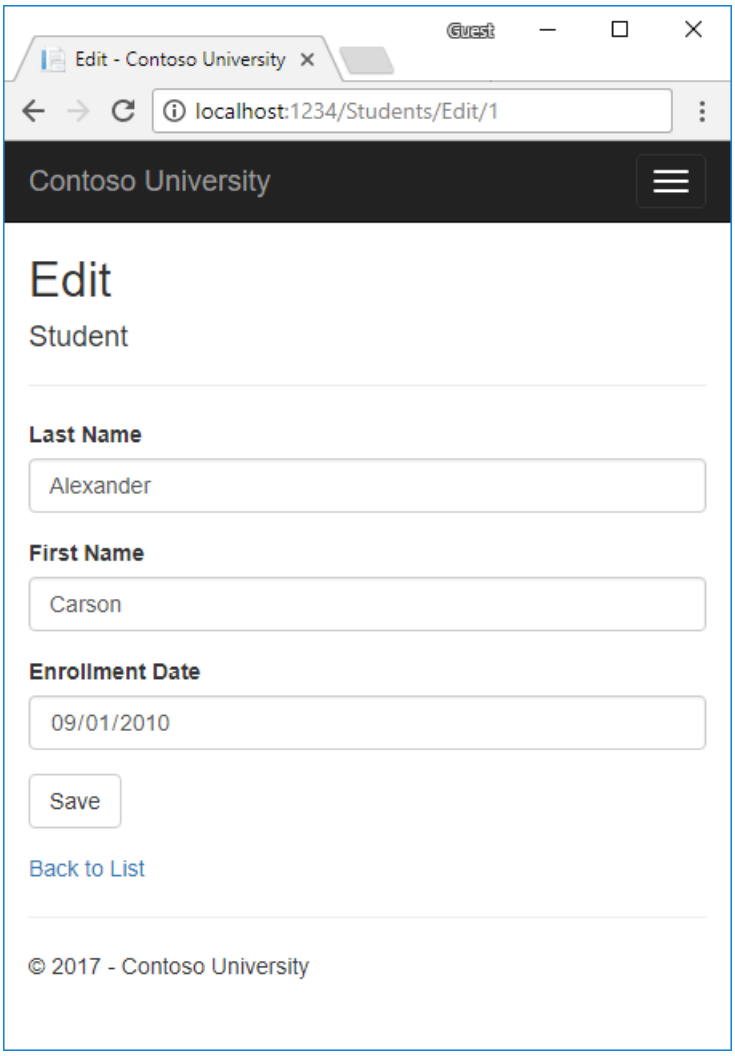
If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the completed project. A good way to get help is by posting a question to StackOverflow.com for ASP.NET Core or EF Core.

The Contoso University web app

The app built in these tutorials is a basic university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is close to what's generated by the built-in templates. The tutorial focus is on EF Core

with Razor Pages, not the UI.

Create the ContosoUniversity Razor Pages web app

- Visual Studio
- Visual Studio Code

- From the Visual Studio **File** menu, select **New** > **Project**.
- Create a new ASP.NET Core Web Application. Name the project **ContosoUniversity**. It's important to name the project *ContosoUniversity* so the namespaces match when code is copy/pasted.
- Select **ASP.NET Core 2.1** in the dropdown, and then select **Web Application**.

For images of the preceding steps, see Create a Razor web app. Run the app.

Set up the site style

A few changes set up the site menu, layout, and home page. Update *Pages/Shared/_Layout.cshtml* with the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Add menu entries for **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Contact** menu entry.

The changes are highlighted. (All the markup is *not* displayed.)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] : Contoso University</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-
test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">Contoso University</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Students/Index">Students</a></li>
                    <li><a asp-page="/Courses/Index">Courses</a></li>
                    <li><a asp-page="/Instructors/Index">Instructors</a></li>
                    <li><a asp-page="/Departments/Index">Departments</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 : Contoso University</p>
        </footer>
    </div>

    @*Remaining markup not shown for brevity.*@
```

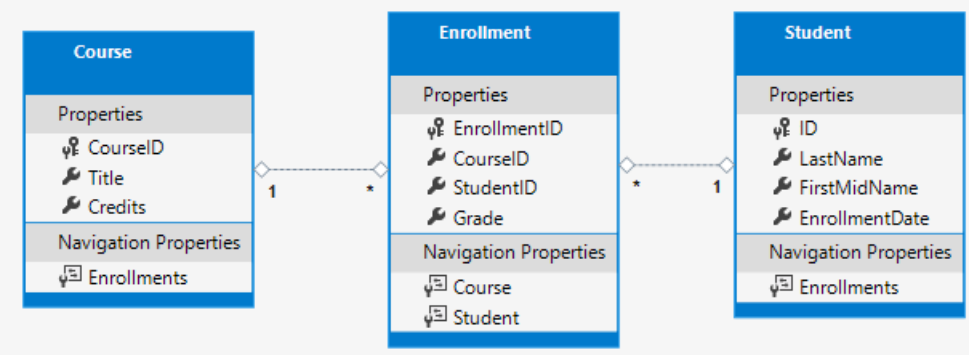
In *Pages/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this app:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core Razor Pages web app.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p>
            <a class="btn btn-default"
               href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro">
                See the tutorial &raquo;
            </a>
        </p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p>
            <a class="btn btn-default"
               href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-
rp/intro/samples/">
                See project source code &raquo;
            </a>
        </p>
    </div>
</div>
```

Create the data model

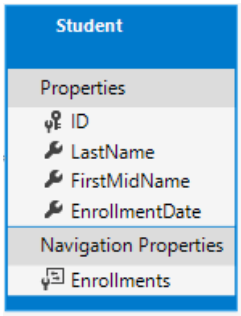
Create entity classes for the Contoso University app. Start with the following three entities:



There's a one-to-many relationship between `Student` and `Enrollment` entities. There's a one-to-many relationship between `Course` and `Enrollment` entities. A student can enroll in any number of courses. A course can have any number of students enrolled in it.

In the following sections, a class for each one of these entities is created.

The Student entity



Create a *Models* folder. In the *Models* folder, create a class file named *Student.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

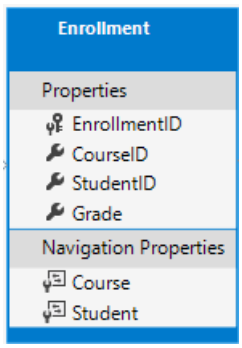
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property becomes the primary key column of the database (DB) table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. In `classnameID`, `classname` is the name of the class. The alternative automatically recognized primary key is `StudentID` in the preceding example.

The `Enrollments` property is a navigation property. Navigation properties link to other entities that are related to this entity. In this case, the `Enrollments` property of a `Student entity` holds all of the `Enrollment` entities that are related to that `Student`. For example, if a Student row in the DB has two related Enrollment rows, the `Enrollments` navigation property contains those two `Enrollment` entities. A related `Enrollment` row is a row that contains that student's primary key value in the `StudentID` column. For example, suppose the student with ID=1 has two rows in the `Enrollment` table. The `Enrollment` table has two rows with `StudentID` = 1. `StudentID` is a foreign key in the `Enrollment` table that specifies the student in the `Student` table.

If a navigation property can hold multiple entities, the navigation property must be a list type, such as `ICollection<T>`. `ICollection<T>` can be specified, or a type such as `List<T>` or `HashSet<T>`. When `ICollection<T>` is used, EF Core creates a `HashSet<T>` collection by default. Navigation properties that hold multiple entities come from many-to-many and one-to-many relationships.

The Enrollment entity



In the *Models* folder, create *Enrollment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key. This entity uses the `classnameID` pattern instead of `ID` like the `Student` entity. Typically developers choose one pattern and use it throughout the data model. In a later tutorial, using ID without classname is shown to make it easier to implement inheritance in the data model.

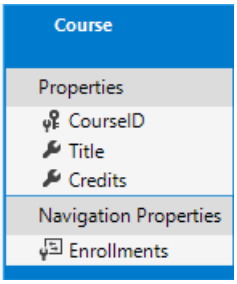
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity. The `Student` entity differs from the `Student.Enrollments` navigation property, which contains multiple `Enrollment` entities.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



In the *Models* folder, create *Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the DB generate it.

Scaffold the student model

In this section, the student model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the student model.

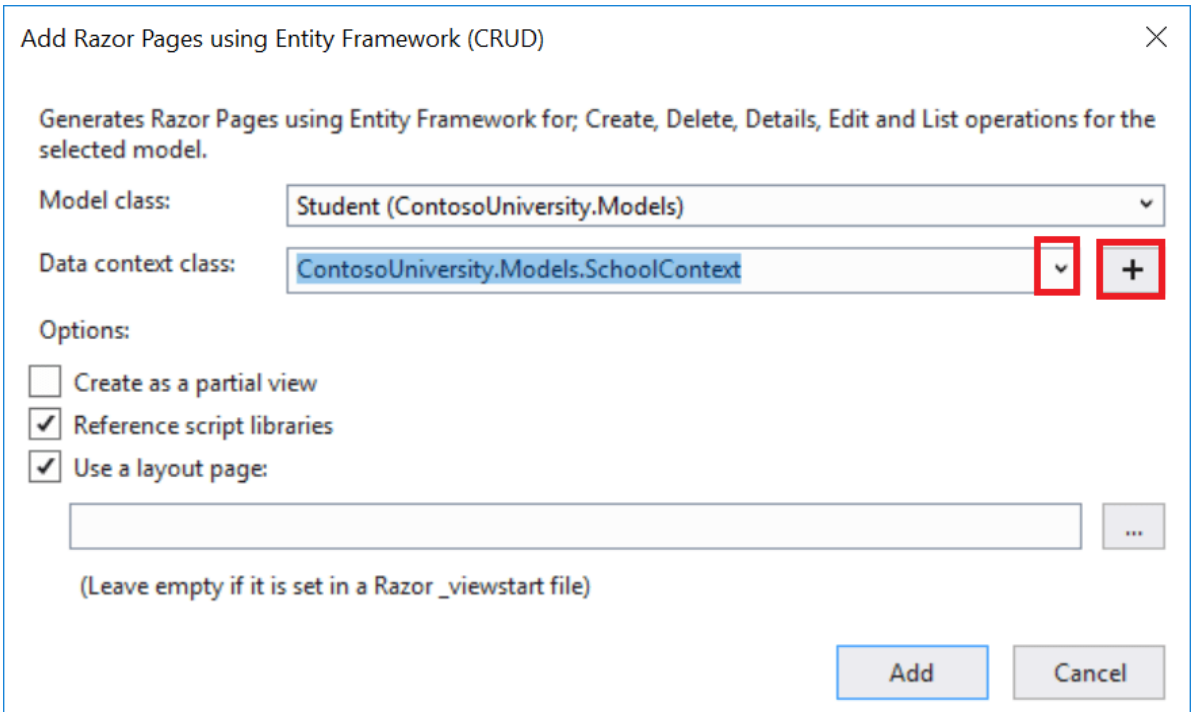
- Build the project.
- Create the *Pages/Students* folder.

- Visual Studio
- Visual Studio Code

- In **Solution Explorer**, right click on the *Pages/Students* folder > **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **ADD**.

Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:

- In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
- In the **Data context class** row, select the **+** (plus) sign and change the generated name to **ContosoUniversity.Models.SchoolContext**.
- In the **Data context class** drop-down, select **ContosoUniversity.Models.SchoolContext**
- Select **Add**.



See Scaffold the movie model if you have a problem with the preceding step.

The scaffold process created and changed the following files:

Files created

- *Pages/Students* Create, Delete, Details, Edit, Index.
- *Data/SchoolContext.cs*

File updates

- *Startup.cs* : Changes to this file are detailed in the next section.
- *appsettings.json* : The connection string used to connect to a local database is added.

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a db context instance is shown later in the tutorial.

The scaffolding tool automatically created a DB Context and registered it with the dependency injection container.

Examine the `ConfigureServices` method in *Startup.cs*. The highlighted line was added by the scaffolder:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for
        //non -essential cookies is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddDbContext<SchoolContext>(options =>
       options.UseSqlServer(Configuration.GetConnectionString("SchoolContext")));
}
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Update main

In *Program.cs*, modify the `Main` method to do the following:

- Get a DB context instance from the dependency injection container.
- Call the EnsureCreated.
- Dispose the context when the `EnsureCreated` method completes.

The following code shows the updated *Program.cs* file.

```
using ContosoUniversity.Models;                  // SchoolContext
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;   // CreateScope
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<SchoolContext>();
                    context.Database.EnsureCreated();
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the DB.");
                }
            }

            host.Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

`EnsureCreated` ensures that the database for the context exists. If it exists, no action is taken. If it does not exist, then the database and all its schema are created. `EnsureCreated` does not use migrations to create the database. A database that is created with `EnsureCreated` cannot be later updated using migrations.

`EnsureCreated` is called on app start, which allows the following work flow:

- Delete the DB.
- Change the DB schema (for example, add an `EmailAddress` field).
- Run the app.
- `EnsureCreated` creates a DB with the `EmailAddress` column.

`EnsureCreated` is convenient early in development when the schema is rapidly evolving. Later in the tutorial the DB is deleted and migrations are used.

Test the app

Run the app and accept the cookie policy. This app doesn't keep personal information. You can read about the cookie policy at EU General Data Protection Regulation (GDPR) support.

- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Examine the SchoolContext DB context

The main class that coordinates EF Core functionality for a given data model is the DB context class. The data context is derived from Microsoft.EntityFrameworkCore.DbContext. The data context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update *SchoolContext.cs* with the following code:

```
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Models
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Student { get; set; }
        public DbSet<Enrollment> Enrollment { get; set; }
        public DbSet<Course> Course { get; set; }
    }
}
```

The highlighted code creates a DbSet<TEntity> property for each entity set. In EF Core terminology:

- An entity set typically corresponds to a DB table.
- An entity corresponds to a row in the table.

`DbSet<Enrollment>` and `DbSet<Course>` could be omitted. EF Core includes them implicitly because the `Student` entity references the `Enrollment` entity, and the `Enrollment` entity references the `Course` entity. For this tutorial, keep `DbSet<Enrollment>` and `DbSet<Course>` in the `SchoolContext`.

SQL Server Express LocalDB

The connection string specifies SQL Server LocalDB. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB creates *.mdf* DB files in the `C:/Users/<user>` directory.

Add code to initialize the DB with test data

EF Core creates an empty DB. In this section, an `Initialize` method is written to populate it with test data.

In the *Data* folder, create a new class file named *DbInitializer.cs* and add the following code:

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Models
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Student.Any())
            {
```

```
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-
09-01")},
            new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-
09-01")},
            new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-
01")},
            new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-
09-01")},
            new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-
01")},
            new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-
01")},
            new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-
01")}
            };
            foreach (Student s in students)
            {
                context.Student.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Course.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollment.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

Note: The preceding code uses `Models` for the namespace (`namespace ContosoUniversity.Models`) rather than `Data` . `Models` is consistent with the scaffolder-generated code. For more information, see this GitHub scaffolding issue.

The code checks if there are any students in the DB. If there are no students in the DB, the DB is initialized with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

The `EnsureCreated` method automatically creates the DB for the DB context. If the DB exists, `EnsureCreated` returns without modifying the DB.

In *Program.cs*, modify the `Main` method to call `Initialize` :

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<SchoolContext>();
                // using ContosoUniversity.Data;
                DbInitializer.Initialize(context);
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred creating the DB.");
            }
        }

        host.Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

- Visual Studio
- Visual Studio Code

Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

View the DB

The database name is generated from the context name you provided earlier plus a dash and a GUID. Thus, the database name will be "SchoolContext-{GUID}". The GUID will be different for each user. Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio. In SSOX, click **(localdb)\MSSQLLocalDB > Databases > SchoolContext-{GUID}**.

Expand the **Tables** node.

Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into

the table.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task OnGetAsync()
{
    Student = await _context.Student.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Automatically create the Task object that's returned. For more information, see Task Return Type.
- The implicit return type `Task` represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the DB are executed asynchronously. That includes, `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the DB.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

In the next tutorial, basic CRUD (create, read, update, delete) operations are examined.

Additional resources

- YouTube version of this tutorial

NEXT

Part 2, Razor Pages with EF Core in ASP.NET Core - CRUD

9/22/2020 • 22 minutes to read • Edit Online

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, you add enrollments to the Details page.

Read enrollments

To display a student's enrollment data on the page, you need to read it. The scaffolded code in *Pages/Students/Details.cshtml.cs* reads only the Student data, without the Enrollment data:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the Reading related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in *Pages/Students/Details.cshtml* with the following code to display a list of enrollments. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for

the selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is unnecessary if the query can only return one entity, as when it searches on a unique key.
- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in *Pages/Students/Create.cshtml.cs* with the following code.

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` has to be converted to DateTime.

Run the app, and create a student entity to test the Create page.

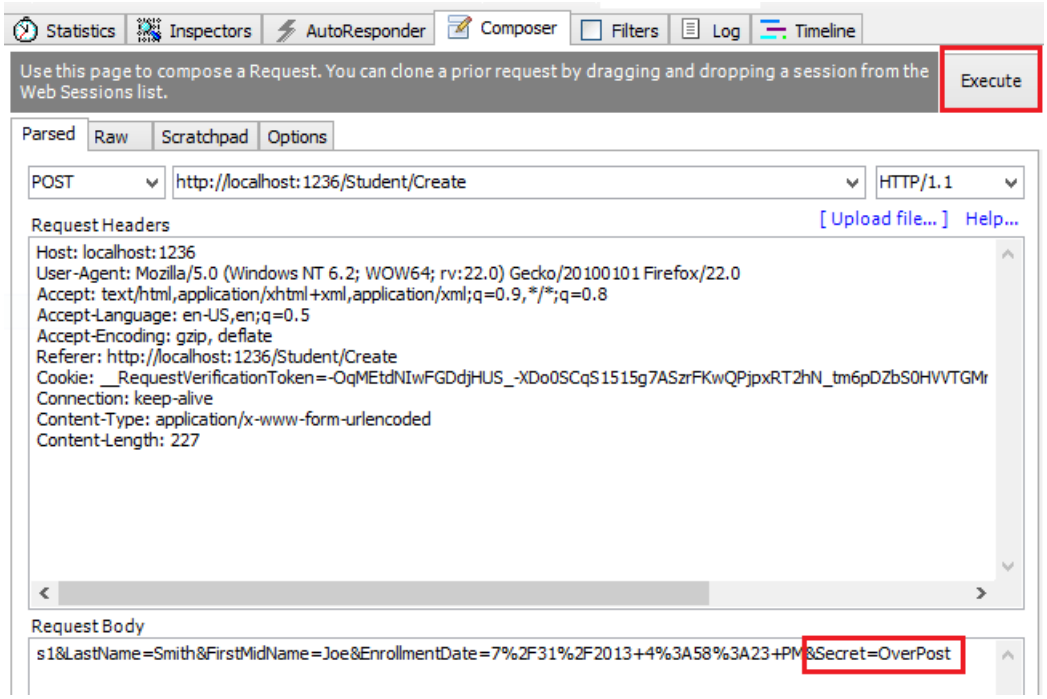
Overposting

Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI that it is used for (for example, the Create page).

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `Student` view model:

```
using System;

namespace ContosoUniversity.Models
{
    public class StudentVM
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }
    }
}
```

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type doesn't need to be related to the model type, it just needs to have properties that match.

Using `StudentVM` requires Create.cshtml be updated to use `StudentVM` rather than `Student`.

Update the Edit page

In *Pages/Students/Edit.cshtml.cs*, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

    if (studentToUpdate == null)
    {
        return NotFound();
    }

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When you don't have to include related data, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.

- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`: The entity isn't being tracked by the database context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL UPDATE statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, you implement a custom error message when the call to `SaveChanges` fails.

Replace the code in *Pages/Students/Delete.cshtml.cs* with the following code. The changes are highlighted (other than cleanup of `using` statements).

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Student == null)
            {
                return NotFound();
            }

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = "Delete failed. Try again";
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
```

```
        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var student = await _context.Students.FindAsync(id);

            if (student == null)
            {
                return NotFound();
            }

            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException /* ex */)
            {
                //Log the error (uncomment ex variable name and write a log.)
                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object. The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is false when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` (because the delete operation failed), the `saveChangesError` parameter is true.

The `OnPostAsync` method retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated. If `Remove` fails:

- The database exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to the Delete Razor Page (*Pages/Students/Delete.cshtml*):

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

PREVIOUS TUTORIAL NEXT TUTORIAL

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

To minimize complexity and keep these tutorials focused on EF Core, EF Core code is used in the page models. Some developers use a service layer or repository pattern in to create an abstraction layer between the UI (Razor Pages) and the data access layer.

In this tutorial, the Create, Edit, Delete, and Details Razor Pages in the *Students* folder are examined.

The scaffolded code uses the following pattern for Create, Edit, and Delete pages:

- Get and display the requested data with the HTTP GET method `OnGetAsync`.

- Save changes to the data with the HTTP POST method `OnPostAsync`.

The Index and Details pages get and display the requested data with the HTTP GET method `OnGetAsync`

SingleOrDefaultAsync vs. FirstOrDefaultAsync

The generated code uses FirstOrDefaultAsync, which is generally preferred over SingleOrDefaultAsync.

`FirstOrDefaultAsync` is more efficient than `SingleOrDefaultAsync` at fetching one entity:

- Unless the code needs to verify that there's not more than one entity returned from the query.
- `SingleOrDefaultAsync` fetches more data and does unnecessary work.
- `SingleOrDefaultAsync` throws an exception if there's more than one entity that fits the filter part.
- `FirstOrDefaultAsync` doesn't throw if there's more than one entity that fits the filter part.

FindAsync

In much of the scaffolded code, FindAsync can be used in place of `FirstOrDefaultAsync`.

`FindAsync`:

- Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the DB.
- Is simple and concise.
- Is optimized to look up a single entity.
- Can have perf benefits in some situations, but that rarely happens for typical web apps.
- Implicitly uses FirstAsync instead of SingleAsync.

But if you want to `Include` other entities, then `FindAsync` is no longer appropriate. This means that you may need to abandon `FindAsync` and move to a query as your app progresses.

Customize the Details page

Browse to `Pages/Students` page. The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the *Pages/Students/Index.cshtml* file.

```
<td>
    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
</td>
```

Run the app and select a **Details** link. The URL is of the form `http://localhost:5000/Students/Details?id=2`. The Student ID is passed using a query string (`?id=2`).

Update the Edit, Details, and Delete Razor Pages to use the `"{id:int}"` route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`.

A request to the page with the "{id:int}" route template that does **not** include a integer route value returns an HTTP 404 (not found) error. For example, `http://localhost:5000/Students/Details` returns a 404 error. To make the ID optional, append `?` to the route constraint:

```
@page "{id:int?}"
```

Run the app, click on a Details link, and verify the URL is passing the ID as route data (

`http://localhost:5000/Students/Details/2`).

Don't globally change `@page` to `@page "{id:int}"`, doing so breaks the links to the Home and Create pages.

Add related data

The scaffolded code for the Students Index page doesn't include the `Enrollments` property. In this section, the contents of the `Enrollments` collection is displayed in the Details page.

The `OnGetAsync` method of *Pages/Students/Details.cshtml.cs* uses the `FirstOrDefaultAsync` method to retrieve a single `Student` entity. Add the following highlighted code:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Student
                        .Include(s => s.Enrollments)
                            .ThenInclude(e => e.Course)
                        .AsNoTracking()
                        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the reading-related data tutorial.

The AsNoTracking method improves performance in scenarios when the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display related enrollments on the Details page

Open *Pages/Students/Details.cshtml*. Add the following highlighted code to display a list of enrollments:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd>
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

If code indentation is wrong after the code is pasted, press CTRL-K-D to correct it.

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for the selected student is displayed.

Update the Create page

Update the `OnPostAsync` method in *Pages/Students/Create.cshtml.cs* with the following code:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Student.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return null;
}
```

TryUpdateModelAsync

Examine the TryUpdateModelAsync code:

```
var emptyStudent = new Student();

if (await TryUpdateModelAsync<Student>(
    emptyStudent,
    "student",   // Prefix for form value.
    s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
{
```

In the preceding code, `TryUpdateModelAsync<Student>` tries to update the `emptyStudent` object using the posted form values from the PageContext property in the PageModel. `TryUpdateModelAsync` only updates the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).

In the preceding sample:

- The second argument (`"student", // Prefix`) is the prefix uses to look up values. It's not case sensitive.
- The posted form values are converted to the types in the `Student` model using model binding.

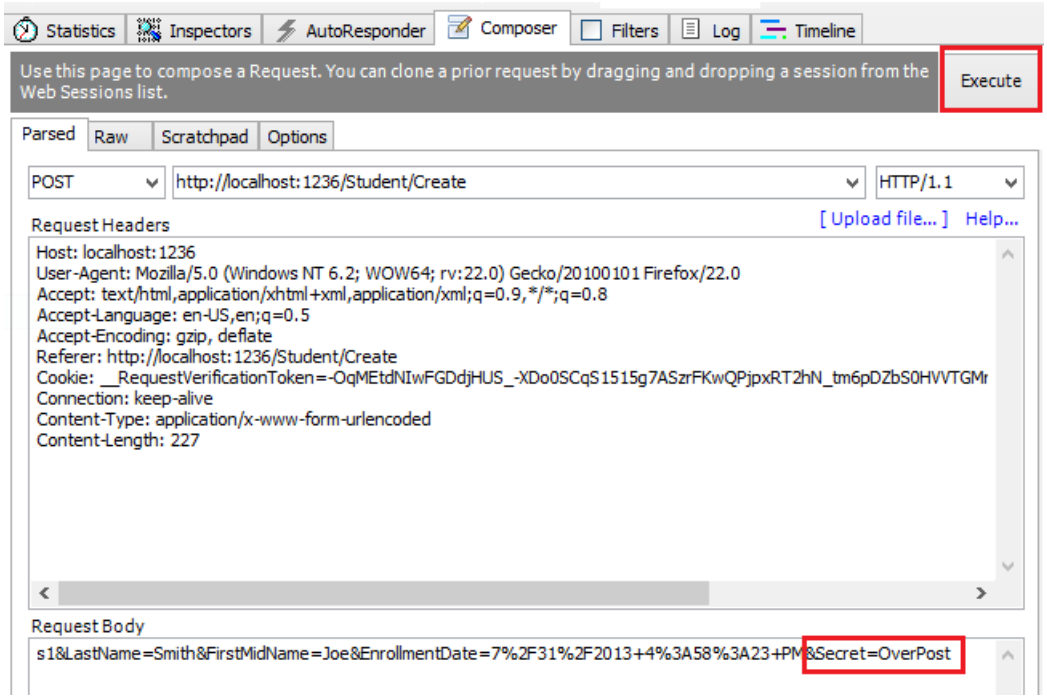
Overposting

Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if the app doesn't have a `Secret` field on the create/update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the DB. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. The app designer never intended the `Secret` property to be set with the Create page.

View model

A view model typically contains a subset of the properties included in the model used by the application. The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the DB. The view model contains only the properties needed for the UI layer (for example, the Create page). In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser. Consider the following `Student` view model:

```
using System;

namespace ContosoUniversity.Models
{
    public class StudentVM
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }
    }
}
```

View models provide an alternative way to prevent overposting. The view model contains only the properties to view (display) or update.

The following code uses the `StudentVM` view model to create a new student:

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type doesn't need to be related to the model type, it just needs to have properties that match.

Using `StudentVM` requires CreateVM.cshtml be updated to use `StudentVM` rather than `Student`.

In Razor Pages, the `PageModel` derived class is the view model.

Update the Edit page

Update the page model for the Edit page. The major changes are highlighted:

```
public class EditModel : PageModel
{
    private readonly SchoolContext _context;

    public EditModel(SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Student = await _context.Student.FindAsync(id);

        if (Student == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var studentToUpdate = await _context.Student.FindAsync(id);

        if (await TryUpdateModelAsync<Student>(
            studentToUpdate,
            "student",
            s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
        {
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }

        return Page();
    }
}
```

The code changes are similar to the Create page with a few exceptions:

- `OnPostAsync` has an optional `id` parameter.
- The current student is fetched from the DB, rather than creating an empty student.
- `FirstOrDefaultAsync` has been replaced with FindAsync. `FindAsync` is a good choice when selecting an entity from the primary key. See FindAsync for more information.

Test the Edit and Create pages

Create and edit a few student entities.

Entity States

The DB context keeps track of whether entities in memory are in sync with their corresponding rows in the DB.

The DB context sync information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the DB context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the DB. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the DB.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`: The entity isn't being tracked by the DB context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state to automatically be changed to `Modified`. Calling `SaveChanges` generates a SQL UPDATE statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Re-reading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, code is added to implement a custom error message when the call to `SaveChanges` fails. Add a string to contain possible error messages:

```
public class DeleteModel : PageModel
{
    private readonly SchoolContext _context;

    public DeleteModel(SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Student Student { get; set; }
    public string ErrorMessage { get; set; }
```

Replace the `OnGetAsync` method with the following code:

```
public async Task<IActionResult> OnGetAsync(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Student
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ErrorMessage = "Delete failed. Try again";
    }

    return Page();
}
```

The preceding code contains the optional parameter `saveChangesError`. `saveChangesError` indicates whether the method was called after a failure to delete the student object. The delete operation might fail because of transient network problems. Transient network errors are more likely in the cloud. `saveChangesError` is false when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` (because the delete operation failed), the `saveChangesError` parameter is true.

The Delete pages OnPostAsync method

Replace the `OnPostAsync` with the following code:

```
public async Task<IActionResult> OnPostAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Student
                    .AsNoTracking()
                    .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    try
    {
        _context.Student.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("./Delete",
                             new { id, saveChangesError = true });
    }
}
```

The preceding code retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated. If `Remove` fails:

- The DB exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Update the Delete Razor Page

Add the following highlighted error message to the Delete Razor Page.

```
@page "{id:int}"
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
```

Test Delete.

Common errors

Students/Index or other links don't work:

Verify the Razor Page contains the correct `@page` directive. For example, The Students/Index Razor Page should **not** contain a route template:

```
@page "{id:int}"
```

Each Razor Page must include the `@page` directive.

Additional resources

- YouTube version of this tutorial

Part 3, Razor Pages with EF Core in ASP.NET Core - Sort, Filter, Paging

9/22/2020 • 29 minutes to read • Edit Online

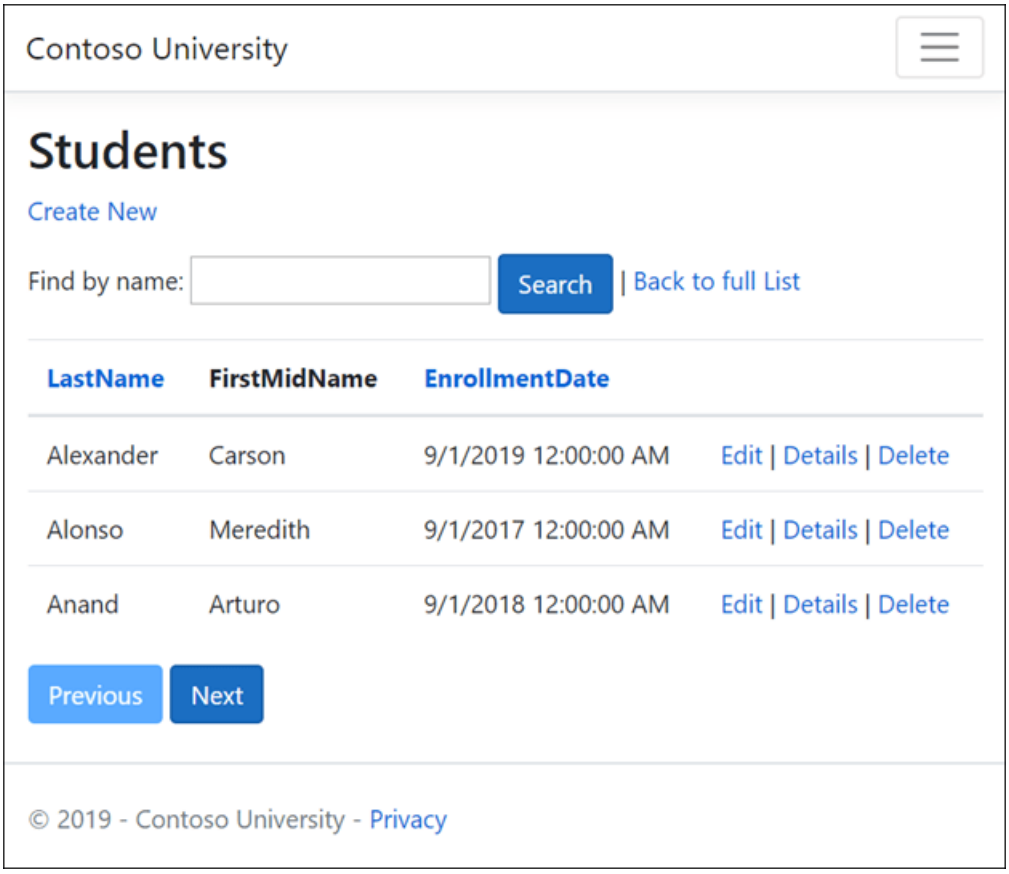
By Tom Dykstra, Rick Anderson, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial adds sorting, filtering, and paging functionality to the Students pages.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Click a column heading repeatedly to switch between ascending and descending sort order.



Add sorting

Replace the code in *Pages/Students/Index.cshtml.cs* with the following code to add sorting.

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;

        public IndexModel(SchoolContext context)
        {
            _context = context;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }

        public IList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder)
        {
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;

            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            Students = await studentsIQ.AsNoTracking().ToListAsync();
        }
    }
}
```

The preceding code:

- Adds properties to contain the sorting parameters.
- Changes the name of the `Student` property to `Students`.
- Replaces the code in the `OnGetAsync` method.

The `OnGetAsync` method receives a `sortOrder` parameter from the query string in the URL. The URL (including the query string) is generated by the Anchor Tag Helper.

The `sortOrder` parameter is either "Name" or "Date." The `sortOrder` parameter is optionally followed by "_desc" to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the default (fall-through case) in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The code uses the C# conditional operator ?:. The `?:` operator is a ternary operator (it takes three operands). The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to "name_desc." If `sortOrder` is **not** null or empty, `NameSort` is set to an empty string.

These two statements enable the page to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

```
IQueryable<Student> studentsIQ = from s in _context.Students
                                 select s;

switch (sortOrder)
{
    case "name_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
        break;
    case "Date":
        studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
        break;
    case "date_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
        break;
    default:
        studentsIQ = studentsIQ.OrderBy(s => s.LastName);
        break;
}

Students = await studentsIQ.AsNoTracking().ToListAsync();
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the

following statement:

```
Students = await studentsIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns. For information about an alternative way to code this functionality, see Use dynamic LINQ to simplify code in the MVC version of this tutorial series.

Add column heading hyperlinks to the Student Index page

Replace the code in *Students/Index.cshtml*, with the following code. The changes are highlighted.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.
- Changes the page heading from Index to Students.
- Changes `Model.Student` to `Model.Students`.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click the column headings.

Add filtering

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Update the OnGetAsync method

Replace the code in *Students/Index.cshtml.cs* with the following code to add filtering:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;

        public IndexModel(SchoolContext context)
        {
            _context = context;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }

        public IList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder, string searchString)
        {
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";

            CurrentFilter = searchString;

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s => s.LastName.Contains(searchString)
                                       || s.FirstMidName.Contains(searchString));
            }

            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            Students = await studentsIQ.AsNoTracking().ToListAsync();
        }
    }
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method, and saves the parameter value in the

`CurrentFilter` property. The search string value is received from a text box that's added in the next section.

- Adds to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is executed only if there's a value to search for.

IQueryable vs. IEnumerable

The code calls the `Where` method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. SQLite defaults to case-sensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())`
```

The preceding code would ensure that the filter is case-insensitive even if the `Where` method is called on an `IEnumerable` or runs on SQLite.

When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used.

Calling `Contains` on an `IQueryable` is usually preferable for performance reasons. With `IQueryable`, the filtering is done by the database server. If an `IEnumerable` is created first, all the rows have to be returned from the database server.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

For more information, see How to use case-insensitive query with Sqlite provider.

Update the Razor page

Replace the code in *Pages/Students/Index.cshtml* to create a **Search** button and assorted chrome.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string. If you're using SQLite, the filter is case-insensitive only if you implemented the optional `ToUpper` code shown earlier.
- Select **Search**.

Notice that the URL contains the search string. For example:

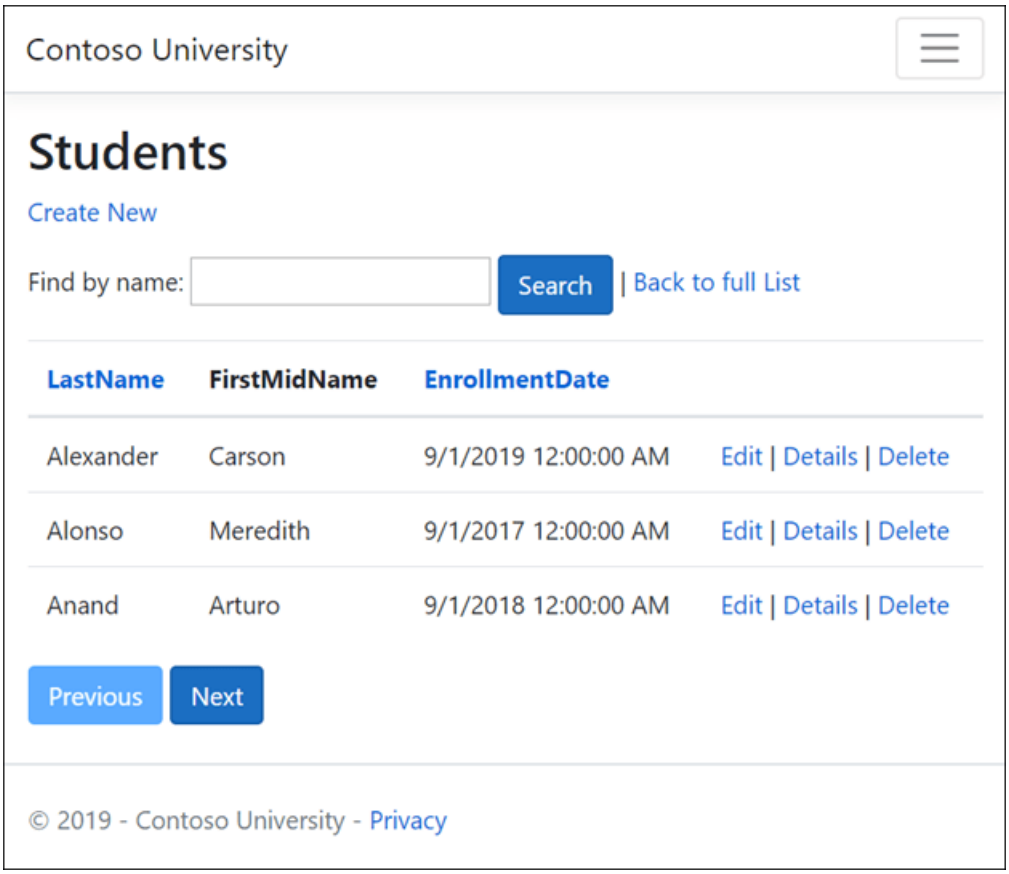
```
https://localhost:<port>/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



Create the PaginatedList class

In the project folder, create `PaginatedList.cs` with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object; constructors can't run asynchronous code.

Add paging to the PageModel class

Replace the code in *Students/Index.cshtml.cs* to add paging.

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;

        public IndexModel(SchoolContext context)
        {
            _context = context;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }

        public PaginatedList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder,
            string currentFilter, string searchString, int? pageIndex)
        {
            CurrentSort = sortOrder;
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";
            if (searchString != null)
            {
                pageIndex = 1;
            }
            else
            {
                searchString = currentFilter;
            }

            CurrentFilter = searchString;

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s => s.LastName.Contains(searchString)
                                       || s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            int pageSize = 3;
            Students = await PaginatedList<Student>.CreateAsync(
                studentsIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
        }
    }
}
```

The preceding code:

- Changes the type of the `Students` property from `IList<Student>` to `PaginatedList<Student>`.
- Adds the page index, the current `sortOrder`, and the `currentFilter` to the `OnGetAsync` method signature.
- Saves the sort order in the CurrentSort property.
- Resets page index to 1 when there's a new search string.
- Uses the `PaginatedList` class to get Student entities.

All the parameters that `OnGetAsync` receives are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

The `CurrentSort` property provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

The `CurrentFilter` property provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

The two question marks after `pageIndex` in the `PaginatedList.CreateAsync` call represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `(pageIndex ?? 1)` means return the value of `pageIndex` if it has a value. If `pageIndex` doesn't have a value, return 1.

Add paging links to the Razor Page

Replace the code in *Students/Index.cshtml* with the following code. The changes are highlighted:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
```

```
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.Students.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Students.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method:

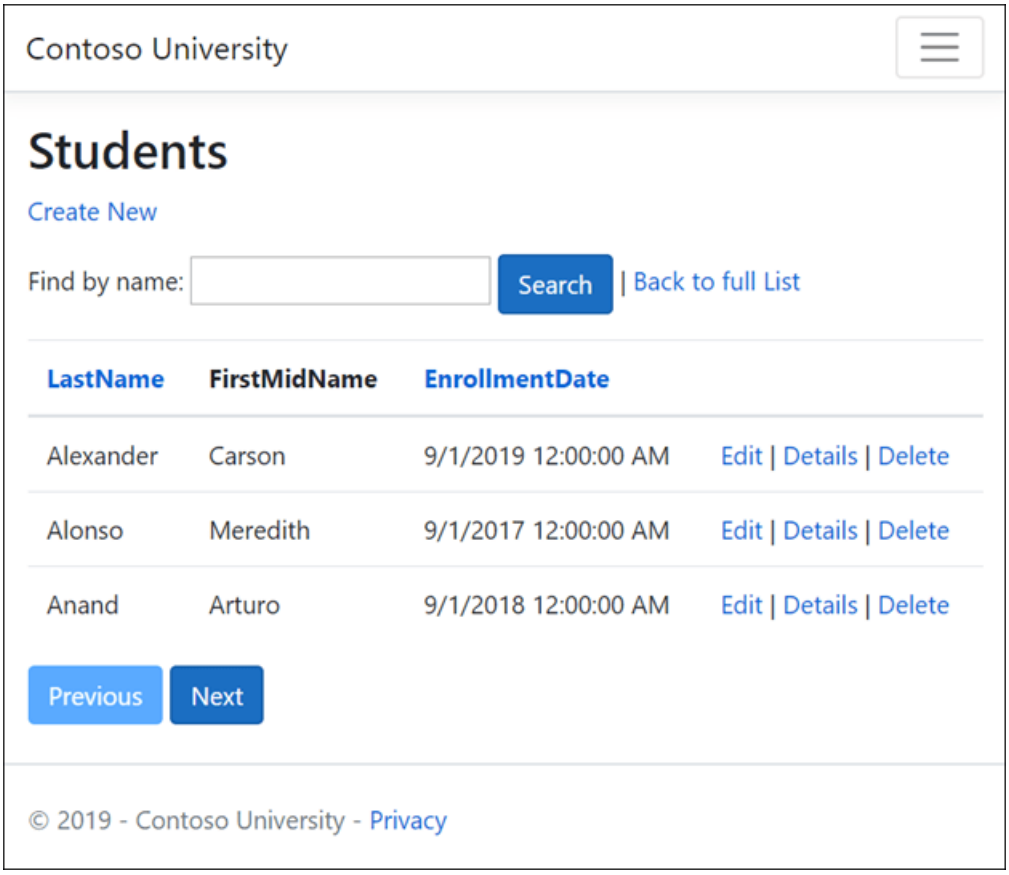
```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Students[0].LastName)
</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



Add grouping

This section creates an About page that displays how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the **About** page.
- Update the About page to use the view model.

Create the view model

Create a *Models/SchoolViewModels* folder.

Create *SchoolViewModels/EnrollmentDateGroup.cs* with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Create the Razor Page

Create a *Pages/About.cshtml* file with the following code:

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Students)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Create the page model

Create a *Pages/About.cshtml.cs* file with the following code:

```
using ContosoUniversity.Models.SchoolViewModels;
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

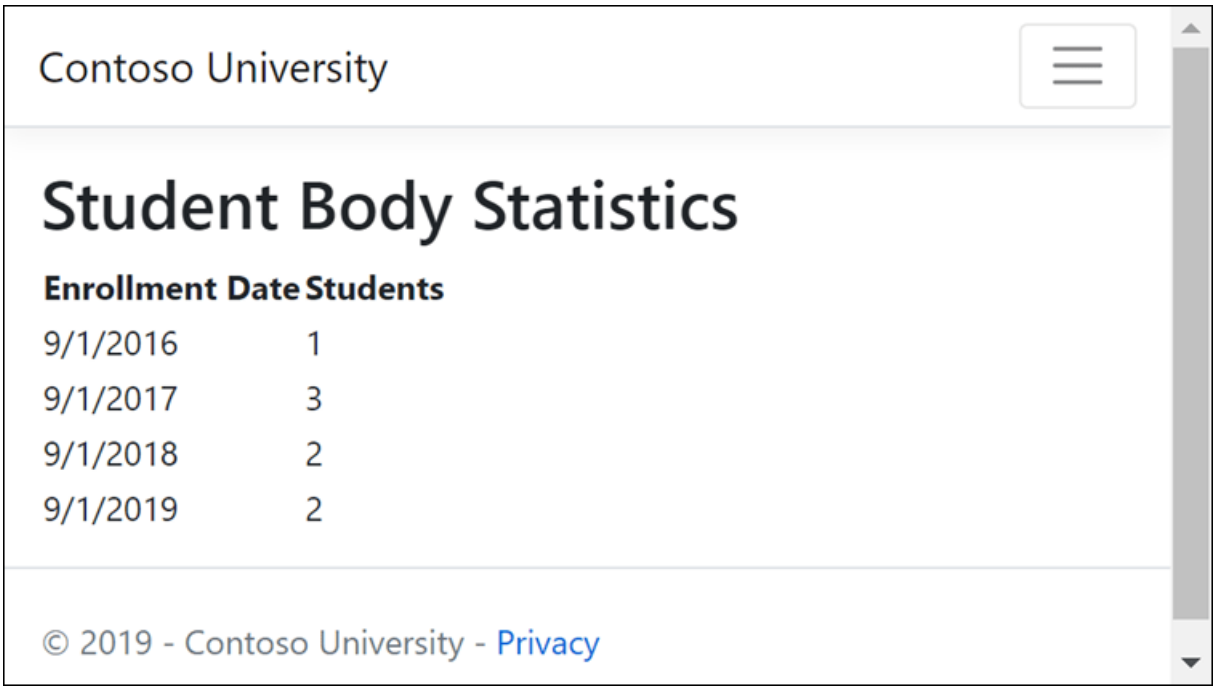
        public IList<EnrollmentDateGroup> Students { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Students = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.



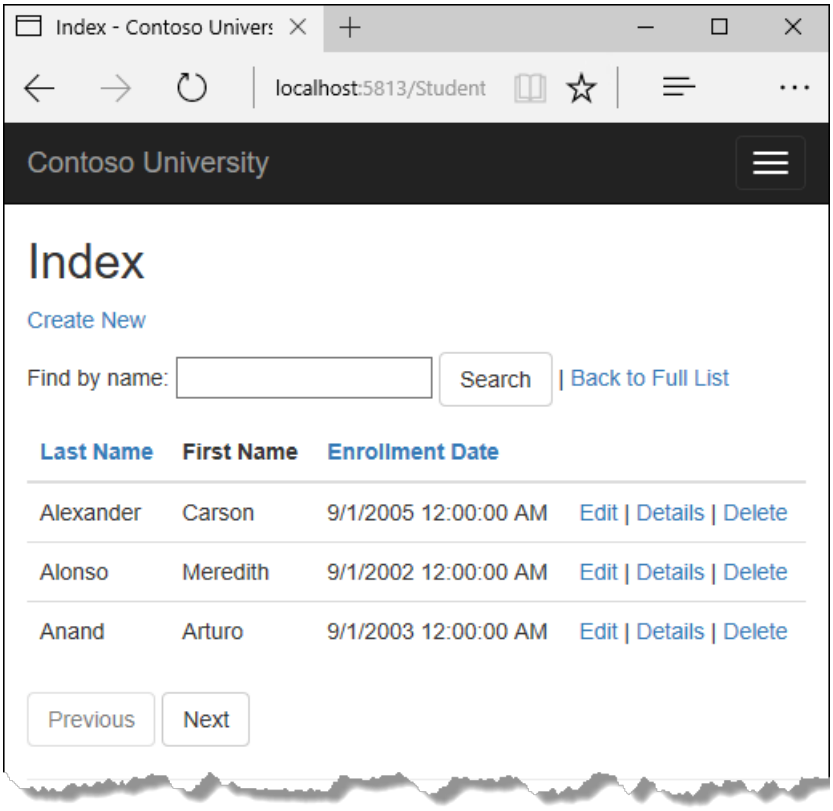
Next steps

In the next tutorial, the app uses migrations to update the data model.

PREVIOUS TUTORIAL NEXT TUTORIAL

In this tutorial, sorting, filtering, grouping, and paging, functionality is added.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Clicking a column heading repeatedly switches between ascending and descending sort order.



If you run into problems you can't solve, download the completed app.

Add sorting to the Index page

Add strings to the *Students/Index.cshtml.cs* `PageModel` to contain the sorting parameters:

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }
```

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The preceding code receives a `sortOrder` parameter from the query string in the URL. The URL (including the query string) is generated by the Anchor Tag Helper

The `sortOrder` parameter is either "Name" or "Date." The `sortOrder` parameter is optionally followed by "_desc" to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the default (fall-through case) in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the

appropriate query string values:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The following code contains the C# conditional ?: operator:

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to "name_desc." If `sortOrder` is **not** null or empty, `NameSort` is set to an empty string.

The `?: operator` is also known as the ternary operator.

These two statements enable the page to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

```
public async Task OnGetAsync(string sortOrder)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following statement:

```
Student = await studentIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns.

Add column heading hyperlinks to the Student Index page

Replace the code in *Students/Index.cshtml*, with the following highlighted code:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click **Last Name**.
- Click **Enrollment Date**.

To get a better understanding of the code:

- In *Students/Index.cshtml.cs*, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort` and `DateSort`.
- In *Students/Index.cshtml*, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Add a Search Box to the Students Index page

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Add filtering functionality to the Index method

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder, string searchString)
{
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }

    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    Student = await studentIQ.AsNoTracking().ToListAsync();
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method. The search string value is received from a text box that's added in the next section.
- Added to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is executed only if there's a value to search for.

Note: The preceding code calls the `Where` method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory

collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())
```

The preceding code would ensure that results are case-insensitive if the code changes to use `IEnumerable`. When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used. Returning an `IEnumerable` from a repository can have a significant performance penalty:

1. All the rows are returned from the DB server.
2. The filter is applied to all the returned rows in the application.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

Add a Search Box to the Student Index page

In *Pages/Students/Index.cshtml*, add the following highlighted code to create a **Search** button and assorted chrome.

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
```

The preceding code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string.

- Select **Search**.

Notice that the URL contains the search string.

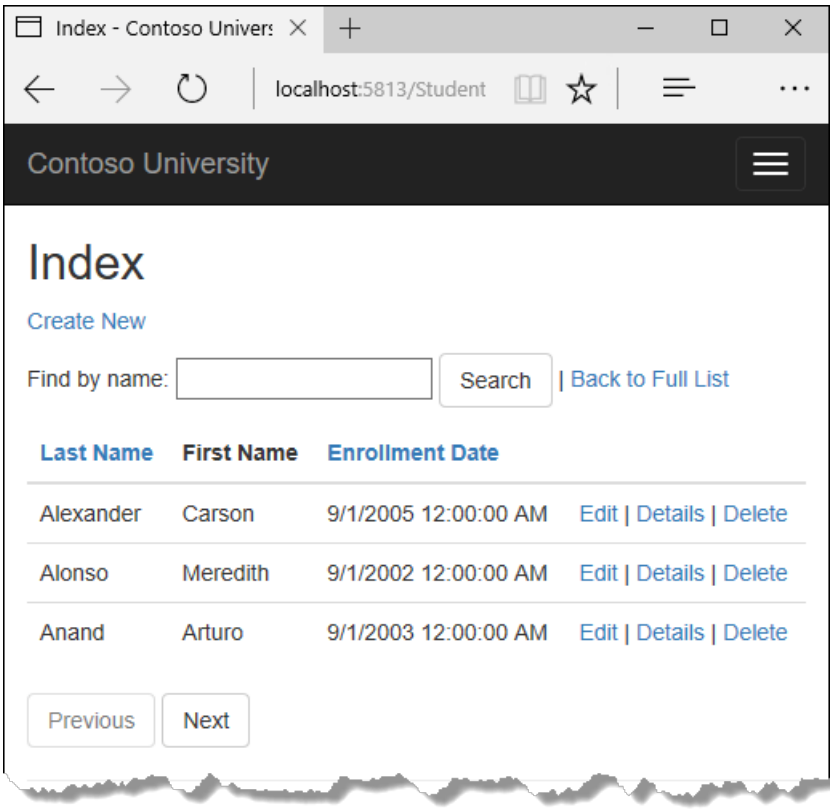
```
http://localhost:5000/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging functionality to the Students Index page

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs` with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object, constructors can't run asynchronous code.

Add paging functionality to the Index method

In *Students/Index.cshtml.cs*, update the type of `Student` from `IList<Student>` to `PaginatedList<Student>`:

```
public PaginatedList<Student> Student { get; set; }
```

Update the *Students/Index.cshtml.cs* `OnGetAsync` with the following code:

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
{
    CurrentSort = sortOrder;
    NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    DateSort = sortOrder == "Date" ? "date_desc" : "Date";
    if (searchString != null)
    {
        pageIndex = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    CurrentFilter = searchString;

    IQueryable<Student> studentIQ = from s in _context.Student
                                    select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        studentIQ = studentIQ.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            studentIQ = studentIQ.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            studentIQ = studentIQ.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            studentIQ = studentIQ.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    Student = await PaginatedList<Student>.CreateAsync(
        studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
}
```

The preceding code adds the page index, the current `sortOrder`, and the `currentFilter` to the method signature.

```
public async Task OnGetAsync(string sortOrder,
    string currentFilter, string searchString, int? pageIndex)
```

All the parameters are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

`CurrentSort` provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

`CurrentFilter` provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

```
if (searchString != null)
{
    pageIndex = 1;
}
else
{
    searchString = currentFilter;
}
```

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

```
Student = await PaginatedList<Student>.CreateAsync(
    studentIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
```

The two question marks in `PaginatedList.CreateAsync` represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `(pageIndex ?? 1)` means return the value of `pageIndex` if it has a value. If `pageIndex` doesn't have a value, return 1.

Add paging links to the student Razor Page

Update the markup in *Students/Index.cshtml*. The changes are highlighted:

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
```

```
                    asp-route-currentFilter="@Model.CurrentFilter">
                     @Html.DisplayNameFor(model => model.Student[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Student[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Student[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Student)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.Student.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Student.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method so that the user can sort within filter results:

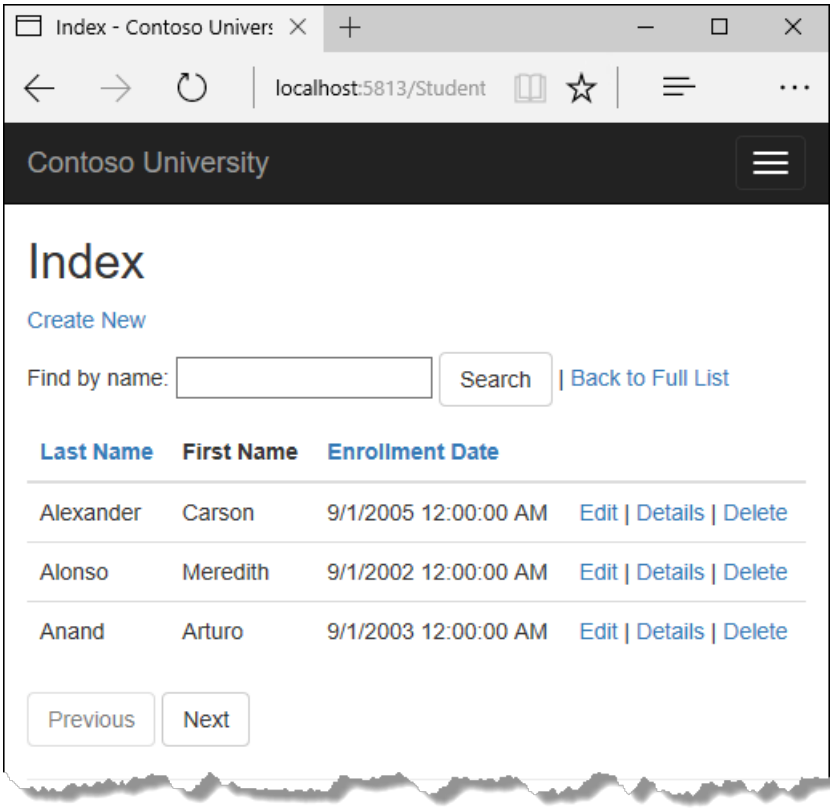
```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Student[0].LastName)
</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Student.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



To get a better understanding of the code:

- In *Students/Index.cshtml.cs*, set a breakpoint on `switch (sortOrder)`.
- Add a watch for `NameSort`, `DateSort`, `CurrentSort`, and `Model.Student.PageIndex`.
- In *Students/Index.cshtml*, set a breakpoint on `@Html.DisplayNameFor(model => model.Student[0].LastName)`.

Step through the debugger.

Update the About page to show student statistics

In this step, *Pages/About.cshtml* is updated to display how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the **About** Page.
- Update the About page to use the view model.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the *SchoolViewModels* folder, add a *EnrollmentDateGroup.cs* with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Update the About page model

The web templates in ASP.NET Core 2.2 do not include the About page. If you are using ASP.NET Core 2.2, create the About Razor Page.

Update the *Pages/About.cshtml.cs* file with the following code:

```
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Student { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Student
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Student = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Modify the About Razor Page

Replace the code in the *Pages/About.cshtml* file with the following code:

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

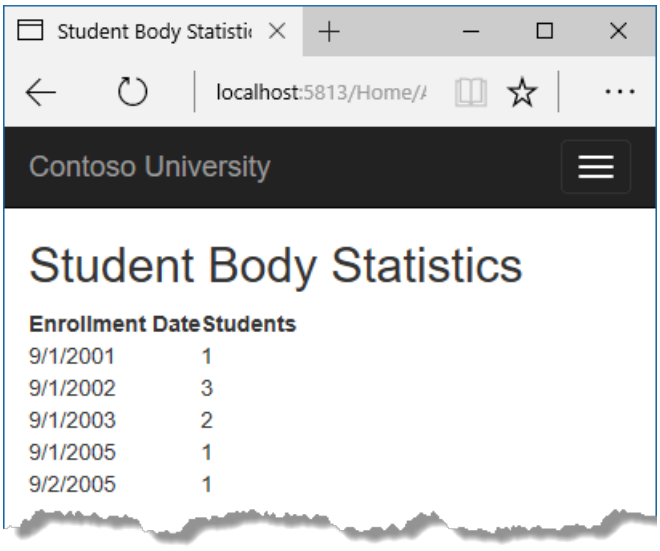
<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Student)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.

If you run into problems you can't solve, download the completed app for this stage.



Additional resources

- Debugging ASP.NET Core 2.x source
- YouTube version of this tutorial

In the next tutorial, the app uses migrations to update the data model.

PREVIOUS NEXT

Part 4, Razor Pages with EF Core migrations in ASP.NET Core

9/22/2020 • 11 minutes to read • Edit Online

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial introduces the EF Core migrations feature for managing data model changes.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial series started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes, you have to drop the database. The next time the app runs, the call to `EnsureCreated` re-creates the database to match the new data model. The `DbInitializer` class then runs to seed the new database.

This approach to keeping the database in sync with the data model works well until you deploy the app to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test database each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the database schema instead of creating a new database.

Rather than dropping and recreating the database when the data model changes, migrations updates the schema and retains existing data.

NOTE

SQLite limitations

This tutorial uses the Entity Framework Core *migrations* feature where possible. Migrations updates the database schema to match changes in the data model. However, migrations only does the kinds of changes that the database engine supports, and SQLite's schema change capabilities are limited. For example, adding a column is supported, but removing a column is not supported. If a migration is created to remove a column, the `ef migrations add` command succeeds but the `ef database update` command fails.

The workaround for the SQLite limitations is to manually write migrations code to perform a table rebuild when something in the table changes. The code would go in the `Up` and `Down` methods for a migration and would involve:

- Creating a new table.
- Copying data from the old table to the new table.
- Dropping the old table.
- Renaming the new table.

Writing database-specific code of this type is outside the scope of this tutorial. Instead, this tutorial drops and re-creates the database whenever an attempt to apply a migration would fail. For more information, see the following resources:

- SQLite EF Core Database Provider Limitations
- Customize migration code
- Data seeding
- SQLite ALTER TABLE statement

Drop the database

- Visual Studio
- Visual Studio Code

Use **SQL Server Object Explorer** (SSOX) to delete the database, or run the following command in the **Package Manager Console** (PMC):

```
Drop-Database
```

Create an initial migration

- Visual Studio
- Visual Studio Code

Run the following commands in the PMC:

```
Add-Migration InitialCreate
Update-Database
```

Up and Down methods

The EF Core `migrations add` command generated code to create the database. This migrations code is in the *Migrations<timestamp>_InitialCreate.cs* file. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

```
using System;
```

```
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ContosoUniversity.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Course",
                columns: table => new
                {
                    CourseID = table.Column<int>(nullable: false),
                    Title = table.Column<string>(nullable: true),
                    Credits = table.Column<int>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Course", x => x.CourseID);
                });

            migrationBuilder.CreateTable(
                name: "Student",
                columns: table => new
                {
                    ID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    LastName = table.Column<string>(nullable: true),
                    FirstMidName = table.Column<string>(nullable: true),
                    EnrollmentDate = table.Column<DateTime>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Student", x => x.ID);
                });

            migrationBuilder.CreateTable(
                name: "Enrollment",
                columns: table => new
                {
                    EnrollmentID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    CourseID = table.Column<int>(nullable: false),
                    StudentID = table.Column<int>(nullable: false),
                    Grade = table.Column<int>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Enrollment", x => x.EnrollmentID);
                    table.ForeignKey(
                        name: "FK_Enrollment_Course_CourseID",
                        column: x => x.CourseID,
                        principalTable: "Course",
                        principalColumn: "CourseID",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_Enrollment_Student_StudentID",
                        column: x => x.StudentID,
                        principalTable: "Student",
                        principalColumn: "ID",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_CourseID",
                table: "Enrollment",
```

```
                column: "CourseID");

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_StudentID",
                table: "Enrollment",
                column: "StudentID");
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Enrollment");

            migrationBuilder.DropTable(
                name: "Course");

            migrationBuilder.DropTable(
                name: "Student");
        }
    }
}
```

The preceding code is for the initial migration. The code:

- Was generated by the `migrations add InitialCreate` command.
- Is executed by the `database update` command.
- Creates a database for the data model specified by the database context class.

The migration name parameter ("InitialCreate" in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

The migrations history table

- Use SSOX or your SQLite tool to inspect the database.
- Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the database.
- View the data in the `__EFMigrationsHistory` table. It shows one row for the first migration.

The data model snapshot

Migrations creates a *snapshot* of the current data model in *Migrations/SchoolContextModelSnapshot.cs*. When you add a migration, EF determines what changed by comparing the current data model to the snapshot file.

Because the snapshot file tracks the state of the data model, you can't delete a migration by deleting the `<timestamp>_<migrationname>.cs` file. To back out the most recent migration, you have to use the `migrations remove` command. That command deletes the migration and ensures the snapshot is correctly reset. For more information, see dotnet ef migrations remove.

Remove EnsureCreated

This tutorial series started by using `EnsureCreated`. `EnsureCreated` doesn't create a migrations history table and so can't be used with migrations. It's designed for testing or rapid prototyping where the database is dropped and re-created frequently.

From this point forward, the tutorials will use migrations.

In *Data/DBInitializer.cs*, comment out the following line:

```
context.Database.EnsureCreated();
```

Run the app and verify that the database is seeded.

Applying migrations in production

We recommend that production apps **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app that is deployed to a server farm. If the app is scaled out to multiple server instances, it's hard to ensure database schema updates don't happen from multiple servers or conflict with read/write access.

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

Troubleshooting

If the app uses SQL Server LocalDB and displays the following exception:

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.
The login failed.
Login failed for user 'user name'.
```

The solution may be to run `dotnet ef database update` at a command prompt.

Additional resources

- EF Core CLI.
- Package Manager Console (Visual Studio)

Next steps

The next tutorial builds out the data model, adding entity properties and new entities.

PREVIOUS TUTORIAL NEXT TUTORIAL

In this tutorial, the EF Core migrations feature for managing data model changes is used.

If you run into problems you can't solve, download the completed app.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes:

- The DB is dropped.
- EF creates a new one that matches the model.
- The app seeds the DB with test data.

This approach to keeping the DB in sync with the data model works well until you deploy the app to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves

this problem by enabling EF Core to update the DB schema instead of creating a new DB.

Rather than dropping and recreating the DB when the data model changes, migrations updates the schema and retains existing data.

Drop the database

Use **SQL Server Object Explorer** (SSOX) or the `database drop` command:

- Visual Studio
- Visual Studio Code

In the **Package Manager Console** (PMC), run the following command:

```
Drop-Database
```

Run `Get-Help about_EntityFrameworkCore` from the PMC to get help information.

Create an initial migration and update the DB

Build the project and create the first migration.

- Visual Studio
- Visual Studio Code

```
Add-Migration InitialCreate
Update-Database
```

Examine the Up and Down methods

The EF Core `migrations add` command generated code to create the DB. This migrations code is in the *Migrations<timestamp>_InitialCreate.cs* file. The `Up` method of the `InitialCreate` class creates the DB tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true),
                Credits = table.Column<int>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        migrationBuilder.CreateTable(
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");

        migrationBuilder.DropTable(
            name: "Course");

        migrationBuilder.DropTable(
            name: "Student");
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, migrations calls the `Down` method.

The preceding code is for the initial migration. That code was created when the `migrations add InitialCreate` command was run. The migration name parameter ("InitialCreate" in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

If the initial migration is created and the DB exists:

- The DB creation code is generated.
- The DB creation code doesn't need to run because the DB already matches the data model. If the DB creation code is run, it doesn't make any changes because the DB already matches the data model.

When the app is deployed to a new environment, the DB creation code must be run to create the DB.

Previously the DB was dropped and doesn't exist, so migrations creates the new DB.

The data model snapshot

Migrations create a *snapshot* of the current database schema in *Migrations/SchoolContextModelSnapshot.cs*. When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

To delete a migration, use the following command:

- Visual Studio
- Visual Studio Code

Remove-Migration

The remove migrations command deletes the migration and ensures the snapshot is correctly reset.

Remove EnsureCreated and test the app

For early development, `EnsureCreated` was used. In this tutorial, migrations are used. `EnsureCreated` has the following limitations:

- Bypasses migrations and creates the DB and schema.
- Doesn't create a migrations table.
- Can *not* be used with migrations.
- Is designed for testing or rapid prototyping where the DB is dropped and re-created frequently.

Remove `EnsureCreated`:

```
context.Database.EnsureCreated();
```

Run the app and verify the DB is seeded.

Inspect the database

Use **SQL Server Object Explorer** to inspect the DB. Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the DB. View the data in the `__EFMigrationsHistory` table, it shows one row for the first migration. The last log in the preceding CLI output example shows the INSERT statement that creates this row.

Run the app and verify that everything works.

Applying migrations in production

We recommend production apps should **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app in server farm. For example, if the app has been cloud deployed with scale-out (multiple instances of the app are running).

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

EF Core uses the `__MigrationsHistory` table to see if any migrations need to run. If the DB is up-to-date, no migration is run.

Troubleshooting

Download the completed app.

The app generates the following exception:

```
SqlException: Cannot open database "ContosoUniversity" requested by the login.
The login failed.
Login failed for user 'user name'.
```

Solution: Run `dotnet ef database update`

Additional resources

- YouTube version of this tutorial
- .NET Core CLI.
- Package Manager Console (Visual Studio)

PREVIOUS NEXT

Part 5, Razor Pages with EF Core in ASP.NET Core - Data Model

9/22/2020 • 57 minutes to read • Edit Online

By Tom Dykstra and Rick Anderson

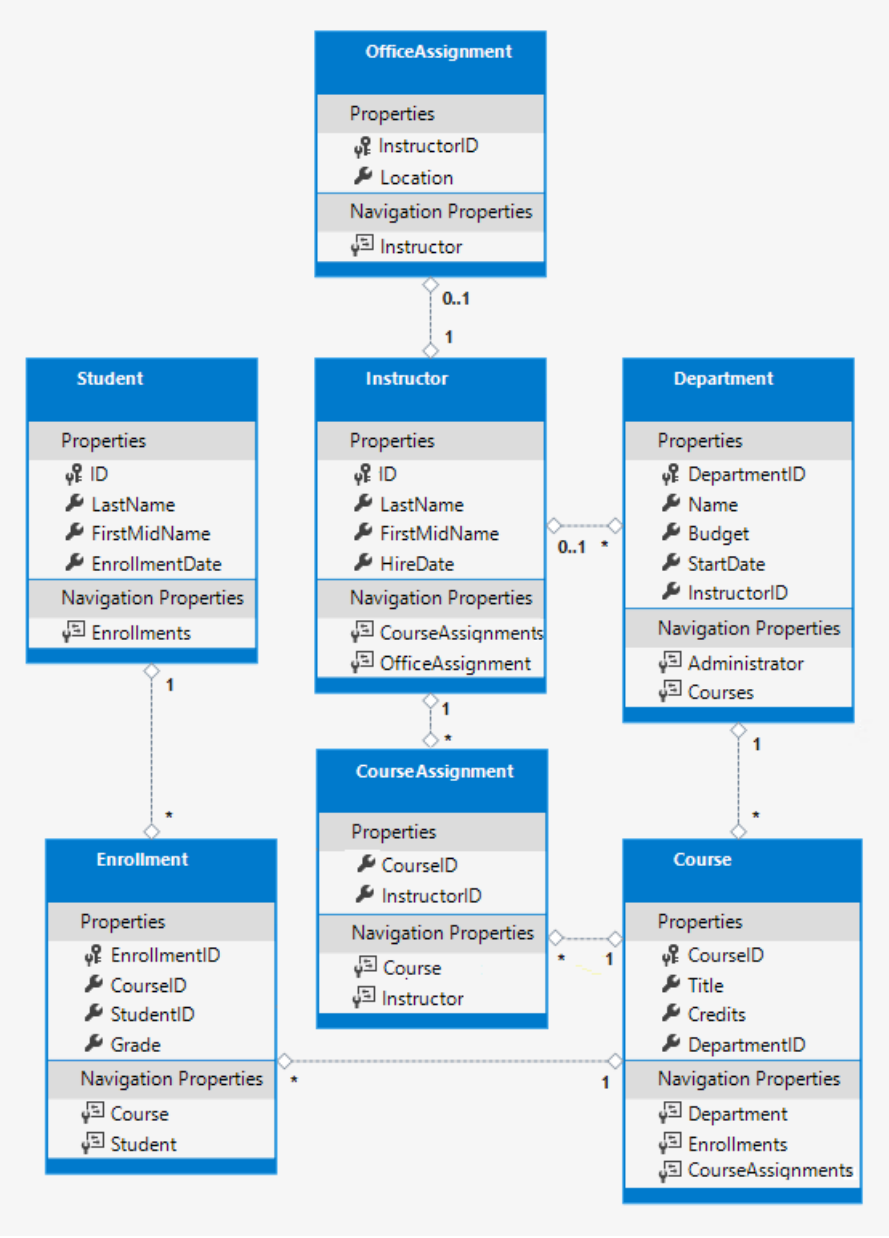
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

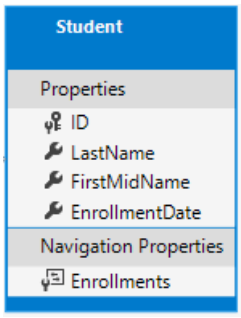
The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The completed data model is shown in the following illustration:



The Student entity



Replace the code in *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code adds a `FullName` property and adds the following attributes to existing properties:

- `[DataType]`
- `[DisplayFormat]`
- `[StringLength]`
- `[Column]`
- `[Required]`
- `[Display]`

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties.

`FullName` can't be set, so it has only a get accessor. No `FullName` column is created in the database.

The DataType attribute

```
[DataType(DataType.Date)]
```

For student enrollment dates, all of the pages currently display the time of day along with the date, although only the date is relevant. By using data annotation attributes, you can make one code change that will fix the display format in every page that shows the data.

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes. The `DataType` attributes don't provide validation.

The DisplayFormat attribute

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format. The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, and client-side input validation.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

The StringLength attribute

```
[StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
```

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The code shown limits names to no more than 50 characters. An example that sets the minimum string length is shown later.

The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

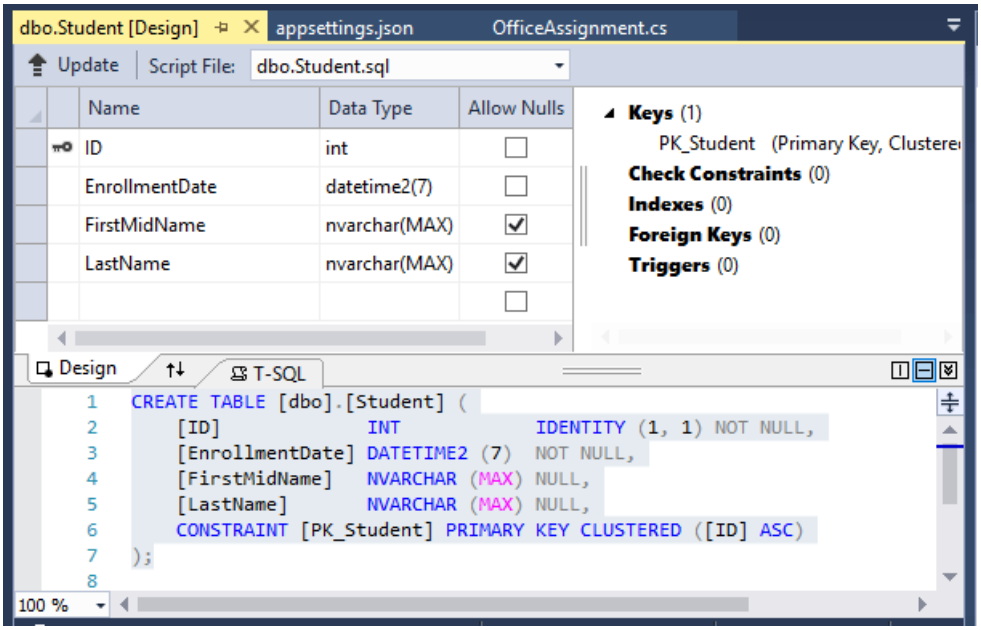
The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute can be used to apply restrictions to the input. For example, the following code requires the first

character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

- Visual Studio
- Visual Studio Code

In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)`. When a migration is created and applied later in this tutorial, the name fields become `nvarchar(50)` as a result of the string length attributes.

The Column attribute

```
[Column("FirstName")]
public string FirstMidName { get; set; }
```

Attributes can control how classes and properties are mapped to the database. In the `Student` model, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the database.

When the database is created, property names on the model are used for column names (except when the `Column` attribute is used). The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

With the `[Column]` attribute, `Student.FirstMidName` in the data model maps to the `FirstName` column of the `Student` table. The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. That discrepancy will be resolved by adding a migration later in this tutorial.

The Required attribute

```
[Required]
```

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (for example, `DateTime`, `int`, and `double`). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for full control over the string.

The Display attribute

```
[Display(Name = "Last Name")]
```

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

Create a migration

Run the app and go to the Students page. An exception is thrown. The `[Column]` attribute causes EF to expect to find a column named `FirstName`, but the column name in the database is still `FirstMidName`.

- Visual Studio
- Visual Studio Code

The error message is similar to the following example:

```
SqlException: Invalid column name 'FirstName'.
```

- In the PMC, enter the following commands to create a new migration and update the database:

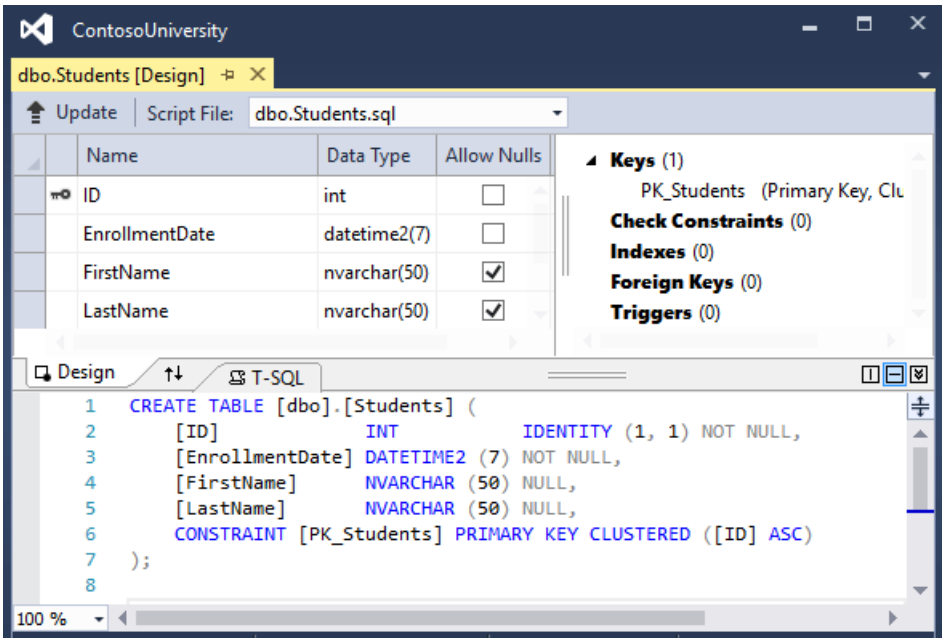
  ```
  Add-Migration ColumnFirstName
  Update-Database
  ```

  The first of these commands generates the following warning message:

  ```
  An operation was scaffolded that may result in the loss of data.
  Please review the migration for accuracy.
  ```

  The warning is generated because the name fields are now limited to 50 characters. If a name in the database had more than 50 characters, the 51 to last character would be lost.

- Open the Student table in SSOX:

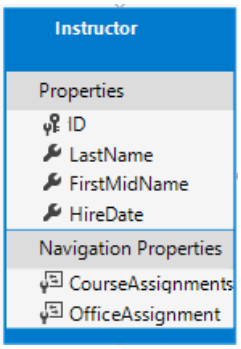


Before the migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

- Run the app and go to the Students page.
- Notice that times are not input or displayed along with dates.
- Select **Create New**, and try to enter a name longer than 50 characters.

> **NOTE**
> In the following sections, building the app at some stages generates compiler errors. The instructions specify when to build the app.

The Instructor Entity



Create *Models/Instructor.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

Navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

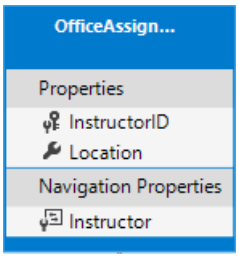
An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

An instructor can have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

```
public OfficeAssignment OfficeAssignment { get; set; }
```

The OfficeAssignment entity



Create *Models/OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than classnameID or ID.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity.

EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because `InstructorID` doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship.

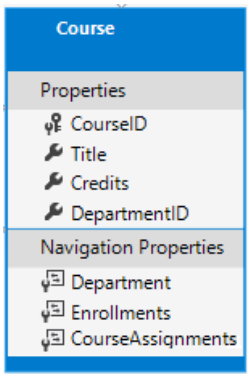
The Instructor navigation property

The `Instructor.OfficeAssignment` navigation property can be null because there might not be an `OfficeAssignment` row for a given instructor. An instructor might not have an office assignment.

The `OfficeAssignment.Instructor` navigation property will always have an instructor entity because the foreign key `InstructorID` type is `int`, a non-nullable value type. An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The Course Entity



Update *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a foreign key property for a data model when the model has a navigation property for a related entity. EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. However, explicitly including the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` property is null if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, EF Core assumes that PK values are generated by the database. Database-generated is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the database can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

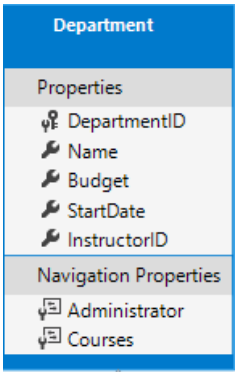
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection:

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

`CourseAssignment` is explained later.

The Department entity



Create *Models/Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required. EF Core chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

The question mark (?) in the preceding code specifies the property is nullable.

A department may have many courses, so there's a Courses navigation property:

```
public ICollection<Course> Courses { get; set; }
```

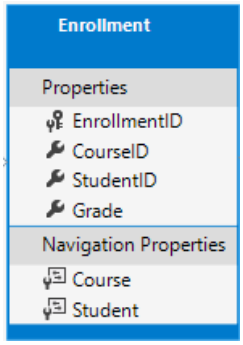
By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. This default behavior can result in circular cascade delete rules. Circular cascade delete rules cause an exception when a migration is added.

For example, if the `Department.InstructorID` property was defined as non-nullable, EF Core would configure a cascade delete rule. In that case, the department would be deleted when the instructor assigned as its administrator is deleted. In this scenario, a restrict rule would make more sense. The following fluent API would set a restrict rule and disable cascade delete.

```
modelBuilder.Entity<Department>()
   .HasOne(d => d.Administrator)
   .WithMany()
   .OnDelete(DeleteBehavior.Restrict)
```

The Enrollment entity

An enrollment record is for one course taken by one student.



Update *Models/Enrollment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

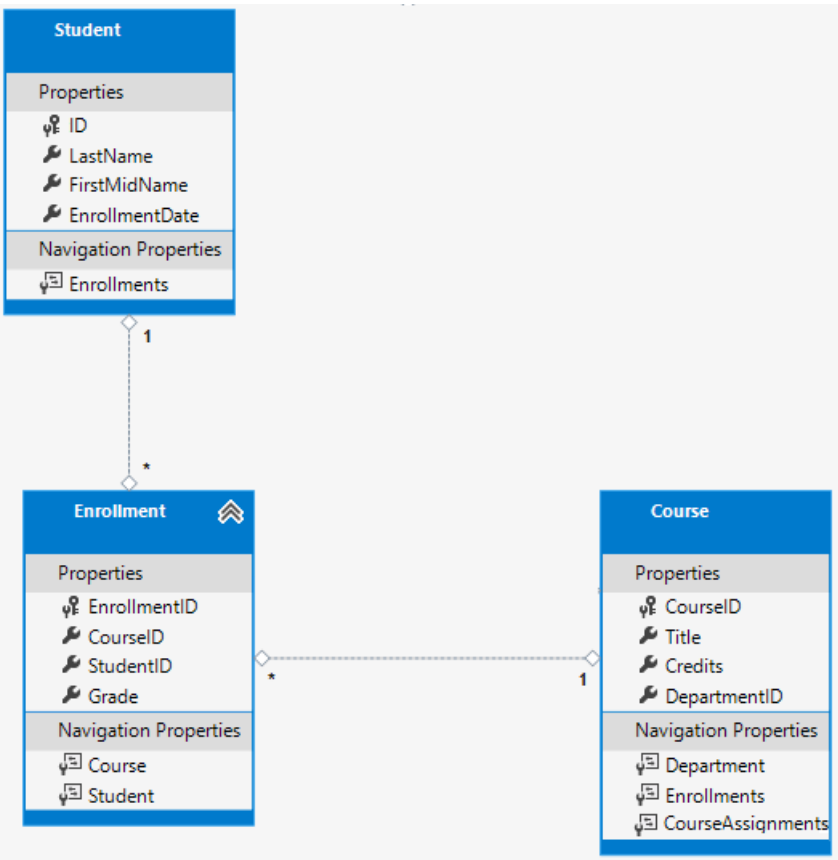
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities. The `Enrollment` entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the `Enrollment` table contains additional data besides FKs for the joined tables (in this case, the PK and `Grade`).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



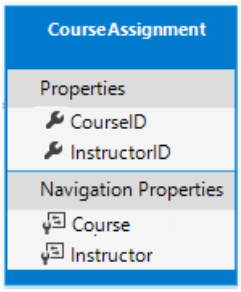
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs (`CourseID` and `StudentID`). A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a pure join table.

Note: EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see Many-to-many relationships in EF Core 2.0.

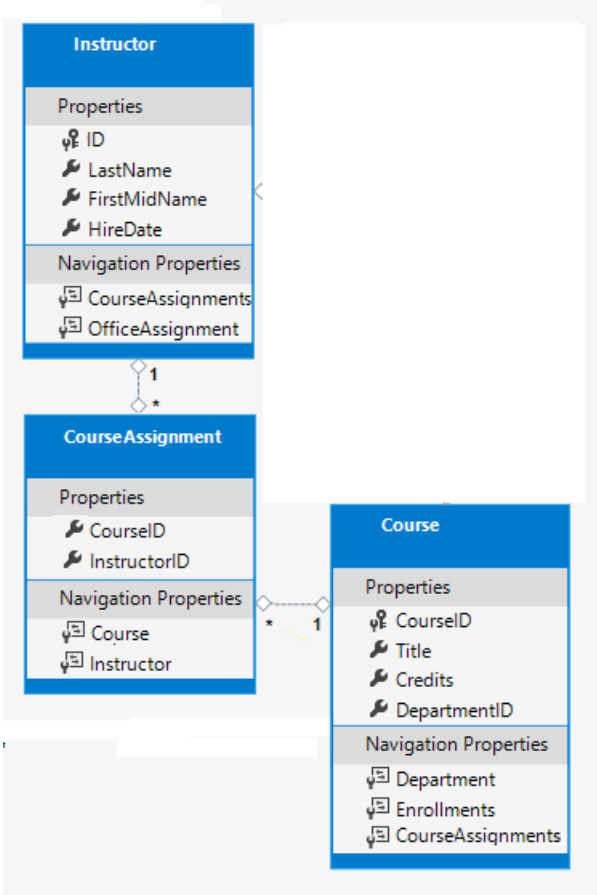
The CourseAssignment entity



Create *Models/CourseAssignment.cs* with the following code:

```
namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

The Instructor-to-Courses many-to-many relationship requires a join table, and the entity for that join table is CourseAssignment.



It's common to name a join entity `EntityName1EntityName2`. For example, the Instructor-to-Courses join table using this pattern would be `CourseInstructor`. However, we recommend using a name that describes the relationship.

Data models start out simple and grow. Join tables without payload (PJTs) frequently evolve to include payload. By starting with a descriptive entity name, the name doesn't need to change when the join table changes. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example,

Books and Customers could be linked with a join entity called Ratings. For the Instructor-to-Courses many-to-many relationship, `CourseAssignment` is preferred over `CourseInstructor`.

Composite key

The two FKs in `CourseAssignment` (`InstructorID` and `CourseID`) together uniquely identify each row of the `CourseAssignment` table. `CourseAssignment` doesn't require a dedicated PK. The `InstructorID` and `CourseID` properties function as a composite PK. The only way to specify composite PKs to EF Core is with the *fluent API*. The next section shows how to configure the composite PK.

The composite key ensures that:

- Multiple rows are allowed for one course.
- Multiple rows are allowed for one instructor.
- Multiple rows aren't allowed for the same instructor and course.

The `Enrollment` join entity defines its own PK, so duplicates of this sort are possible. To prevent such duplicates:

- Add a unique index on the FK fields, or
- Configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Update *Data/SchoolContext.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

The preceding code adds the new entities and configures the `CourseAssignment` entity's composite PK.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

In this tutorial, the fluent API is used only for database mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API (specifying a composite PK). There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

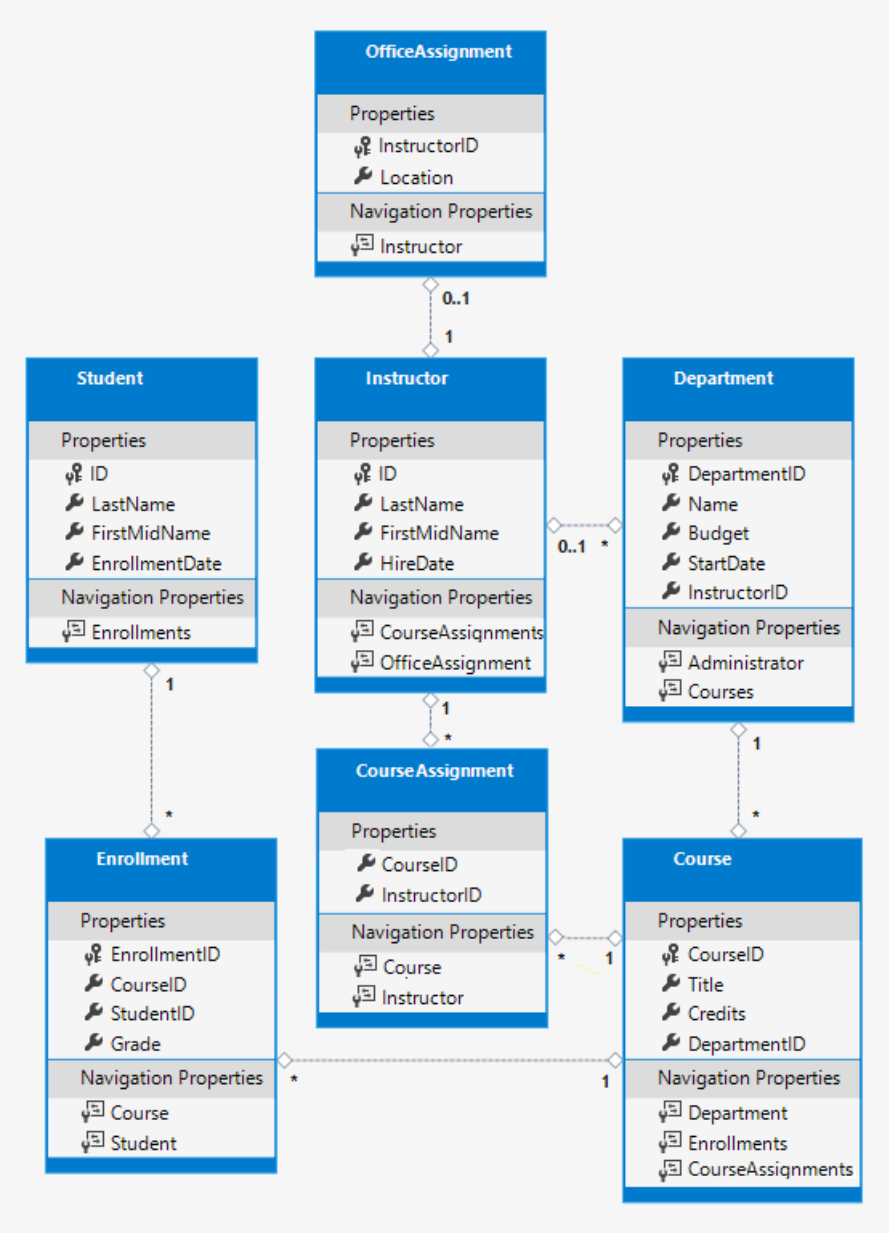
Some of the attributes used in this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Entity diagram

The following illustration shows the diagram that EF Power Tools create for the completed School model.



The preceding diagram shows:

- Several one-to-many relationship lines (1 to *).
- The one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment` entities.
- The zero-or-one-to-many relationship line (0..1 to *) between the `Instructor` and `Department` entities.

Seed the database

Update the code in *Data/DbInitializer.cs*:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();
```

```
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2016-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2019-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2018-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2017-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2019-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2011-09-01") }
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            context.Instructors.AddRange(instructors);
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
            };

            context.Departments.AddRange(departments);
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
                    DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
```

```
        },
        new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
        new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
    };

    context.Courses.AddRange(courses);
    context.SaveChanges();

    var officeAssignments = new OfficeAssignment[]
    {
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
            Location = "Smith 17" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
            Location = "Gowan 27" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
            Location = "Thompson 304" },
    };

    context.OfficeAssignments.AddRange(officeAssignments);
    context.SaveChanges();

    var courseInstructors = new CourseAssignment[]
    {
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
```

```
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
    };

    context.CourseAssignments.AddRange(courseInstructors);
    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            Grade = Grade.A
        },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            Grade = Grade.C
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
            Grade = Grade.B
            },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Li").ID,
            CourseID = courses.Single(c => c.Title == "Composition").CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Justice").ID,
            CourseID = courses.Single(c => c.Title == "Literature").CourseID,
            Grade = Grade.B
            }
    };

    foreach (Enrollment e in enrollments)
```

```
            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing. See `Enrollments` and `CourseAssignments` for examples of how many-to-many join tables can be seeded.

Add a migration

Build the project.

- Visual Studio
- Visual Studio Code

In PMC, run the following command.

```
Add-Migration ComplexDataModel
```

The preceding command displays a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
To undo this action, use 'ef migrations remove'
```

If the `database update` command is run, the following error is produced:

```
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint
"FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in
database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.
```

In the next section, you see what to do about this error.

Apply the migration or drop and re-create

Now that you have an existing database, you need to think about how to apply changes to it. This tutorial shows two alternatives:

- Drop and re-create the database. Choose this section if you're using SQLite.
- Apply the migration to the existing database. The instructions in this section work for SQL Server only, **not for SQLite**.

Either choice works for SQL Server. While the apply-migration method is more complex and time-consuming, it's the preferred approach for real-world, production environments.

Drop and re-create the database

Skip this section if you're using SQL Server and want to do the apply-migration approach in the following section.

To force EF Core to create a new database, drop and update the database:

- Visual Studio
- Visual Studio Code

- In the **Package Manager Console** (PMC), run the following command:

```
Drop-Database
```

- Delete the *Migrations* folder, then run the following command:

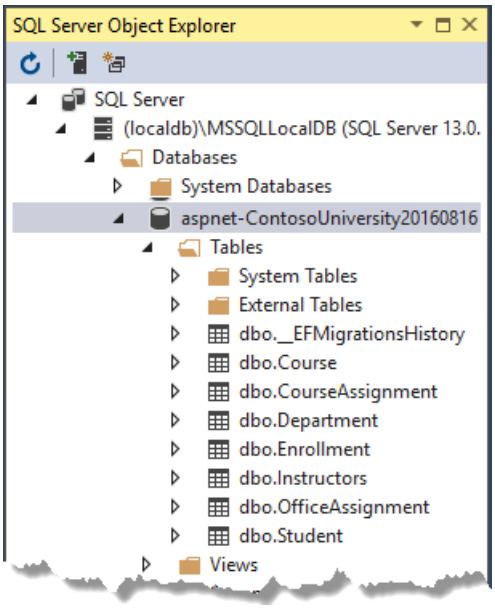
```
Add-Migration InitialCreate
Update-Database
```

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

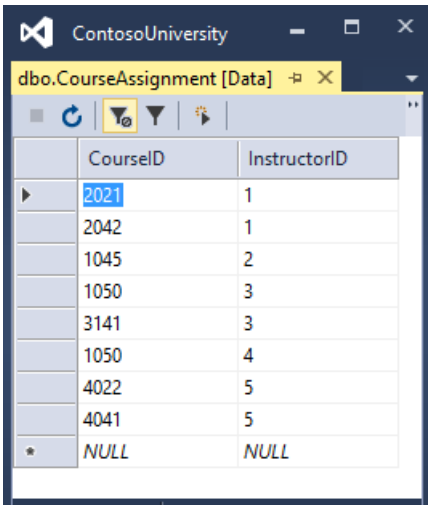
- Visual Studio
- Visual Studio Code

Open the database in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.



- Examine the **CourseAssignment** table:
  - Right-click the **CourseAssignment** table and select **View Data**.
  - Verify the **CourseAssignment** table contains data.



Apply the migration

This section is optional. These steps work only for SQL Server LocalDB and only if you skipped the preceding Drop and re-create the database section.

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. With production data, steps must be taken to migrate the existing data. This section provides an example of fixing FK constraint violations. Don't make these code changes without a backup. Don't make these code changes if you completed the preceding Drop and re-create the database section.

The *{timestamp}_ComplexDataModel.cs* file contains the following code:

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

The preceding code adds a non-nullable `DepartmentID` FK to the `Course` table. The database from the previous tutorial contains rows in `Course`, so that table cannot be updated by migrations.

To make the `ComplexDataModel` migration work with existing data:

- Change the code to give the new column (`DepartmentID`) a default value.
- Create a fake department named "Temp" to act as the default department.

Fix the foreign key constraints

In the `ComplexDataModel` migration class, update the `Up` method:

- Open the *{timestamp}_ComplexDataModel.cs* file.
- Comment out the line of code that adds the `DepartmentID` column to the `Course` table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

Add the following highlighted code. The new code goes after the `.CreateTable( name: "Department"` block:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(type: "int", nullable: true),
        Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(type: "datetime2", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

 migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

With the preceding changes, existing `Course` rows will be related to the "Temp" department after the `ComplexDataModel.Up` method runs.

The way of handling the situation shown here is simplified for this tutorial. A production app would:

- Include code or scripts to add `Department` rows and related `Course` rows to the new `Department` rows.
- Not use the "Temp" department or the default value for `Course.DepartmentID`.

- Visual Studio
- Visual Studio Code

- In the **Package Manager Console** (PMC), run the following command:

```
Update-Database
```

Because the `DbInitializer.Initialize` method is designed to work only with an empty database, use SSOX to delete all the rows in the Student and Course tables. (Cascade delete will take care of the Enrollment table.)

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

Next steps

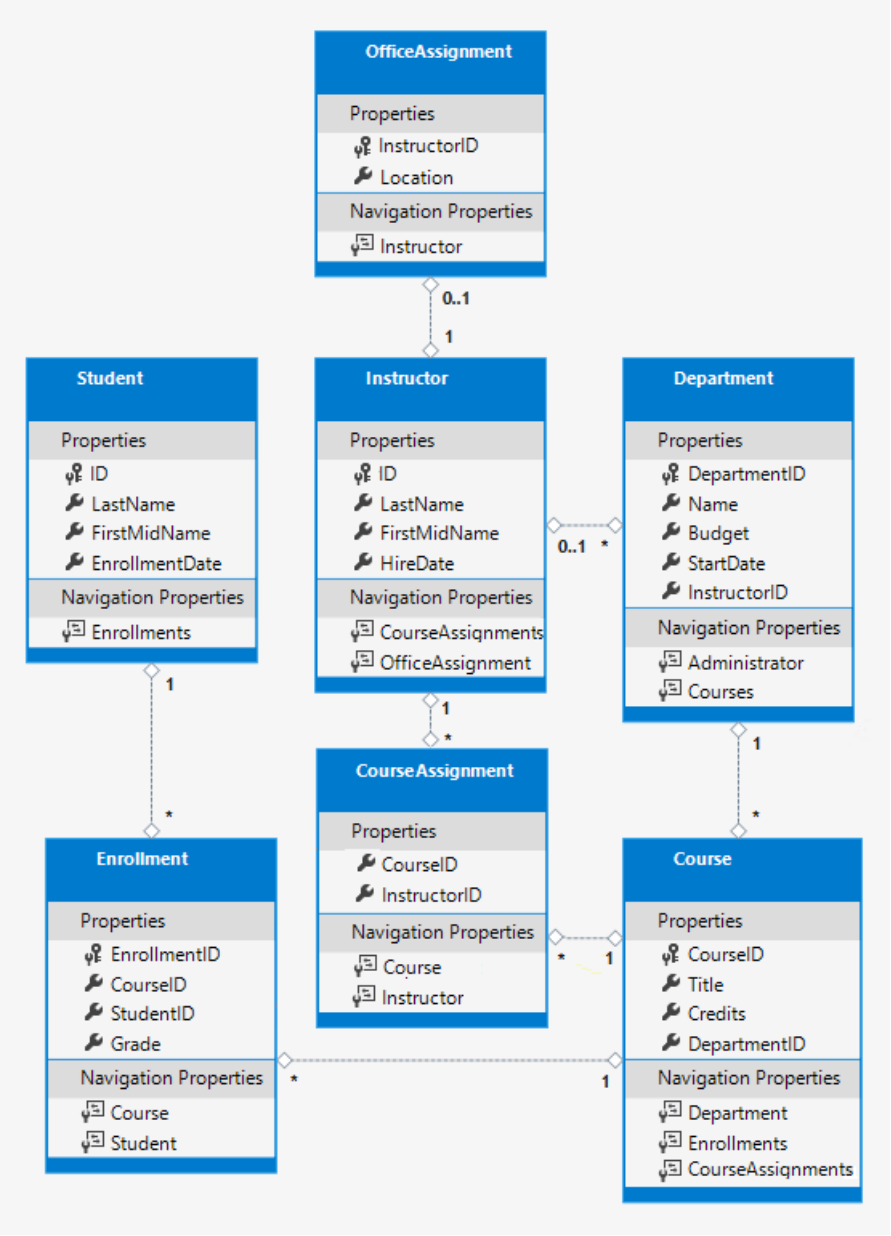
The next two tutorials show how to read and update related data.

PREVIOUS TUTORIAL NEXT TUTORIAL

The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The entity classes for the completed data model are shown in the following illustration:



If you run into problems you can't solve, download the completed app.

Customize the data model with attributes

In this section, the data model is customized using attributes.

The DataType attribute

The student pages currently displays the time of the enrollment date. Typically, date fields show only the date and not the time.

Update *Models/Student.cs* with the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers consume. The `DataType` attributes don't provide validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

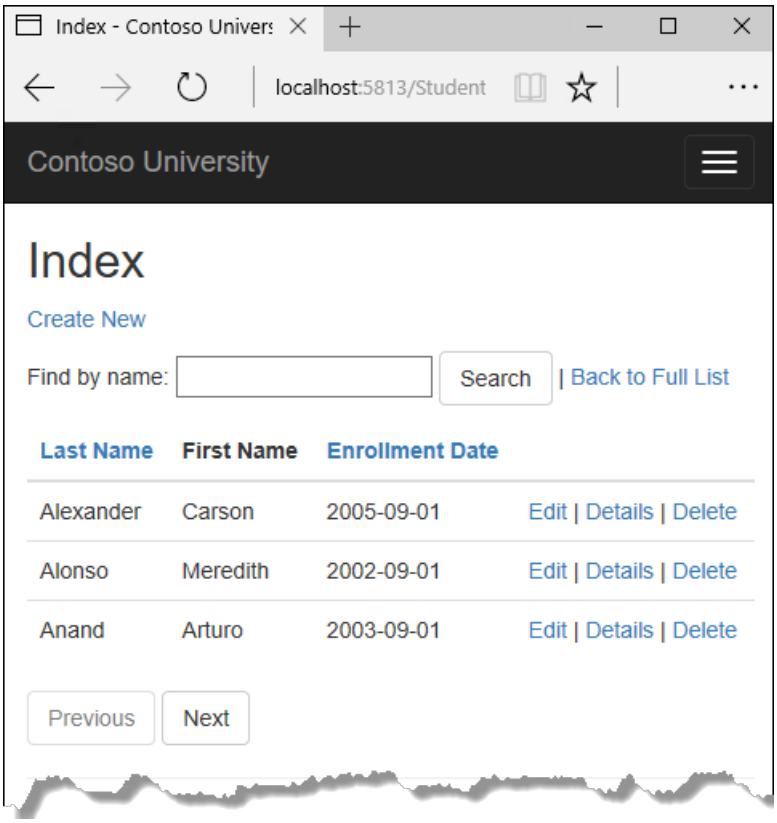
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, client-side input validation, etc.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

Run the app. Navigate to the Students Index page. Times are no longer displayed. Every view that uses the `Student` model displays the date without time.



The StringLength attribute

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

Update the `Student` model with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

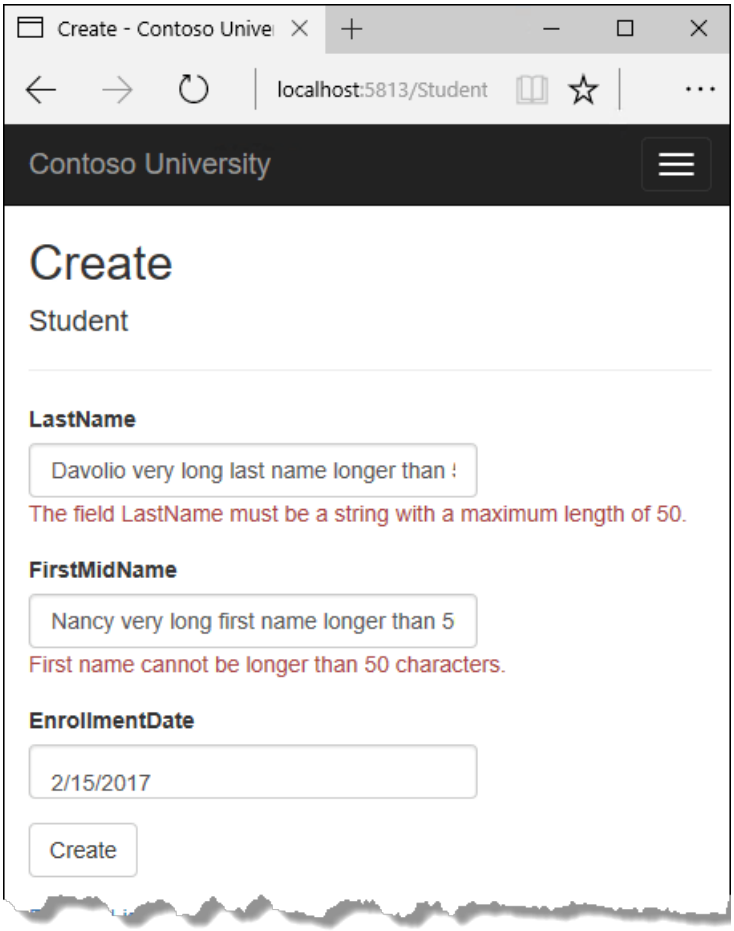
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code limits names to no more than 50 characters. The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute is used to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

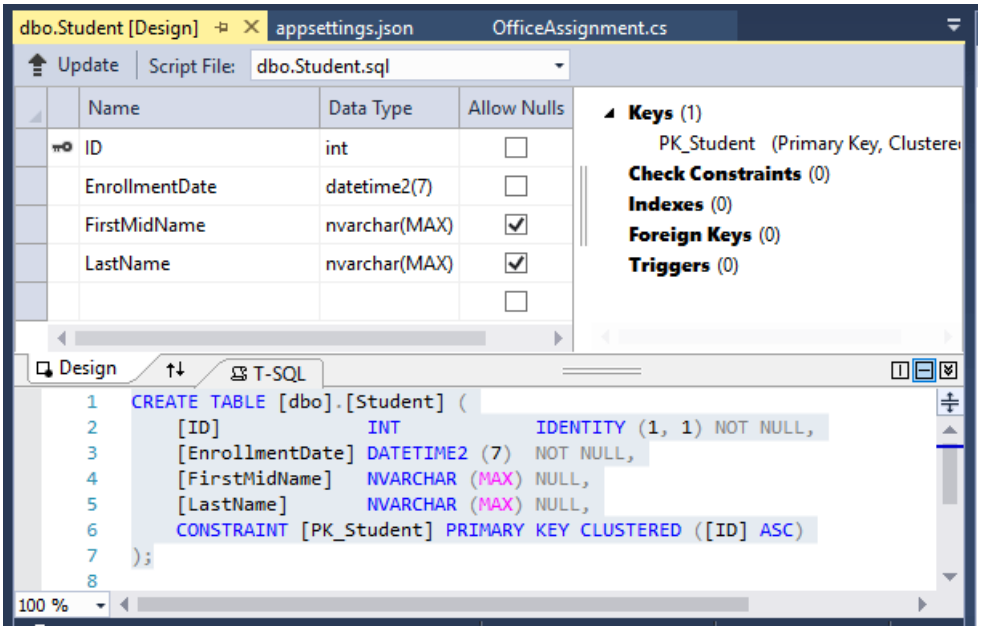
```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

Run the app:

- Navigate to the Students page.
- Select **Create New**, and enter a name longer than 50 characters.
- Select **Create**, client-side validation shows an error message.



In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)` because migrations has not been run on the DB. When migrations are run later in this tutorial, the name fields

become `nvarchar(50)`.

The Column attribute

Attributes can control how classes and properties are mapped to the database. In this section, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the DB.

When the DB is created, property names on the model are used for column names (except when the `Column` attribute is used).

The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

Update the *Student.cs* file with the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

With the preceding change, `Student.FirstMidName` in the app maps to the `FirstName` column of the `Student` table.

The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. If the app is run before applying migrations, the following exception is generated:

```
SqlException: Invalid column name 'FirstName'.
```

To update the DB:

- Build the project.
- Open a command window in the project folder. Enter the following commands to create a new migration and update the DB:

- Visual Studio
- Visual Studio Code

```
Add-Migration ColumnFirstName
Update-Database
```

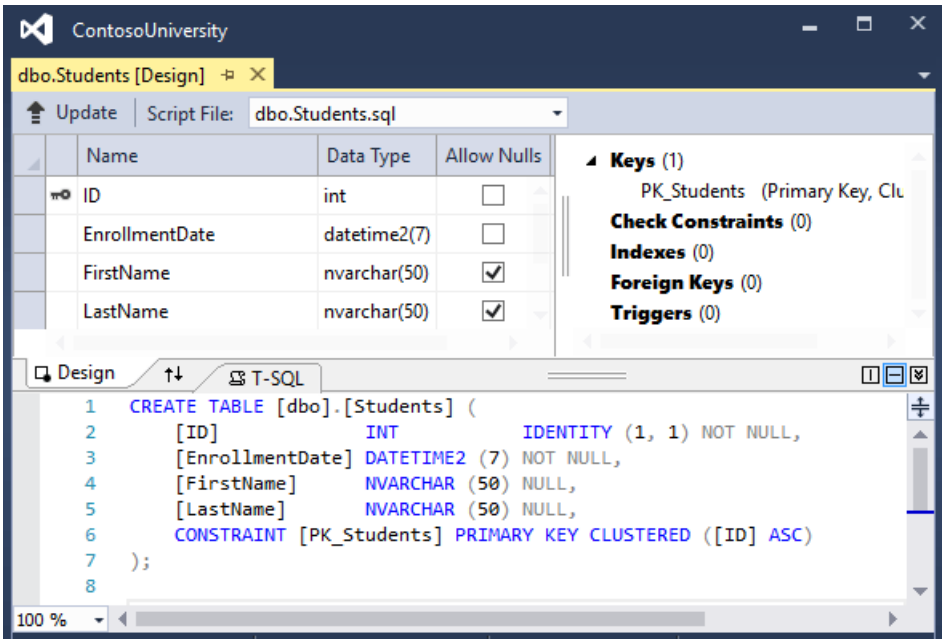
The `migrations add ColumnFirstName` command generates the following warning message:

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
```

The warning is generated because the name fields are now limited to 50 characters. If a name in the DB had more than 50 characters, the 51 to last character would be lost.

- Test the app.

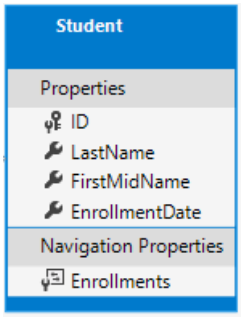
Open the Student table in SSOX:



Before migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

> **NOTE**
> In the following section, building the app at some stages generates compiler errors. The instructions specify when to build the app.

Student entity update



Update *Models/Student.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (`DateTime` , `int` , `double` , etc.). Types that can't be null are automatically treated as required fields.

The `Required` attribute could be replaced with a minimum length parameter in the `StringLength` attribute:

```
[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }
```

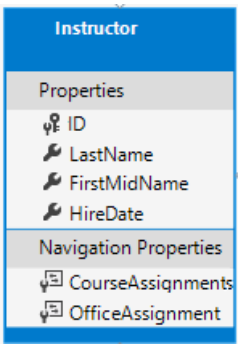
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` cannot be set, it has only a get accessor. No `FullName` column is created in the database.

Create the Instructor Entity



Create *Models/Instructor.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

If a navigation property holds multiple entities:

- It must be a list type where the entries can be added, deleted, and updated.

Navigation property types include:

- `ICollection<T>`
- `List<T>`
- `HashSet<T>`

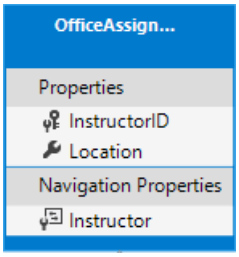
If `ICollection<T>` is specified, EF Core creates a `HashSet<T>` collection by default.

The `CourseAssignment` entity is explained in the section on many-to-many relationships.

Contoso University business rules state that an instructor can have at most one office. The `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment entity



Create *Models/OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The `[Key]` attribute is used to identify a property as the primary key (PK) when the property name is something other than classnameID or ID.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key

(FK) to the `Instructor` entity. EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because:

- `InstructorID` doesn't follow the ID or classnameID naming convention.

Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The `OfficeAssignment` navigation property for the `Instructor` entity is nullable because:

- Reference types (such as classes are nullable).
- An instructor might not have an office assignment.

The `OfficeAssignment` entity has a non-nullable `Instructor` navigation property because:

- `InstructorID` is non-nullable.
- An office assignment can't exist without an instructor.

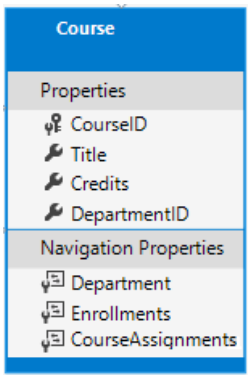
When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The `[Required]` attribute could be applied to the `Instructor` navigation property:

```
[Required]
public Instructor Instructor { get; set; }
```

The preceding code specifies that there must be a related instructor. The preceding code is unnecessary because the `InstructorID` foreign key (which is also the PK) is non-nullable.

Modify the Course Entity



Update *Models/Course.cs* with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a FK property for a data model when the model has a navigation property for a related entity.

EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. Having the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is *not* included. When a course entity is fetched to edit:

- The `Department` entity is null if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, EF Core assumes that PK values are generated by the DB. DB generated PK values is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the DB can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

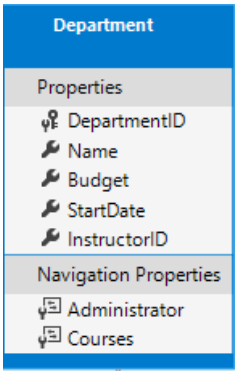
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection:

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

`CourseAssignment` is explained later.

Create the Department entity



Create *Models/Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the DB:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required. EF Core generally chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

The question mark (?) in the preceding code specifies the property is nullable.

A department may have many courses, so there's a Courses navigation property:

```
public ICollection<Course> Courses { get; set; }
```

Note: By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. Cascading delete can result in circular cascade delete rules. Circular cascade delete rules causes an exception when a migration is added.

For example, if the `Department.InstructorID` property was defined as non-nullable:

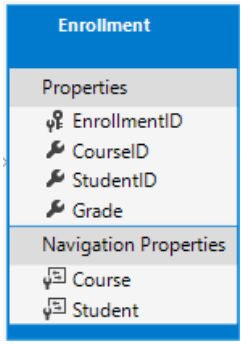
- EF Core configures a cascade delete rule to delete the department when the instructor is deleted.
- Deleting the department when the instructor is deleted isn't the intended behavior.
- The following fluent API would set a restrict rule instead of cascade.

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

The preceding code disables cascade delete on the department-instructor relationship.

Update the Enrollment entity

An enrollment record is for one course taken by one student.



Update *Models/Enrollment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

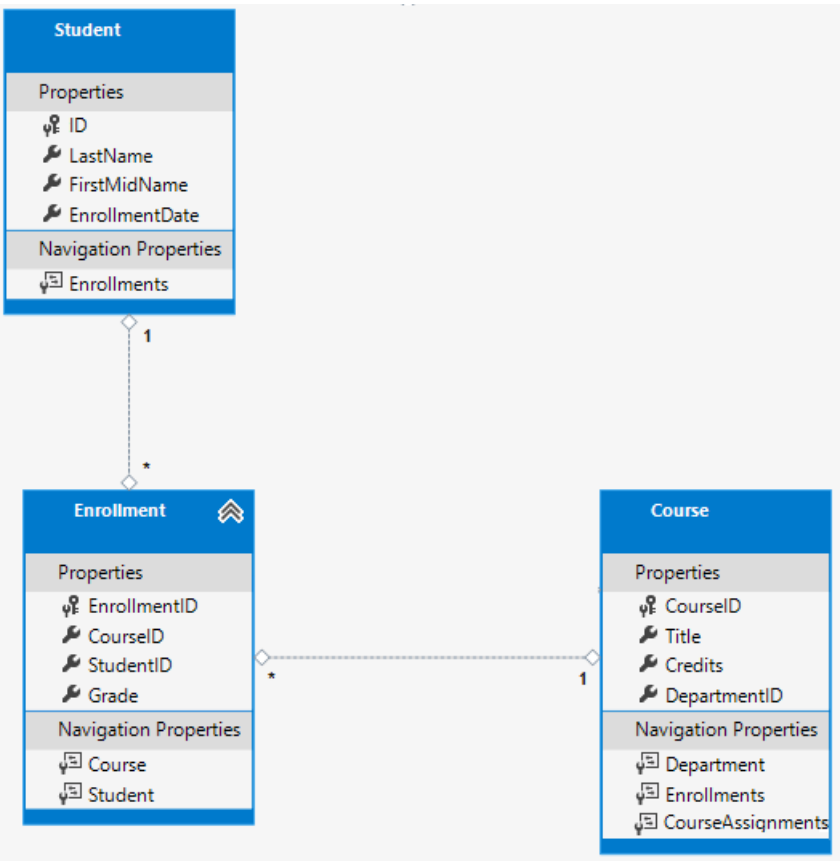
An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities. The `Enrollment` entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the `Enrollment` table contains additional data besides FKs for the joined tables (in this case, the PK and `Grade`).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



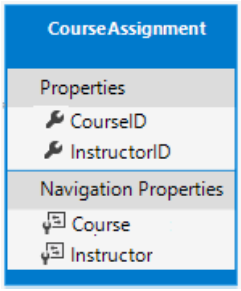
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs (`CourseID` and `StudentID`). A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a pure join table.

Note: EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see Many-to-many relationships in EF Core 2.0.

The CourseAssignment entity

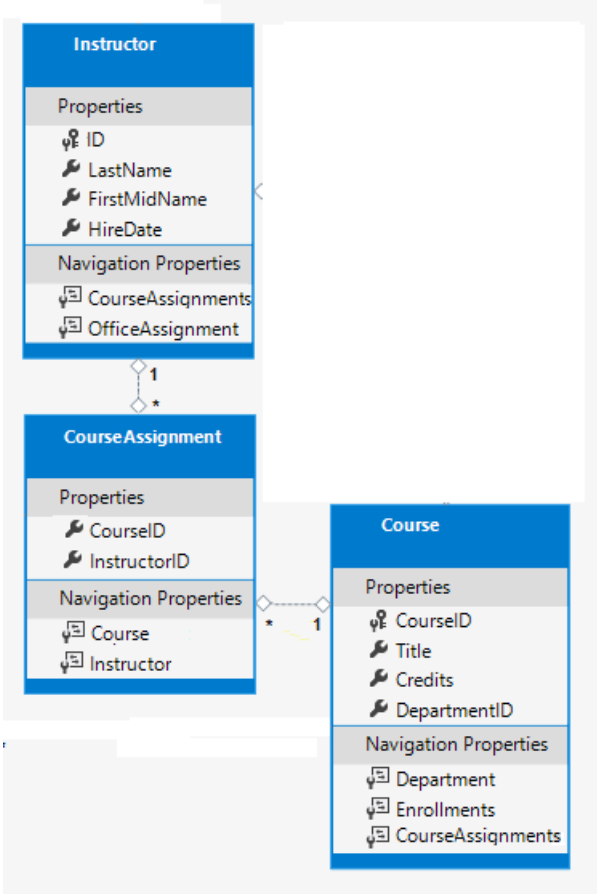


Create *Models/CourseAssignment.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Instructor-to-Courses



The Instructor-to-Courses many-to-many relationship:

- Requires a join table that must be represented by an entity set.
- Is a pure join table (table without payload).

It's common to name a join entity `EntityName1EntityName2`. For example, the Instructor-to-Courses join table using this pattern is `CourseInstructor`. However, we recommend using a name that describes the relationship.

Data models start out simple and grow. No-payload joins (PJTs) frequently evolve to include payload. By starting with a descriptive entity name, the name doesn't need to change when the join table changes. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked with a join entity called Ratings. For the Instructor-to-Courses many-to-many relationship, `CourseAssignment` is preferred over `CourseInstructor`.

Composite key

FKs are not nullable. The two FKs in `CourseAssignment` (`InstructorID` and `CourseID`) together uniquely identify each row of the `CourseAssignment` table. `CourseAssignment` doesn't require a dedicated PK. The `InstructorID` and `CourseID` properties function as a composite PK. The only way to specify composite PKs to EF Core is with the *fluent API*. The next section shows how to configure the composite PK.

The composite key ensures:

- Multiple rows are allowed for one course.
- Multiple rows are allowed for one instructor.
- Multiple rows for the same instructor and course isn't allowed.

The `Enrollment` join entity defines its own PK, so duplicates of this sort are possible. To prevent such duplicates:

- Add a unique index on the FK fields, or
- Configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the DB context

Add the following highlighted code to *Data/SchoolContext.cs*:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Models
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollment { get; set; }
        public DbSet<Student> Student { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

The preceding code adds the new entities and configures the `CourseAssignment` entity's composite PK.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is

called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

In this tutorial, the fluent API is used only for DB mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API (specifying a composite PK). There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

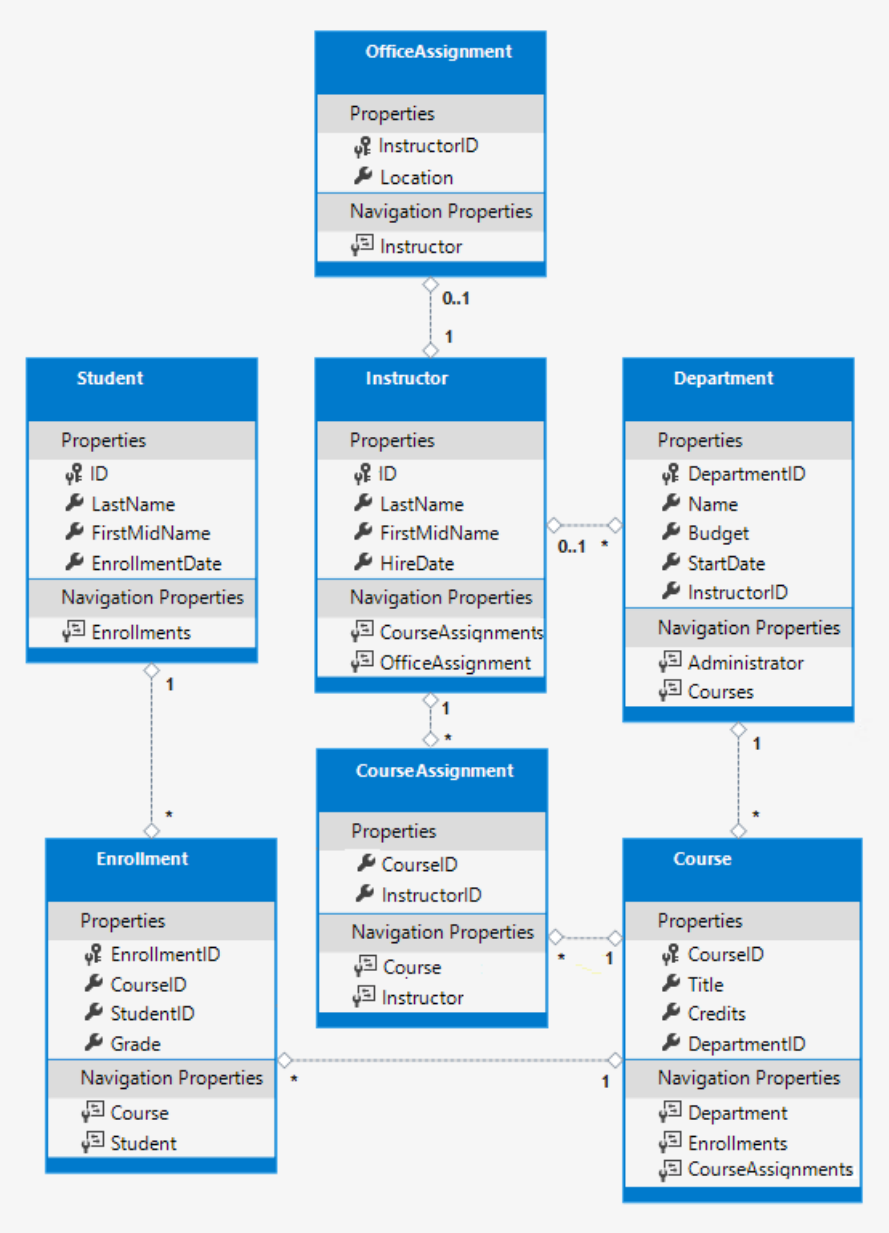
Some of the attributes used in the this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

The following illustration shows the diagram that EF Power Tools create for the completed School model.



The preceding diagram shows:

- Several one-to-many relationship lines (1 to *).
- The one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment` entities.
- The zero-or-one-to-many relationship line (0..1 to *) between the `Instructor` and `Department` entities.

Seed the DB with Test Data

Update the code in *Data/DbInitializer.cs*:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();
```

```
            // Look for any students.
            if (context.Student.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

            foreach (Student s in students)
            {
                context.Student.Add(s);
            }
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            foreach (Instructor i in instructors)
            {
                context.Instructors.Add(i);
            }
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
            };

            foreach (Department d in departments)
```

```
{
    context.Departments.Add(d);
}
context.SaveChanges();

var courses = new Course[]
{
    new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
    },
    new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
    },
    new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
    },
    new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
    new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
        DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
    },
};

foreach (Course c in courses)
{
    context.Courses.Add(c);
}
context.SaveChanges();

var officeAssignments = new OfficeAssignment[]
{
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
        Location = "Smith 17" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
        Location = "Gowan 27" },
    new OfficeAssignment {
        InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
        Location = "Thompson 304" },
};

foreach (OfficeAssignment o in officeAssignments)
{
    context.OfficeAssignments.Add(o);
}
context.SaveChanges();

var courseInstructors = new CourseAssignment[]
{
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
        },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
        },
```

```
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
            },
    };

    foreach (CourseAssignment ci in courseInstructors)
    {
        context.CourseAssignments.Add(ci);
    }
    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            Grade = Grade.A
        },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
            Grade = Grade.C
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alexander").ID,
            CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Alonso").ID,
            CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
            Grade = Grade.B
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
            },
            new Enrollment {
            StudentID = students.Single(s => s.LastName == "Anand").ID,
            CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
            Grade = Grade.B
            },
```

```
                    },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").ID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollment.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollment.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing. See `Enrollments` and `CourseAssignments` for examples of how many-to-many join tables can be seeded.

Add a migration

Build the project.

- Visual Studio
- Visual Studio Code

```
Add-Migration ComplexDataModel
```

The preceding command displays a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data.
Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If the `database update` command is run, the following error is produced:

```
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint
"FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in
database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.
```

Apply the migration

Now that you have an existing database, you need to think about how to apply future changes to it. This tutorial shows two approaches:

- Drop and re-create the database
- Apply the migration to the existing database. While this method is more complex and time-consuming, it's the preferred approach for real-world, production environments. **Note**: This is an optional section of the tutorial. You can do the drop and re-create steps and skip this section. If you do want to follow the steps in this section, don't do the drop and re-create steps.

Drop and re-create the database

The code in the updated `DbInitializer` adds seed data for the new entities. To force EF Core to create a new DB, drop and update the DB:

- Visual Studio
- Visual Studio Code

In the **Package Manager Console** (PMC), run the following command:

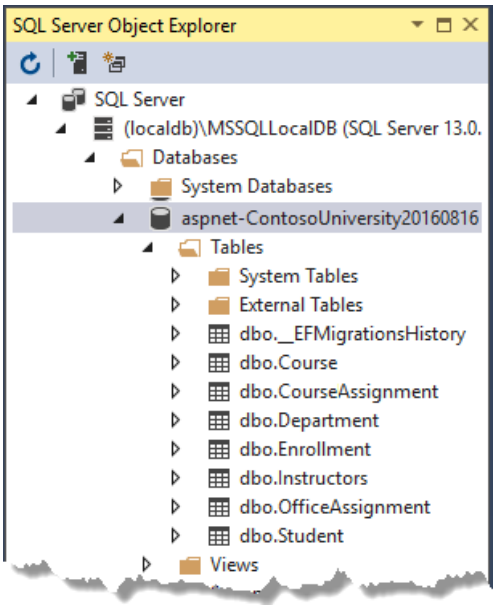
```
Drop-Database
Update-Database
```

Run `Get-Help about_EntityFrameworkCore` from the PMC to get help information.

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new DB.

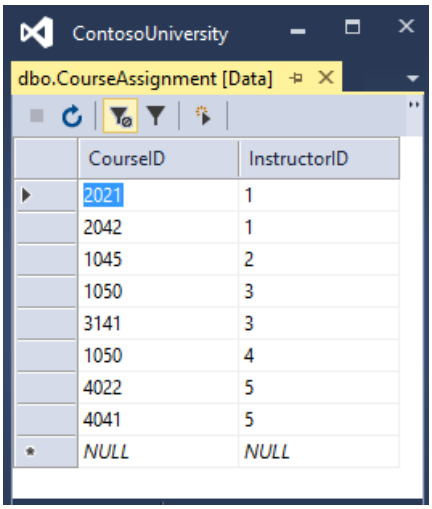
Open the DB in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.



Examine the **CourseAssignment** table:

- Right-click the **CourseAssignment** table and select **View Data**.
- Verify the **CourseAssignment** table contains data.



Apply the migration to the existing database

This section is optional. These steps work only if you skipped the preceding Drop and re-create the database section.

When migrations are run with existing data, there may be FK constraints that are not satisfied with the existing data. With production data, steps must be taken to migrate the existing data. This section provides an example of fixing FK constraint violations. Don't make these code changes without a backup. Don't make these code changes if you completed the previous section and updated the database.

The *{timestamp}_ComplexDataModel.cs* file contains the following code:

```
migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    type: "int",
    nullable: false,
    defaultValue: 0);
```

The preceding code adds a non-nullable `DepartmentID` FK to the `Course` table. The DB from the previous tutorial contains rows in `Course`, so that table cannot be updated by migrations.

To make the `ComplexDataModel` migration work with existing data:

- Change the code to give the new column (`DepartmentID`) a default value.
- Create a fake department named "Temp" to act as the default department.

Fix the foreign key constraints

Update the `ComplexDataModel` classes `Up` method:

- Open the *{timestamp}_ComplexDataModel.cs* file.
- Comment out the line of code that adds the `DepartmentID` column to the `Course` table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

Add the following highlighted code. The new code goes after the `.CreateTable( name: "Department"` block:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(type: "int", nullable: true),
        Name = table.Column<string>(type: "nvarchar(50)", maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(type: "datetime2", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

 migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

With the preceding changes, existing `Course` rows will be related to the "Temp" department after the `ComplexDataModel` `Up` method runs.

A production app would:

- Include code or scripts to add `Department` rows and related `Course` rows to the new `Department` rows.
- Not use the "Temp" department or the default value for `Course.DepartmentID`.

The next tutorial covers related data.

Additional resources

- YouTube version of this tutorial(Part 1)
- YouTube version of this tutorial(Part 2)

PREVIOUS NEXT

Part 6, Razor Pages with EF Core in ASP.NET Core - Read Related Data

9/22/2020 • 28 minutes to read • Edit Online

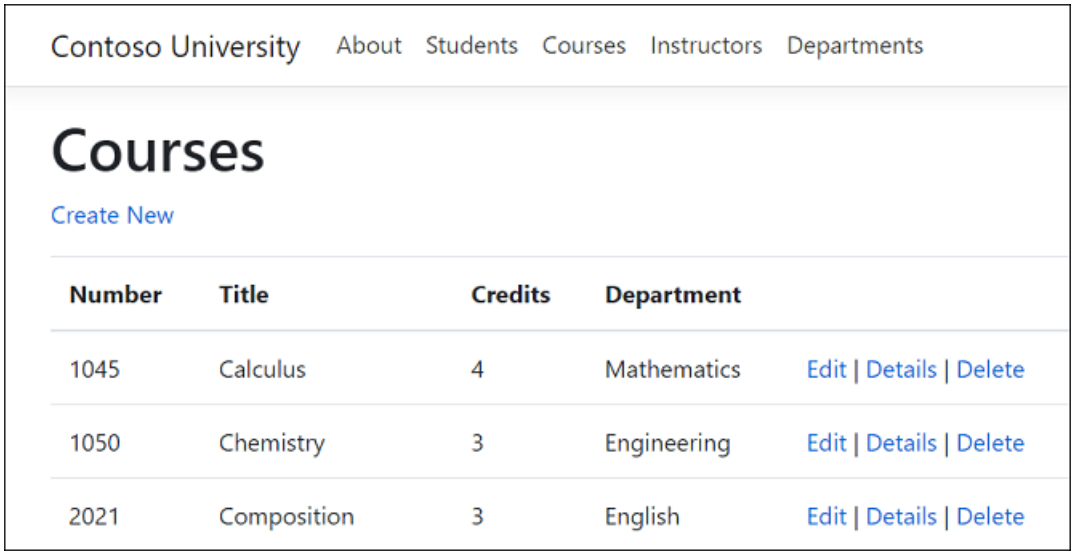
By Tom Dykstra, Jon P Smith, and Rick Anderson

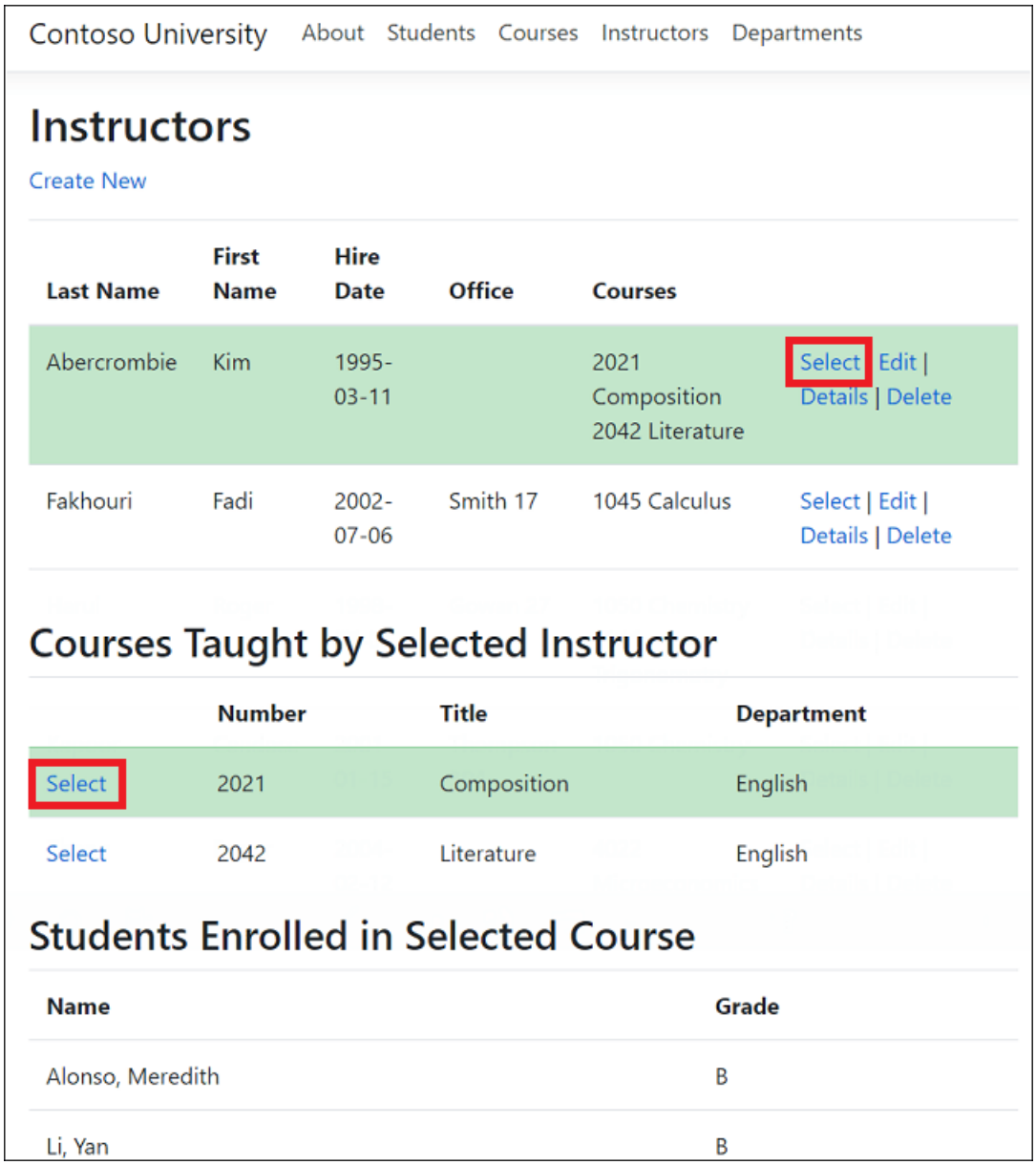
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to read and display related data. Related data is data that EF Core loads into navigation properties.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy loading

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When an entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than a giant single query. Eager loading is specified with the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

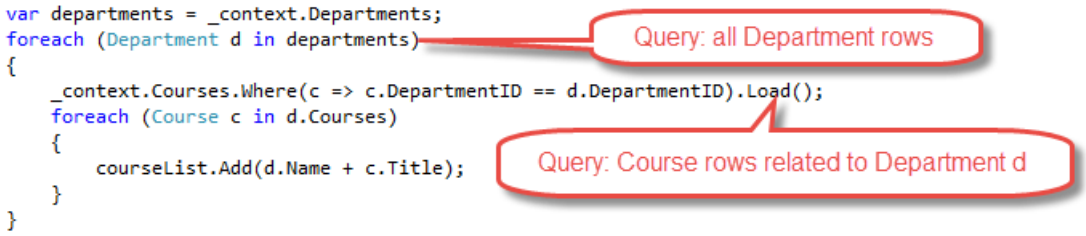
Query: all Department entities and related Course entities

  Eager loading sends multiple queries when a collection navigation is included:

  - One query for the main query
  - One query for each collection "edge" in the load tree.

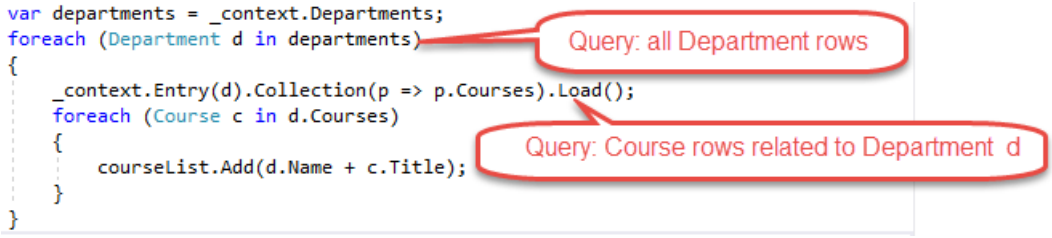
- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "Fixes up" means that EF Core automatically populates the navigation properties.

Separate queries with `Load` is more like explicit loading than eager loading.



Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

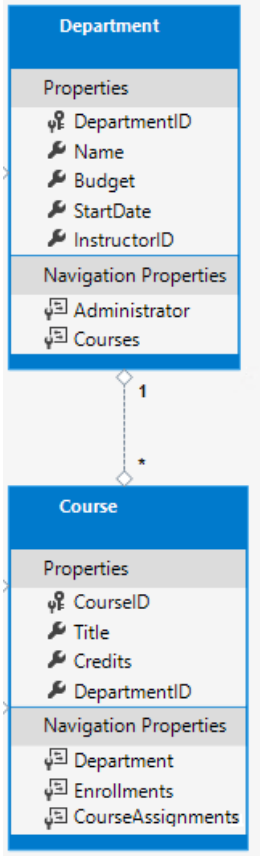
- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the database. With explicit loading, the code specifies the navigation properties to be loaded. Use the `Load` method to do explicit loading. For example:



- Lazy loading. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the database each time a navigation property is accessed for the first time. Lazy loading can hurt performance, for example when developers use N+1 patterns, loading a parent and enumerating through children.

Create Course pages

The `Course` entity includes a navigation property that contains the related `Department` entity.



To display the name of the assigned department for a course:

- Load the related `Department` entity into the `Course.Department` navigation property.
- Get the name from the `Department` entity's `Name` property.

Scaffold Course pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold Student pages with the following exceptions:
  - Create a *Pages/Courses* folder.
  - Use `Course` for the model class.
  - Use the existing context class instead of creating a new one.

- Open *Pages/Courses/Index.cshtml.cs* and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

- Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Display the department name

Update Pages/Courses/Index.cshtml.cs with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IList<Course> Courses { get; set; }

        public async Task OnGetAsync()
        {
            Courses = await _context.Courses
                .Include(c => c.Department)
                .AsNoTracking()
                .ToListAsync();
        }
    }
}
```

The preceding code changes the `Course` property to `Courses` and adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities don't need to be tracked because they're not updated in the current context.

Update *Pages/Courses/Index.cshtml* with the following code.

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel

@{
    ViewData["Title"] = "Courses";
}

<h1>Courses</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Courses)
{
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.CourseID)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Credits)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Department.Name)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

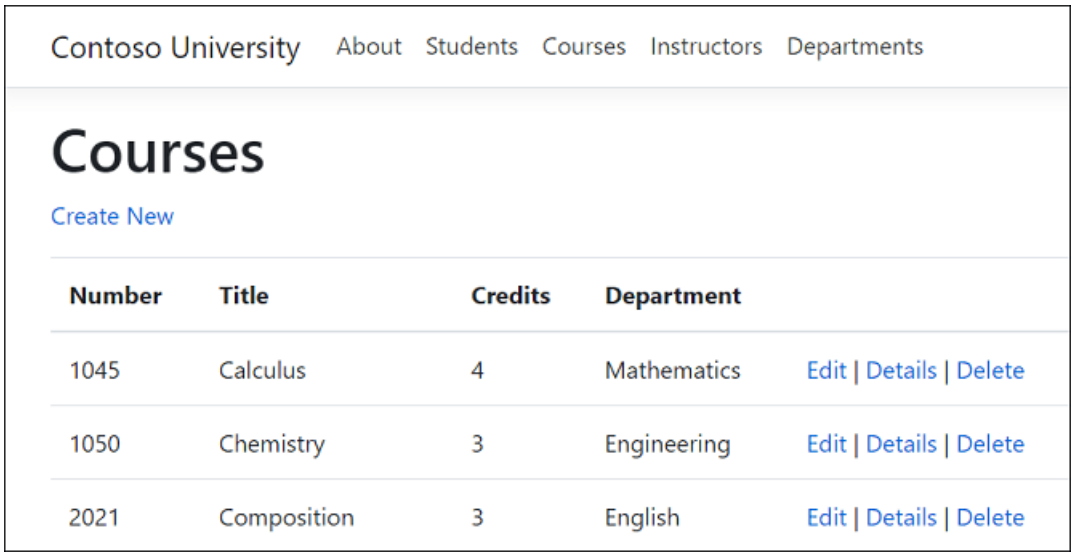
The following changes have been made to the scaffolded code:

- Changed the `Course` property name to `Courses`.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- Changed the **Department** column to display the department name. The code displays the `Name` property

of the `Department` entity that's loaded into the `Department` navigation property:

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method. The `Select` method is an alternative that loads only the related data needed. For single items, like the `Department.Name` it uses a SQL INNER JOIN. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
            .Select(p => new CourseViewModel
            {
                CourseID = p.CourseID,
                Title = p.Title,
                Credits = p.Credits,
                DepartmentName = p.Department.Name
            }).ToListAsync();
}
```

The preceding code doesn't return any entity types, therefore no tracking is done. For more information about the EF tracking, see Tracking vs. No-Tracking Queries.

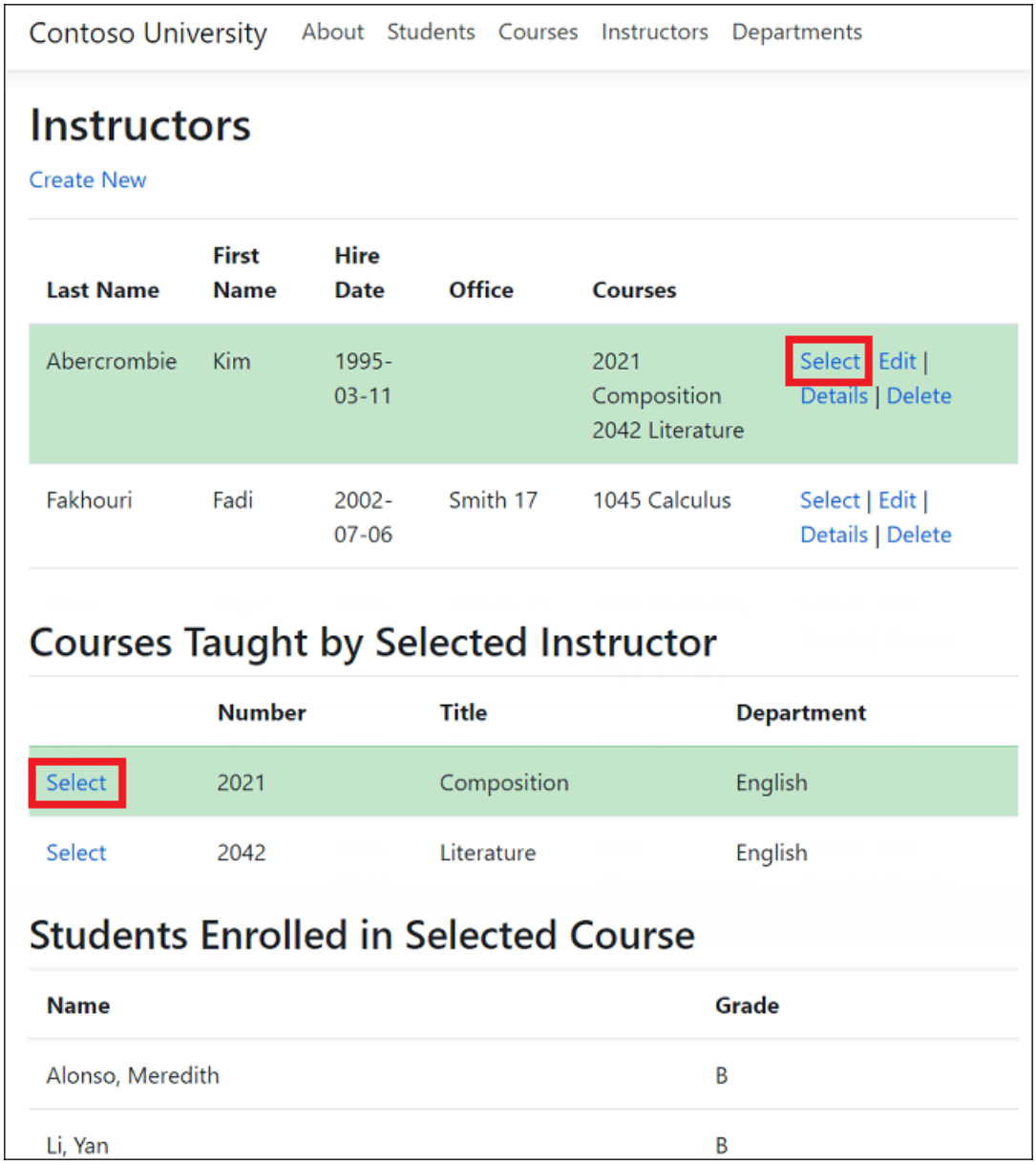
The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelect.cshtml and IndexSelect.cshtml.cs for a complete example.

Create Instructor pages

This section scaffolds Instructor pages and adds related Courses and Enrollments to the Instructors Index page.



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model

The instructors page shows data from three different tables. A view model is needed that includes three properties representing the three tables.

Create *SchoolViewModels/InstructorIndexData.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Scaffold Instructor pages

- Visual Studio
- Visual Studio Code

- Follow the instructions in Scaffold the student pages with the following exceptions:
    - Create a *Pages/Instructors* folder.
    - Use `Instructor` for the model class.
    - Use the existing context class instead of creating a new one.

To see what the scaffolded page looks like before you update it, run the app and navigate to the Instructors page.

Update *Pages/Instructors/Index.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Department)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Enrollments)
                            .ThenInclude(i => i.Student)
                .AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
                Instructor instructor = InstructorData.Instructors
                    .Where(i => i.ID == id.Value).Single();
                InstructorData.Courses = instructor.CourseAssignments.Select(s => s.Course);
            }

            if (courseID != null)
            {
                CourseID = courseID.Value;
                var selectedCourse = InstructorData.Courses
                    .Where(x => x.CourseID == courseID).Single();
                InstructorData.Enrollments = selectedCourse.Enrollments;
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the *Pages/Instructors/Index.cshtml.cs* file:

```
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The code specifies eager loading for the following navigation properties:

- `Instructor.OfficeAssignment`
- `Instructor.CourseAssignments`
  - `CourseAssignments.Course`
    - `Course.Department`
    - `Course.Enrollments`
      - `Enrollment.Student`

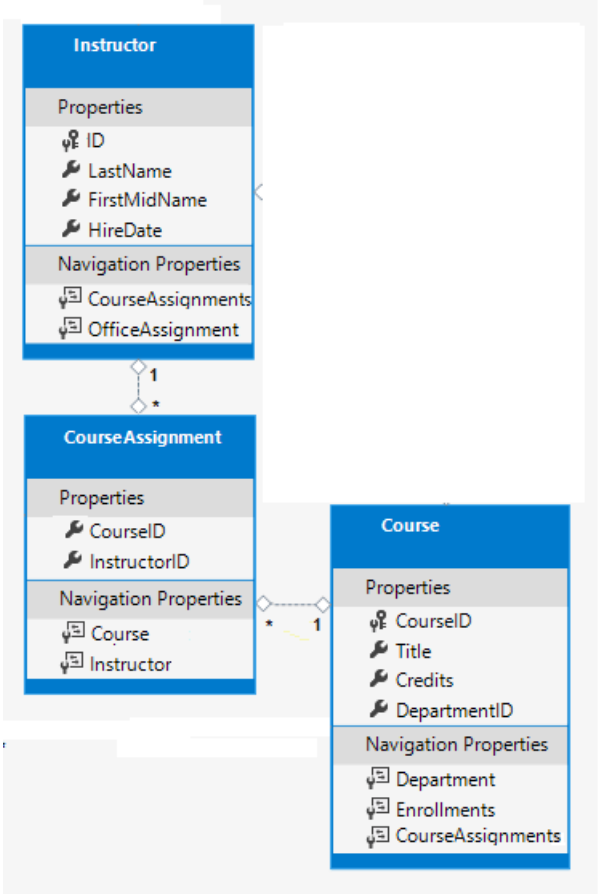
Notice the repetition of `Include` and `ThenInclude` methods for `CourseAssignments` and `Course`. This repetition is necessary to specify eager loading for two navigation properties of the `Course` entity.

The following code executes when an instructor is selected (`id != null`).

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = InstructorData.Instructors
        .Where(i => i.ID == id.Value).Single();
    InstructorData.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

The `Where` method returns a collection. But in this case, the filter will select a single entity, so the `Single` method is called to convert the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `CourseAssignments` property. `CourseAssignments` provides access to the related `Course` entities.



The `Single` method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    var selectedCourse = InstructorData.Courses
        .Where(x => x.CourseID == courseID).Single();
    InstructorData.Enrollments = selectedCourse.Enrollments;
}
```

Update the instructors Index page

Update *Pages/Instructors/Index.cshtml* with the following code.

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
```

```
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.InstructorData.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.InstructorData.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.InstructorData.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
```

```
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}

@if (Model.InstructorData.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.InstructorData.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `https://localhost:5001/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the URL is:

  `https://localhost:5001/Instructors/2`

- Adds an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

  ```
  @if (item.OfficeAssignment != null)
  {
      @item.OfficeAssignment.Location
  }
  ```

- Adds a **Courses** column that displays courses taught by each instructor. See Explicit line transition for

more about this razor syntax.

- Adds code that dynamically adds `class="success"` to the `tr` element of the selected instructor and course. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.CourseID == Model.CourseID)
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Adds a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

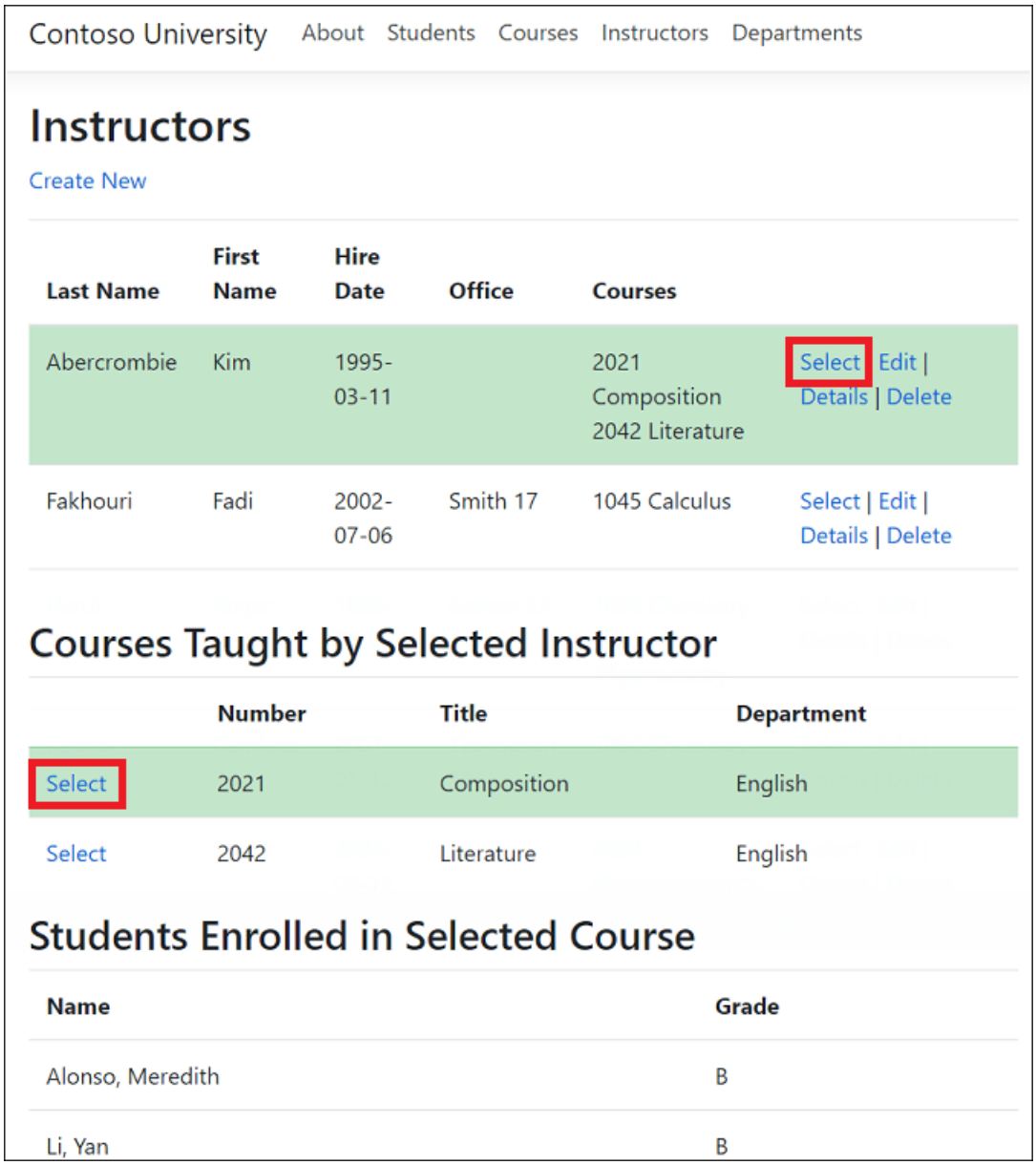
```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

- Adds a table of courses for the selected Instructor.
- Adds a table of student enrollments for the selected course.

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If `OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link for an instructor. The row style changes and courses assigned to that instructor are displayed.

Select a course to see the list of enrolled students and their grades.



Using Single

The `Single` method can pass in the `Where` condition instead of calling the `Where` method separately:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    InstructorData = new InstructorIndexData();

    InstructorData.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Enrollments)
                        .ThenInclude(i => i.Student)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = InstructorData.Instructors.Single(
            i => i.ID == id.Value);
        InstructorData.Courses = instructor.CourseAssignments.Select(
            s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        InstructorData.Enrollments = InstructorData.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}
```

Use of `Single` with a Where condition is a matter of personal preference. It provides no benefits over using the `Where` method.

Explicit loading

The current code specifies eager loading for `Enrollments` and `Students`:

```
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

Suppose users rarely want to see enrollments in a course. In that case, an optimization would be to only load the enrollment data if it's requested. In this section, the `OnGetAsync` is updated to use explicit loading of `Enrollments` and `Students`.

Update *Pages/Instructors/Index.cshtml.cs* with the following code.

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                        .ThenInclude(i => i.Department)
                //.Include(i => i.CourseAssignments)
                //    .ThenInclude(i => i.Course)
                //        .ThenInclude(i => i.Enrollments)
                //            .ThenInclude(i => i.Student)
                //.AsNoTracking()
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
                Instructor instructor = InstructorData.Instructors
                    .Where(i => i.ID == id.Value).Single();
                InstructorData.Courses = instructor.CourseAssignments.Select(s => s.Course);
            }

            if (courseID != null)
            {
                CourseID = courseID.Value;
                var selectedCourse = InstructorData.Courses
                    .Where(x => x.CourseID == courseID).Single();
                await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
                foreach (Enrollment enrollment in selectedCourse.Enrollments)
                {
                    await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
                }
                InstructorData.Enrollments = selectedCourse.Enrollments;
            }
        }
    }
}
```

The preceding code drops the *ThenInclude* method calls for enrollment and student data. If a course is selected, the explicit loading code retrieves:

- The `Enrollment` entities for the selected course.

- The `Student` entities for each `Enrollment`.

Notice that the preceding code comments out `.AsNoTracking()`. Navigation properties can only be explicitly loaded for tracked entities.

Test the app. From a user's perspective, the app behaves identically to the previous version.

Next steps

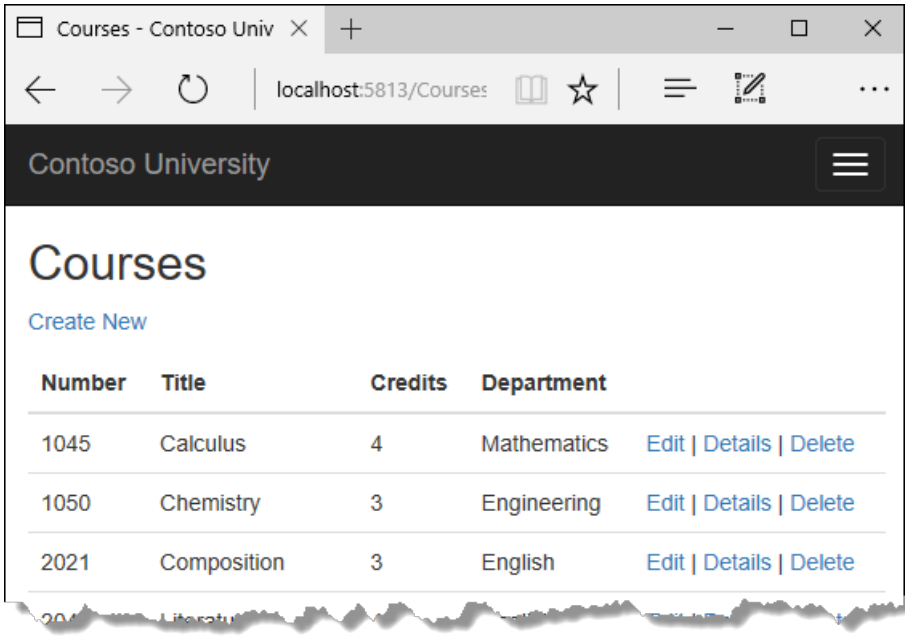
The next tutorial shows how to update related data.

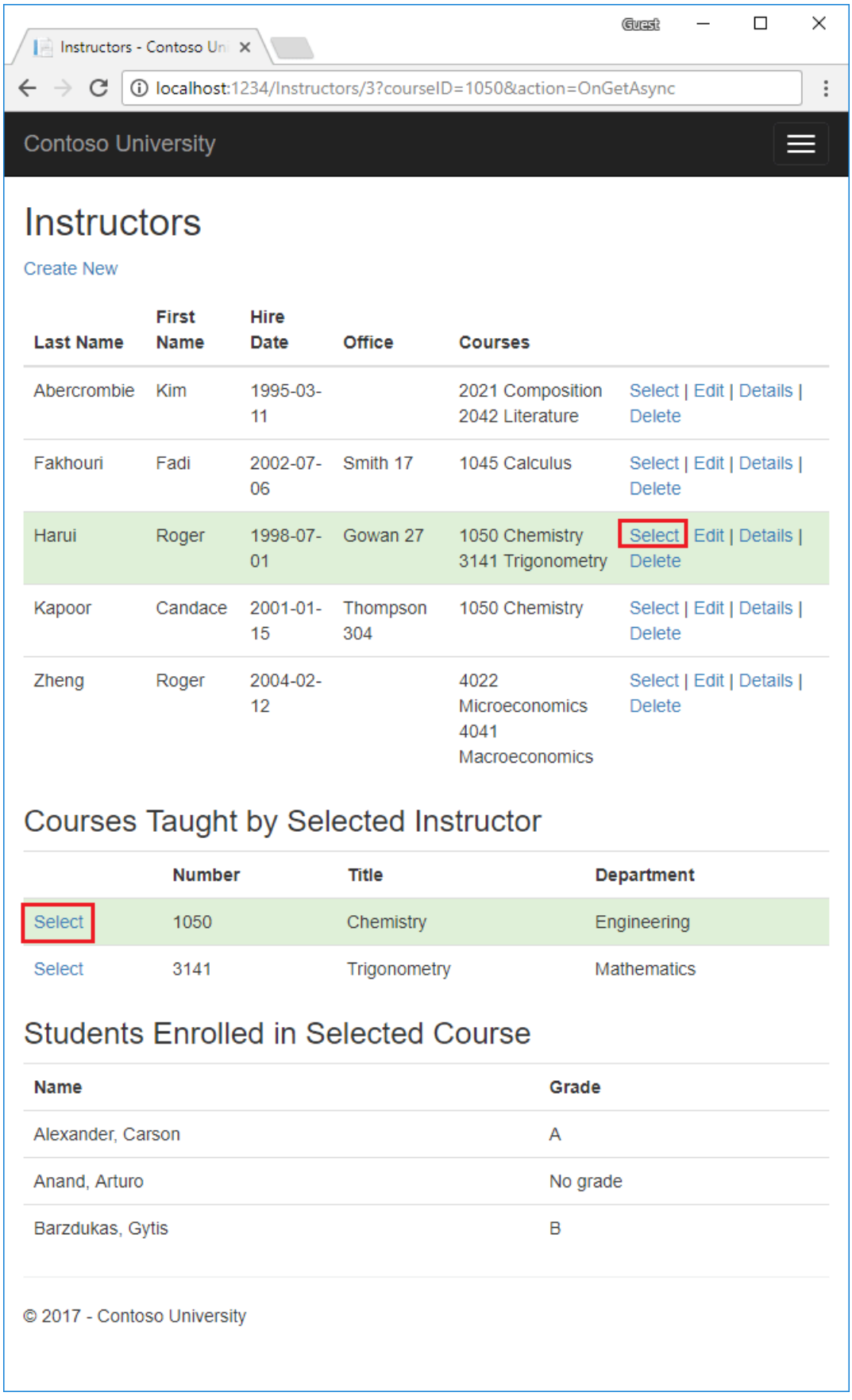
PREVIOUS TUTORIAL NEXT TUTORIAL

In this tutorial, related data is read and displayed. Related data is data that EF Core loads into navigation properties.

If you run into problems you can't solve, download or view the completed app. Download instructions.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy Loading of related data

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When the entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than was the case for some queries in EF6 where there was a single

query. Eager loading is specified with the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

Eager loading sends multiple queries when a collection navigation is included:

- One query for the main query
- One query for each collection "edge" in the load tree.

- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "fixes up" means that EF Core automatically populates the navigation properties. Separate queries with `Load` is more like explicit loading than eager loading.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the DB. With explicit loading, the code specifies the navigation properties to be loaded. Use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Lazy loading. Lazy loading was added to EF Core in version 2.1. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the DB each time a navigation property is accessed for the first time.

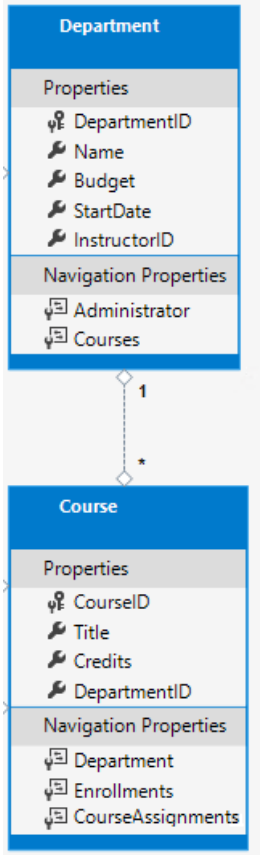
- The `Select` operator loads only the related data needed.

Create a Course page that displays department name

The Course entity includes a navigation property that contains the `Department` entity. The `Department` entity contains the department that the course is assigned to.

To display the name of the assigned department in a list of courses:

- Get the `Name` property from the `Department` entity.
- The `Department` entity comes from the `Course.Department` navigation property.



Scaffold the Course model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Course` for the model class.

The preceding command scaffolds the `Course` model. Open the project in Visual Studio.

Open *Pages/Courses/Index.cshtml.cs* and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Update the `OnGetAsync` method with the following code:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The preceding code adds `AsNoTracking`. `AsNoTracking` improves performance because the entities returned are not tracked. The entities are not tracked because they're not updated in the current context.

Update *Pages/Courses/Index.cshtml* with the following highlighted markup:

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel
@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

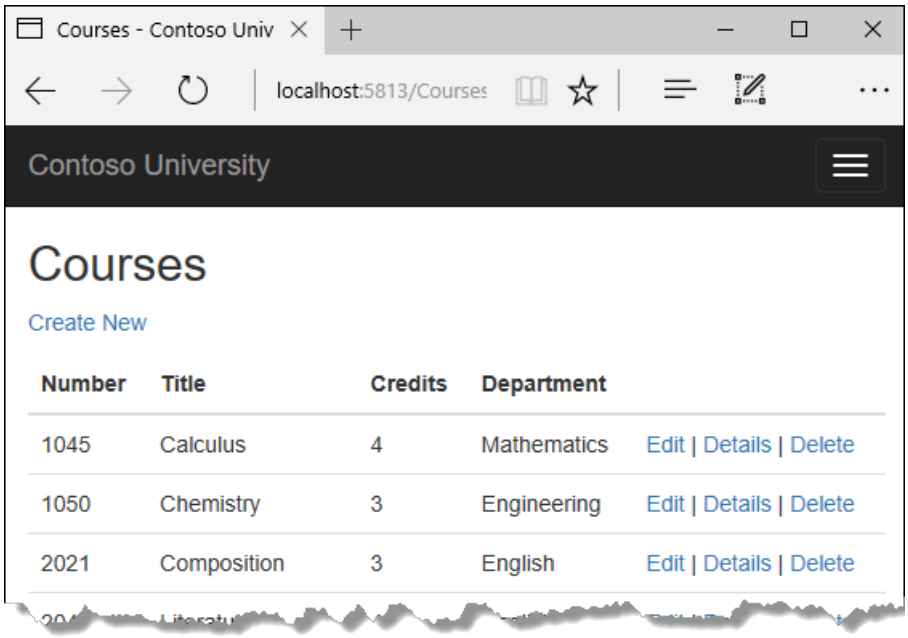
<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Course[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Course)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The following changes have been made to the scaffolded code:

- Changed the heading from Index to Courses.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method:

```
public async Task OnGetAsync()
{
    Course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .ToListAsync();
}
```

The `Select` operator loads only the related data needed. For single items, like the `Department.Name` it uses a SQL INNER JOIN. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
            .Select(p => new CourseViewModel
            {
                CourseID = p.CourseID,
                Title = p.Title,
                Credits = p.Credits,
                DepartmentName = p.Department.Name
            }).ToListAsync();
}
```

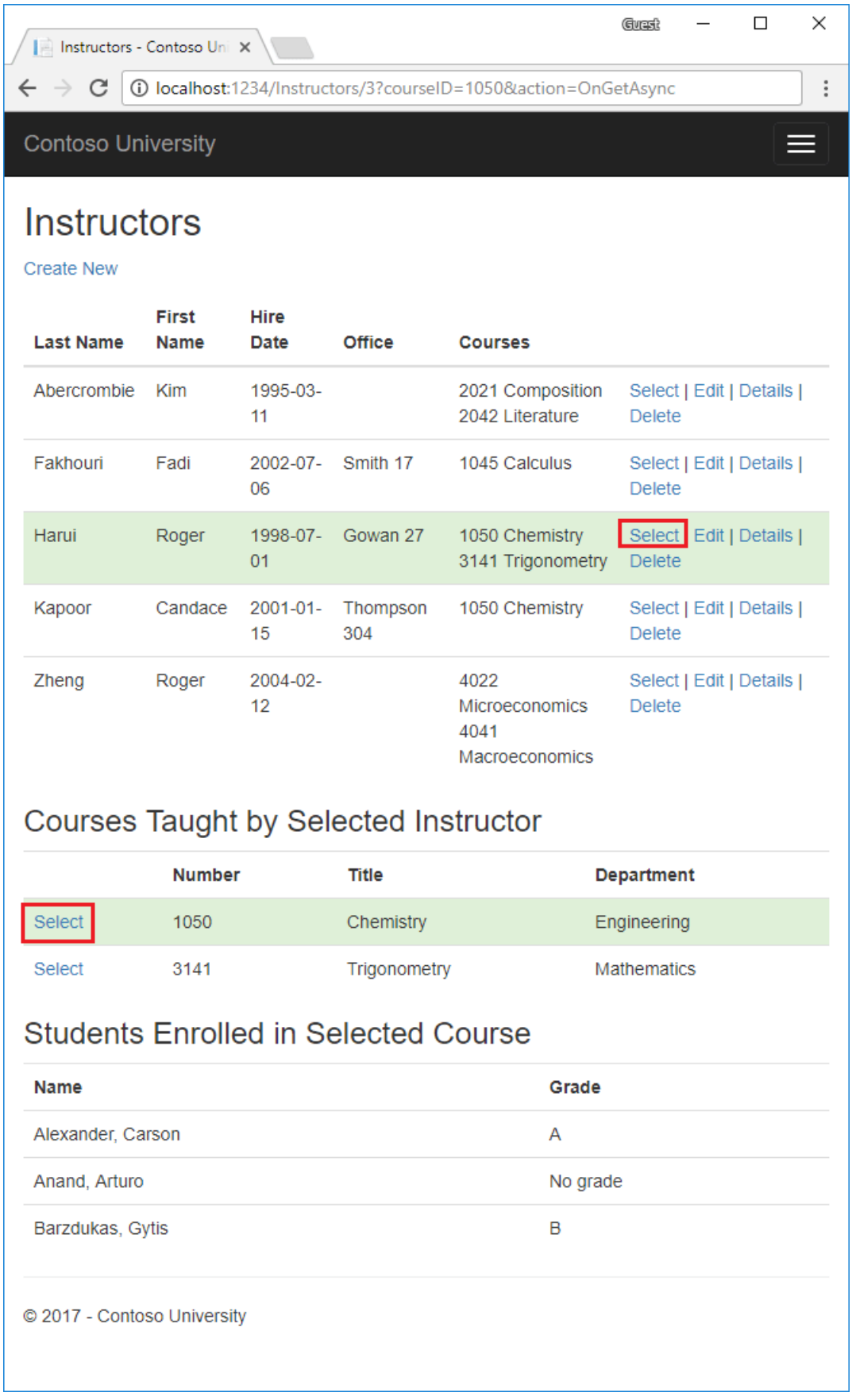
The `CourseViewModel`:

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelect.cshtml and IndexSelect.cshtml.cs for a complete example.

Create an Instructors page that shows Courses and Enrollments

In this section, the Instructors page is created.



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor (Harui in the preceding image), related `Course` entities are displayed. The

`Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.

- When the user selects a course (Chemistry in the preceding image), related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model for the Instructor Index view

The instructors page shows data from three different tables. A view model is created that includes the three entities representing the three tables.

In the *SchoolViewModels* folder, create *InstructorIndexData.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Scaffold the Instructor model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Instructor` for the model class.

The preceding command scaffolds the `Instructor` model. Run the app and navigate to the instructors page.

Replace *Pages/Instructors/Index.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData Instructor { get; set; }
        public int InstructorID { get; set; }

        public async Task OnGetAsync(int? id)
        {
            Instructor = new InstructorIndexData();
            Instructor.Instructors = await _context.Instructors
                  .Include(i => i.OfficeAssignment)
                  .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                  .AsNoTracking()
                  .OrderBy(i => i.LastName)
                  .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the *Pages/Instructors/Index.cshtml.cs* file:

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The query has two includes:

- `OfficeAssignment`: Displayed in the instructors view.
- `CourseAssignments`: Which brings in the courses taught.

Update the instructors Index page

Update *Pages/Instructors/Index.cshtml* with the following markup:

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructor.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding markup makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `http://localhost:1234/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the previous URL is:

  `http://localhost:1234/Instructors/2`

- Page title is **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

  ```
  @if (item.OfficeAssignment != null)
  {
      @item.OfficeAssignment.Location
  }
  ```

- Added a **Courses** column that displays courses taught by each instructor. See Explicit line transition for more about this razor syntax.
- Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

  ```
  string selectedRow = "";
  if (item.CourseID == Model.CourseID)
  {
      selectedRow = "success";
  }
  <tr class="@selectedRow">
  ```

- Added a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

  ```
  <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
  ```

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link. The row style changes.

Add courses taught by selected instructor

Update the `OnGetAsync` method in *Pages/Instructors/Index.cshtml.cs* with the following code:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }
}
```

Add `public int CourseID { get; set; }`

```
public class IndexModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public IndexModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    public InstructorIndexData Instructor { get; set; }
    public int InstructorID { get; set; }
    public int CourseID { get; set; }

    public async Task OnGetAsync(int? id, int? courseID)
    {
        Instructor = new InstructorIndexData();
        Instructor.Instructors = await _context.Instructors
              .Include(i => i.OfficeAssignment)
              .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Department)
              .AsNoTracking()
              .OrderBy(i => i.LastName)
              .ToListAsync();

        if (id != null)
        {
            InstructorID = id.Value;
            Instructor instructor = Instructor.Instructors.Where(
                i => i.ID == id.Value).Single();
            Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
        }

        if (courseID != null)
        {
            CourseID = courseID.Value;
            Instructor.Enrollments = Instructor.Courses.Where(
                x => x.CourseID == courseID).Single().Enrollments;
        }
    }
```

Examine the updated query:

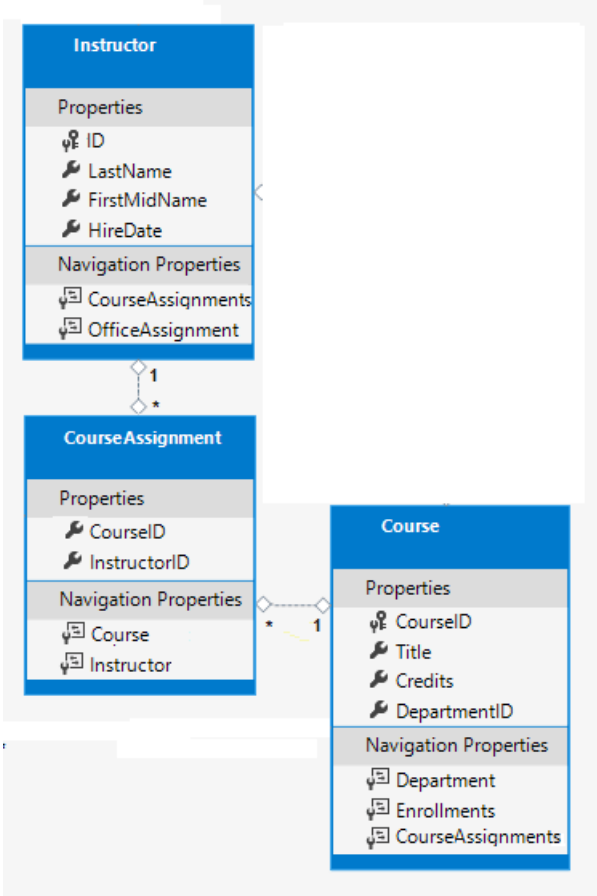
```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The preceding query adds the `Department` entities.

The following code executes when an instructor is selected (`id != null`). The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = Instructor.Instructors.Where(
        i => i.ID == id.Value).Single();
    Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection. In the preceding `Where` method, only a single `Instructor` entity is returned. The `Single` method converts the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `CourseAssignments` property. `CourseAssignments` provides access to the related `Course` entities.



The `Single` method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. Using `SingleOrDefault` on an empty collection:

- Results in an exception (from trying to find a `Courses` property on a null reference).
- The exception message would less clearly indicate the cause of the problem.

The following code populates the view model's `Enrollments` property when a course is selected:

```
if (courseID != null)
{
    CourseID = courseID.Value;
    Instructor.Enrollments = Instructor.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Add the following markup to the end of the *Pages/Instructors/Index.cshtml* Razor Page:

```
                    <a asp-page="./Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.Instructor.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Instructor.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    <a asp-page="./Index" asp-route-courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

The preceding markup displays a list of courses related to an instructor when an instructor is selected.

Test the app. Click on a **Select** link on the instructors page.

Show student data

In this section, the app is updated to show the student data for a selected course.

Update the query in the `OnGetAsync` method in *Pages/Instructors/Index.cshtml.cs* with the following code:

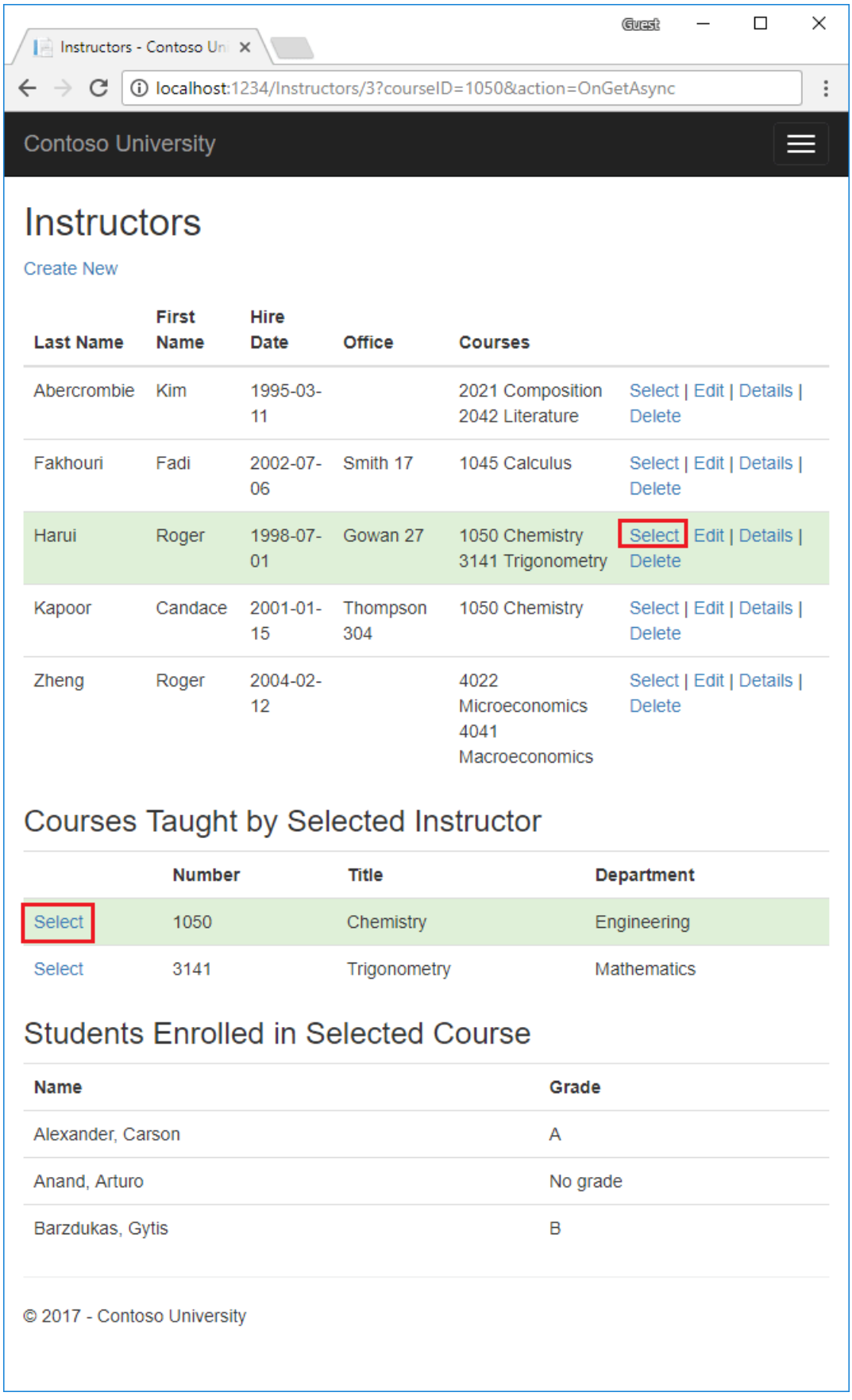
```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Update *Pages/Instructors/Index.cshtml*. Add the following markup to the end of the file:

```
@if (Model.Instructor.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Instructor.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding markup displays a list of the students who are enrolled in the selected course.

Refresh the page and select an instructor. Select a course to see the list of enrolled students and their grades.



Using Single

The `Single` method can pass in the `Where` condition instead of calling the `Where` method separately:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();

    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
                    .ThenInclude(i => i.Enrollments)
                        .ThenInclude(i => i.Student)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Single(
            i => i.ID == id.Value);
        Instructor.Courses = instructor.CourseAssignments.Select(
            s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        Instructor.Enrollments = Instructor.Courses.Single(
            x => x.CourseID == courseID).Enrollments;
    }
}
```

The preceding `Single` approach provides no benefits over using `Where`. Some developers prefer the `Single` approach style.

Explicit loading

The current code specifies eager loading for `Enrollments` and `Students`:

```
Instructor.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Suppose users rarely want to see enrollments in a course. In that case, an optimization would be to only load the enrollment data if it's requested. In this section, the `OnGetAsync` is updated to use explicit loading of `Enrollments` and `Students`.

Update the `OnGetAsync` with the following code:

```
public async Task OnGetAsync(int? id, int? courseID)
{
    Instructor = new InstructorIndexData();
    Instructor.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
            //.Include(i => i.CourseAssignments)
            //    .ThenInclude(i => i.Course)
            //        .ThenInclude(i => i.Enrollments)
            //            .ThenInclude(i => i.Student)
         // .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();


    if (id != null)
    {
        InstructorID = id.Value;
        Instructor instructor = Instructor.Instructors.Where(
            i => i.ID == id.Value).Single();
        Instructor.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        CourseID = courseID.Value;
        var selectedCourse = Instructor.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        Instructor.Enrollments = selectedCourse.Enrollments;
    }
}
```

The preceding code drops the *ThenInclude* method calls for enrollment and student data. If a course is selected, the highlighted code retrieves:

- The `Enrollment` entities for the selected course.
- The `Student` entities for each `Enrollment`.

Notice the preceding code comments out `.AsNoTracking()`. Navigation properties can only be explicitly loaded for tracked entities.

Test the app. From a users perspective, the app behaves identically to the previous version.

The next tutorial shows how to update related data.

Additional resources

- YouTube version of this tutorial (part1)
- YouTube version of this tutorial (part2)

Part 7, Razor Pages with EF Core in ASP.NET Core - Update Related Data

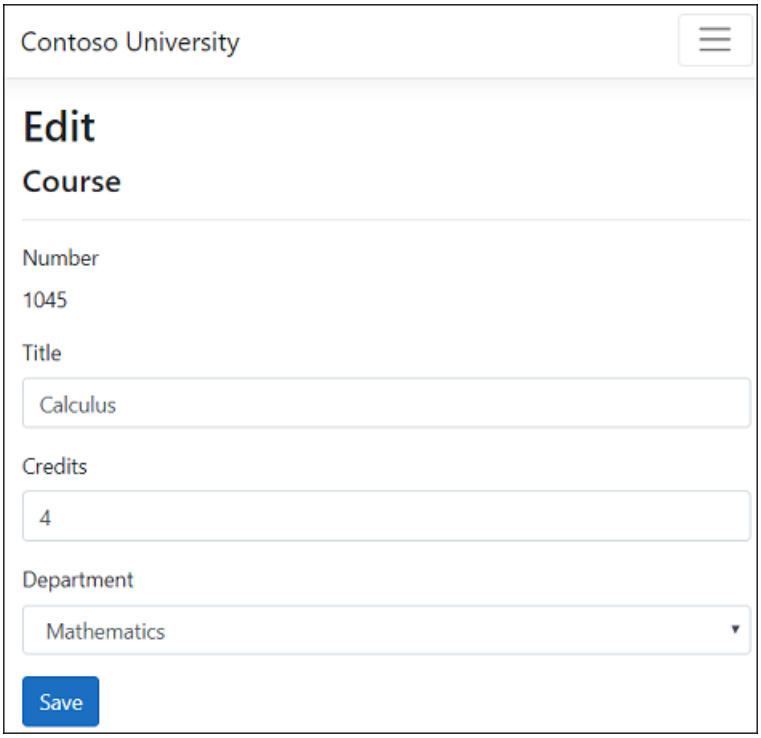
9/22/2020 • 32 minutes to read • Edit Online

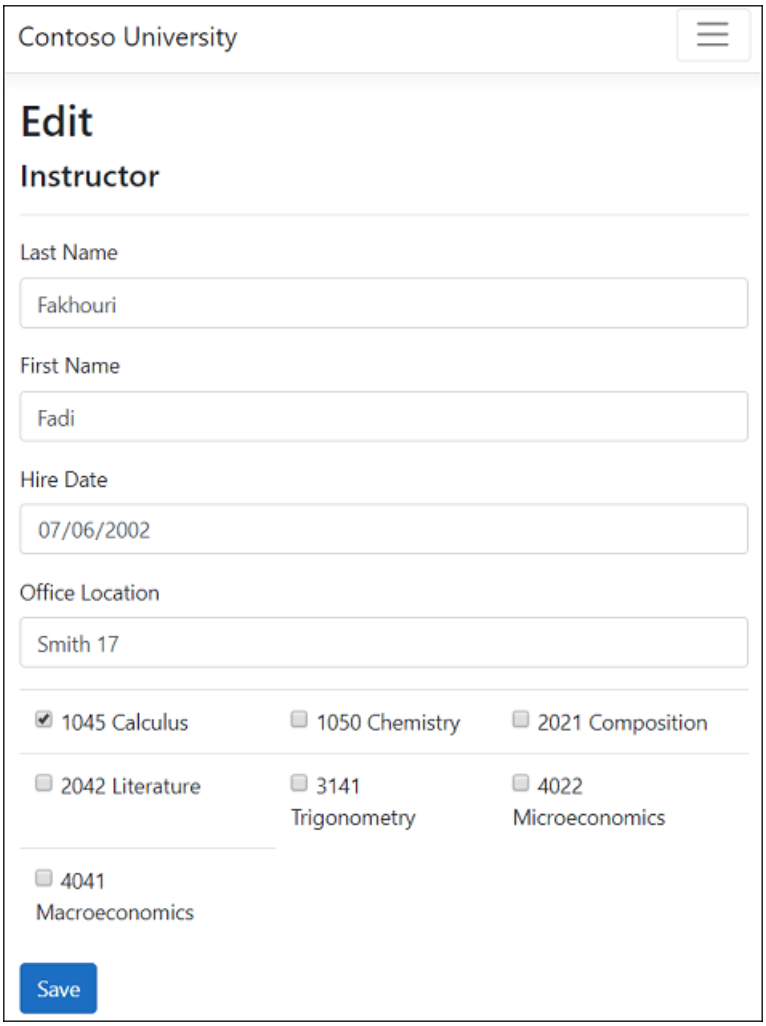
By Tom Dykstra, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to update related data. The following illustrations show some of the completed pages.





Update the Course Create and Edit pages

The scaffolded code for the Course Create and Edit pages has a Department drop-down list that shows Department ID (an integer). The drop-down should show the Department name, so both of these pages need a list of department names. To provide that list, use a base class for the Create and Edit pages.

Create a base class for Course Create and Edit

Create a *Pages/Courses/DepartmentNamePageModel.cs* file with the following code:

```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

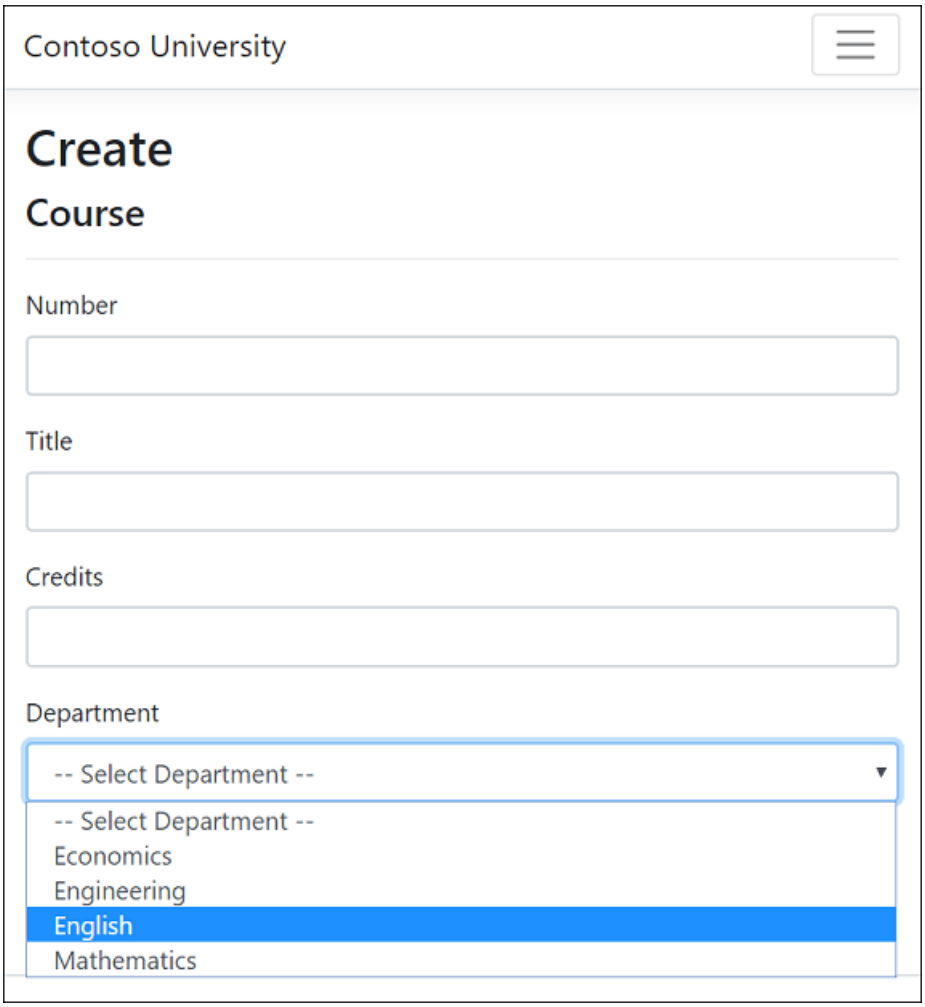
            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Update the Course Create page model

A Course is assigned to a Department. The base class for the Create and Edit pages provides a `SelectList` for selecting the department. The drop-down list that uses the `SelectList` sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update *Pages/Courses/Create.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses `TryUpdateModelAsync` to prevent overposting.
- Removes `ViewData["DepartmentID"]`. `DepartmentNameSL` from the base class is a strongly typed model and will be used by the Razor page. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Course Create Razor page

Update *Pages/Courses/Create.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" in the drop-down when no department has been selected yet, rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Course Edit page model

Update *Pages/Courses/Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (courseToUpdate == null)
            {
                return NotFound();
            }
```

```
            }

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which selects that department in the drop-down list.

Update the Course Edit Razor page

Update *Pages/Courses/Edit.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption for the Department drop-down from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for the course number to be included in the posted data when the user clicks **Save**.

Update the Course Details and Delete pages

AsNoTracking can improve performance when tracking isn't required.

Update the Course page models

Update *Pages/Courses/Delete.cshtml.cs* with the following code to add `AsNoTracking`:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .AsNoTracking()
                .Include(c => c.Department)
                .FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses.FindAsync(id);

            if (Course != null)
            {
                _context.Courses.Remove(Course);
                await _context.SaveChangesAsync();
            }

            return RedirectToPage("./Index");
        }
    }
}
```

Make the same change in the *Pages/Courses/Details.cshtml.cs* file:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class DetailsModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DetailsModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                 .AsNoTracking()
                 .Include(c => c.Department)
                 .FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }
            return Page();
        }
    }
}
```

Update the Course Razor pages

Update *Pages/Courses/Delete.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

```
@page
@model ContosoUniversity.Pages.Courses.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

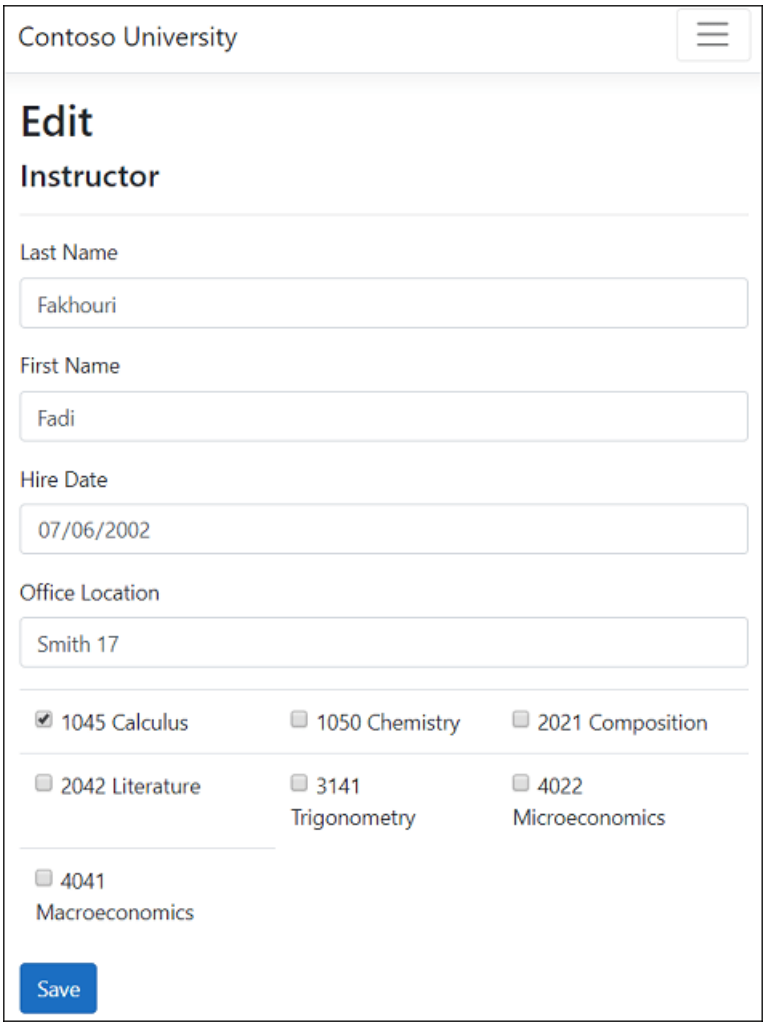
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Course.CourseID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

Test the Course pages

Test the create, edit, details, and delete pages.

Update the instructor Create and Edit pages

Instructors may teach any number of courses. The following image shows the instructor Edit page with an array of course checkboxes.



The checkboxes enable changes to courses an instructor is assigned to. A checkbox is displayed for every course in the database. Courses that the instructor is assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, a different UI might work better. But the method of managing a many-to-many relationship shown here wouldn't change. To create or delete relationships, you manipulate a join entity.

Create a class for assigned courses data

Create *SchoolViewModels/AssignedCourseData.cs* with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the check boxes for courses assigned to an instructor.

Create an Instructor page model base class

Create the *Pages/Instructors/InstructorCoursesPageModel.cs* base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
```

```
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            string[] selectedCourses, Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
            foreach (var course in context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.CourseAssignments.Add(
                            new CourseAssignment
                            {
                                InstructorID = instructorToUpdate.ID,
                                CourseID = course.CourseID
                            });
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        CourseAssignment courseToRemove
                            = instructorToUpdate
                                .CourseAssignments
                                .SingleOrDefault(i => i.CourseID == course.CourseID);
                        context.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class you will use for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used for efficient lookups.

Since the Razor page doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload with declared properties and `CourseAssignments` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```
if (selectedCourses == null)
{
    instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
    return;
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the page. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.CourseAssignments.Add(
            new CourseAssignment
            {
                InstructorID = instructorToUpdate.ID,
                CourseID = course.CourseID
            });
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        CourseAssignment courseToRemove
            = instructorToUpdate
                .CourseAssignments
                .SingleOrDefault(i => i.CourseID == course.CourseID);
        context.Remove(courseToRemove);
    }
}
```

Handle office location

Another relationship the edit page has to handle is the one-to-zero-or-one relationship that the Instructor entity has with the `OfficeAssignment` entity. The instructor edit code must handle the following scenarios:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.
- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the Instructor Edit page model

Update *Pages/Instructors/Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class EditModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            PopulateAssignedCourseData(_context, Instructor);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
        {
            if (id == null)
            {
                return NotFound();
            }

            var instructorToUpdate = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.CourseAssignments)
                    .ThenInclude(i => i.Course)
                .FirstOrDefaultAsync(s => s.ID == id);

            if (instructorToUpdate == null)
            {
```

```
            {
                return NotFound();
            }

            if (await TryUpdateModelAsync<Instructor>(
                instructorToUpdate,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                if (String.IsNullOrWhiteSpace(
                    instructorToUpdate.OfficeAssignment?.Location))
                {
                    instructorToUpdate.OfficeAssignment = null;
                }
                UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            PopulateAssignedCourseData(_context, instructorToUpdate);
            return Page();
        }
    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment`, `CourseAssignment`, and `CourseAssignment.Course` navigation properties.
- Updates the retrieved `Instructor` entity with values from the model binder. `TryUpdateModel` prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.
- Calls `PopulateAssignedCourseData` in `OnGetAsync` to provide information for the checkboxes using the `AssignedCourseData` view model class.
- Calls `UpdateInstructorCourses` in `OnPostAsync` to apply information from the checkboxes to the Instructor entity being edited.
- Calls `PopulateAssignedCourseData` and `UpdateInstructorCourses` in `OnPostAsync` if `TryUpdateModel` fails. These method calls restore the assigned course data entered on the page when it is redisplayed with an error message.

Update the Instructor Edit Razor page

Update *Pages/Instructors/Edit.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
```

```
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code creates an HTML table that has three columns. Each column has a checkbox and a caption containing the course number and title. The checkboxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each checkbox is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the checkboxes that are selected.

When the checkboxes are initially rendered, courses assigned to the instructor are selected.

Note: The approach taken here to edit instructor course data works well when there's a limited number of

courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Run the app and test the updated Instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Update the Instructor Create page

Update the Instructor Create page model and Razor page with code similar to the Edit page:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List<CourseAssignment>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
            var newInstructor = new Instructor();
            if (selectedCourses != null)
            {
                newInstructor.CourseAssignments = new List<CourseAssignment>();
                foreach (var course in selectedCourses)
                {
                    var courseToAdd = new CourseAssignment
                    {
                        CourseID = int.Parse(course)
                    };
                    newInstructor.CourseAssignments.Add(courseToAdd);
                }
            }

            if (await TryUpdateModelAsync<Instructor>(
                newInstructor,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel
```

```
@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>
```

```
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test the instructor Create page.

Update the Instructor Delete page

Update *Pages/Instructors/Delete.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            if (instructor == null)
            {
                return RedirectToPage("./Index");
            }

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `CourseAssignments` navigation property. `CourseAssignments` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

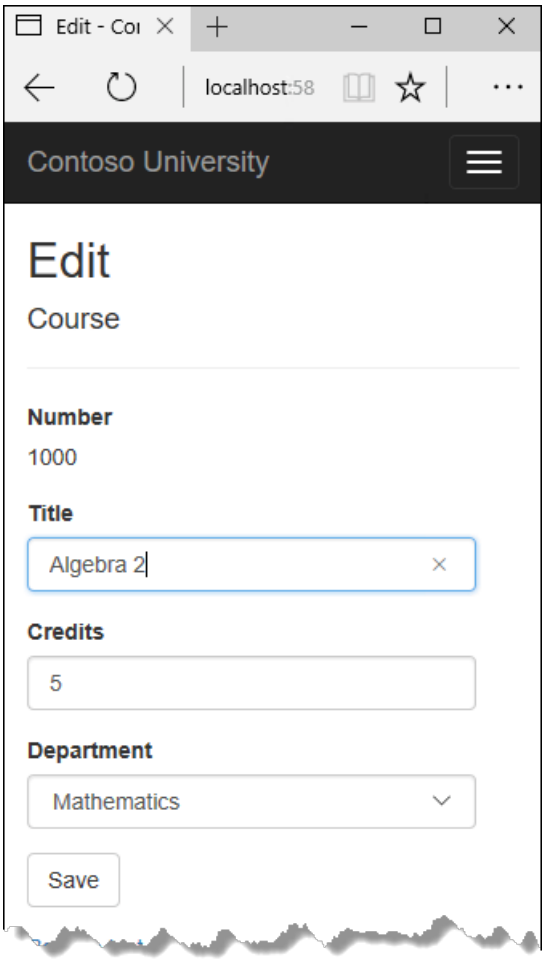
Run the app and test the Delete page.

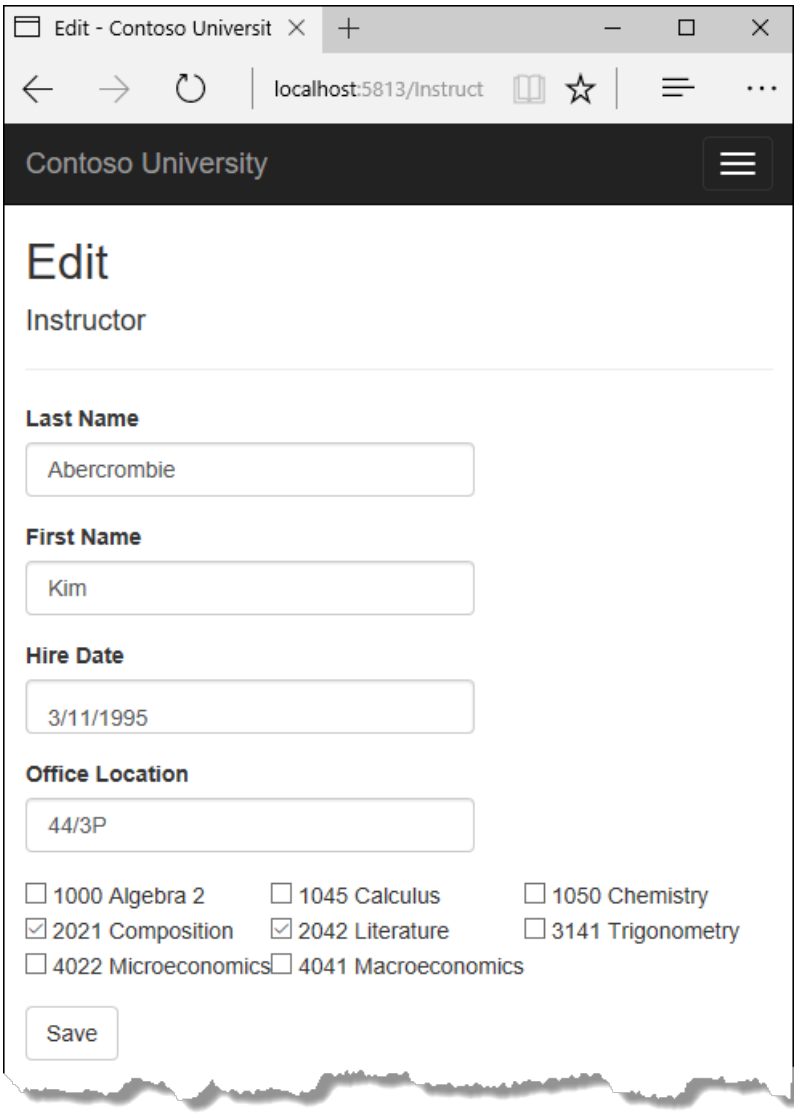
Next steps

PREVIOUS TUTORIAL NEXT TUTORIAL

This tutorial demonstrates updating related data. If you run into problems you can't solve, download or view the completed app. Download instructions.

The following illustrations shows some of the completed pages.





Examine and test the Create and Edit course pages. Create a new course. The department is selected by its primary key (an integer), not its name. Edit the new course. When you have finished testing, delete the new course.

Create a base class to share common code

The Courses/Create and Courses/Edit pages each need a list of department names. Create the *Pages/Courses/DepartmentNamePageModel.cshtml.cs* base class for the Create and Edit pages:

```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

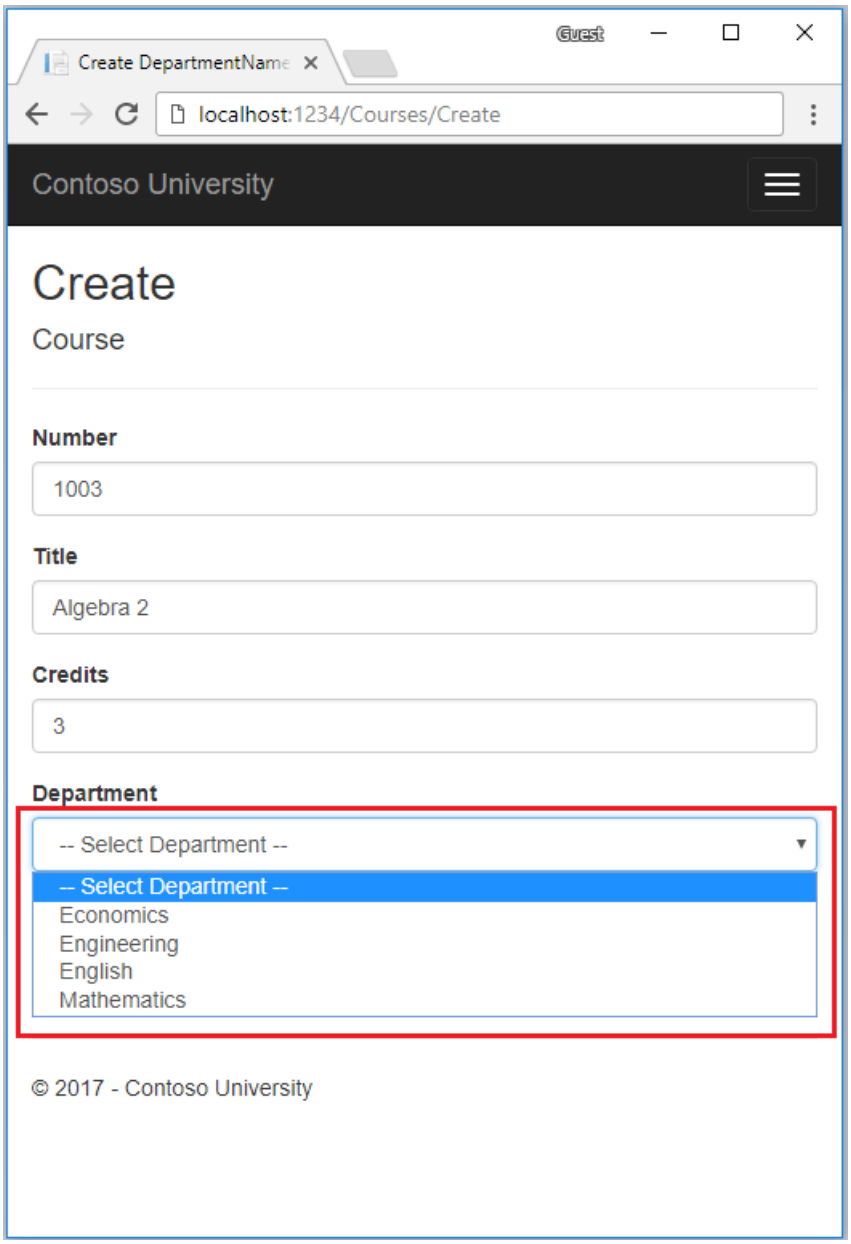
            DepartmentNameSL = new SelectList(departmentsQuery.AsNoTracking(),
                        "DepartmentID", "Name", selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Customize the Courses Pages

When a new course entity is created, it must have a relationship to an existing department. To add a department while creating a course, the base class for Create and Edit contains a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update the Create page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s => s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, emptyCourse.DepartmentID);
            return Page();
        }
      }
}
```

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses `TryUpdateModelAsync` to prevent overposting.
- Replaces `ViewData["DepartmentID"]` with `DepartmentNameSL` (from the base class).

`ViewData["DepartmentID"]` is replaced with the strongly typed `DepartmentNameSL`. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Courses Create page

Update *Pages/Courses/Create.cshtml* with the following code:

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Courses Edit page.

Replace the code in *Pages/Courses/Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m => m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context,Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context, courseToUpdate.DepartmentID);
            return Page();
        }
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which select the department specified in the drop-

down list.

Update *Pages/Courses/Edit.cshtml* with the following markup:

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label"></label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label"></label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label"></label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is

required for the course number to be included in the posted data when the user clicks **Save**.

Test the updated code. Create, edit, and delete a course.

Add AsNoTracking to the Details and Delete page models

AsNoTracking can improve performance when tracking isn't required. Add `AsNoTracking` to the Delete and Details page model. The following code shows the updated Delete page model:

```
public class DeleteModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public DeleteModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Course Course { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .Include(c => c.Department)
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Course = await _context.Courses
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.CourseID == id);

        if (Course != null)
        {
            _context.Courses.Remove(Course);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage("./Index");
    }
}
```

Update the `OnGetAsync` method in the *Pages/Courses/Details.cshtml.cs* file:

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Course = await _context.Courses
         .AsNoTracking()
         .Include(c => c.Department)
         .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

Modify the Delete and Details pages

Update the Delete Razor page with the following markup:

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Course.Department.DepartmentID)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

Test the Course pages

Test create, edit, details, and delete.

Update the instructor pages

The following sections update the instructor pages.

Add office location

When editing an instructor record, you may want to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity. The instructor code must handle:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.

- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the instructors Edit page model with the following code:

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            await _context.SaveChangesAsync();
        }
        return RedirectToPage("./Index");

    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property.
- Updates the retrieved `Instructor` entity with values from the model binder. `TryUpdateModel` prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.

Update the instructor Edit page

Update *Pages/Instructors/Edit.cshtml* with the office location:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

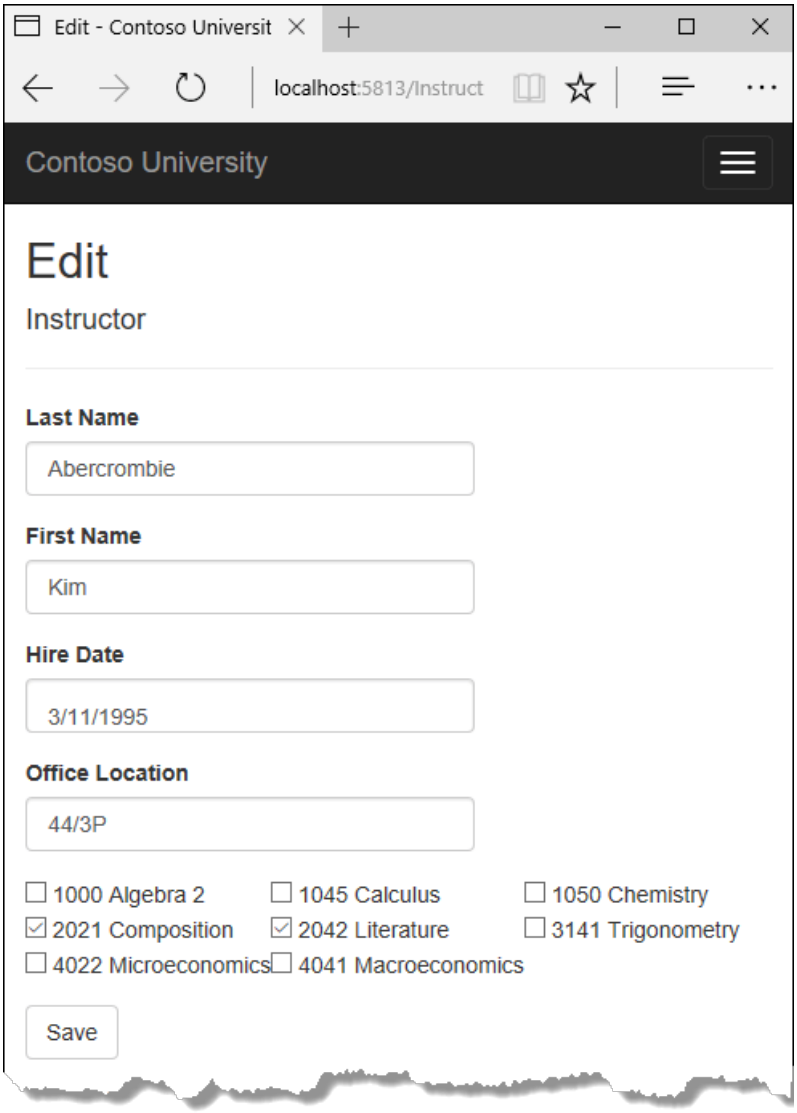
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Verify you can change an instructors office location.

Add Course assignments to the instructor Edit page

Instructors may teach any number of courses. In this section, you add the ability to change course assignments.

The following image shows the updated instructor Edit page:



`Course` and `Instructor` has a many-to-many relationship. To add and remove relationships, you add and remove entities from the `CourseAssignments` join entity set.

Check boxes enable changes to courses an instructor is assigned to. A check box is displayed for every course in the database. Courses that the instructor is assigned to are checked. The user can select or clear check boxes to change course assignments. If the number of courses were much greater:

- You'd probably use a different user interface to display the courses.
- The method of manipulating a join entity to create or delete relationships wouldn't change.

Add classes to support Create and Edit instructor pages

Create *SchoolViewModels/AssignedCourseData.cs* with the following code:

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the check boxes for assigned courses by an instructor.

Create the *Pages/Instructors/InstructorCoursesPageModel.cshtml.cs* base class:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {

        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.CourseAssignments.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }

        public void UpdateInstructorCourses(SchoolContext context,
            string[] selectedCourses, Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
            foreach (var course in context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.CourseAssignments.Add(
                            new CourseAssignment
                            {
                                InstructorID = instructorToUpdate.ID,
                                CourseID = course.CourseID
                            });
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        CourseAssignment courseToRemove
                            = instructorToUpdate
                                .CourseAssignments
                                .SingleOrDefault(i => i.CourseID == course.CourseID);
```

```
                        context.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class you will use for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used to create efficient lookups.

Instructors Edit page model

Update the instructor Edit page model with the following code:

```
public class EditModel : InstructorCoursesPageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Instructor Instructor { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Instructor = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
            .AsNoTracking()
            .FirstOrDefaultAsync(m => m.ID == id);

        if (Instructor == null)
        {
            return NotFound();
        }
        PopulateAssignedCourseData(_context, Instructor);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int? id, string[] selectedCourses)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var instructorToUpdate = await _context.Instructors
            .Include(i => i.OfficeAssignment)
            .Include(i => i.CourseAssignments)
                .ThenInclude(i => i.Course)
            .FirstOrDefaultAsync(s => s.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "Instructor",
            i => i.FirstMidName, i => i.LastName,
            i => i.HireDate, i => i.OfficeAssignment))
        {
            if (String.IsNullOrWhiteSpace(
                instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
        UpdateInstructorCourses(_context, selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(_context, instructorToUpdate);
        return Page();
    }
}
```

The preceding code handles office assignment changes.

Update the instructor Razor View:

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
```

```
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

> **NOTE**
>
> When you paste the code in Visual Studio, line breaks are changed in a way that breaks the code. Press Ctrl+Z one time to undo the automatic formatting. Ctrl+Z fixes the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown. With the block of new code selected, press Tab three times to line up the new code with the existing code. Vote on or review the status of this bug with this link.

The preceding code creates an HTML table that has three columns. Each column has a check box and a caption containing the course number and title. The check boxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each check box is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the check boxes that are selected.

When the check boxes are initially rendered, courses assigned to the instructor have checked attributes.

Run the app and test the updated instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Note: The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Update the instructors Create page

Update the instructor Create page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.CourseAssignments = new List<CourseAssignment>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData( context, instructor);
```

```
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var newInstructor = new Instructor();
            if (selectedCourses != null)
            {
                newInstructor.CourseAssignments = new List<CourseAssignment>();
                foreach (var course in selectedCourses)
                {
                    var courseToAdd = new CourseAssignment
                    {
                        CourseID = int.Parse(course)
                    };
                    newInstructor.CourseAssignments.Add(courseToAdd);
                }
            }

            if (await TryUpdateModelAsync<Instructor>(
                newInstructor,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

The preceding code is similar to the *Pages/Instructors/Edit.cshtml.cs* code.

Update the instructor Create Razor page with the following markup:

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label"></label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-danger"></span>
            </div>
```

```
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-control" />
                <span asp-validation-for="Instructor.FirstMidName" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label"></label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location" class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location" class="form-control" />
                <span asp-validation-for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="col-md-offset-2 col-md-10">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                               @course.CourseID @:  @course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test the instructor Create page.

Update the Delete page

Update the Delete page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.SingleAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            Instructor instructor = await _context.Instructors
                .Include(i => i.CourseAssignments)
                .SingleAsync(i => i.ID == id);

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `CourseAssignments` navigation property. `CourseAssignments` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Additional resources

- YouTube version of this tutorial (Part 1)
- YouTube version of this tutorial (Part 2)

PREVIOUS NEXT

Part 8, Razor Pages with EF Core in ASP.NET Core - Concurrency

9/22/2020 • 36 minutes to read • Edit Online

By Rick Anderson, Tom Dykstra, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to handle conflicts when multiple users update an entity concurrently (at the same time).

Concurrency conflicts

A concurrency conflict occurs when:

- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the database.

If concurrency detection isn't enabled, whoever updates the database last overwrites the other user's changes. If this risk is acceptable, the cost of programming for concurrency might outweigh the benefit.

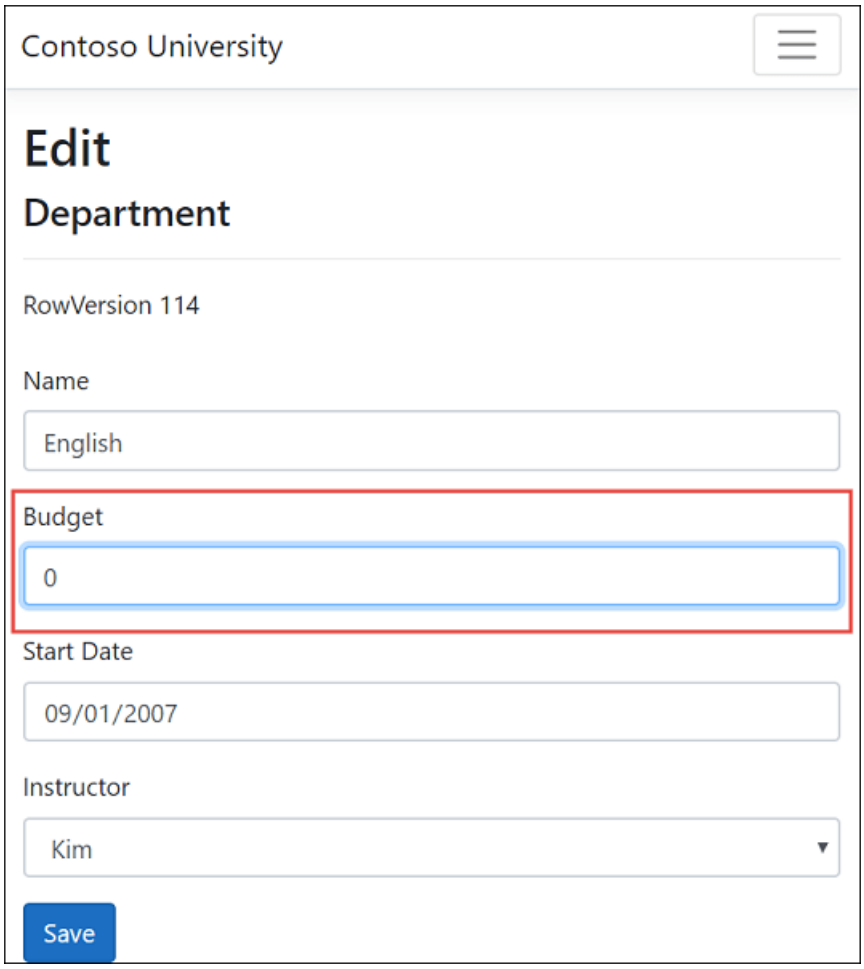
Pessimistic concurrency (locking)

One way to prevent concurrency conflicts is to use database locks. This is called pessimistic concurrency. Before the app reads a database row that it intends to update, it requests a lock. Once a row is locked for update access, no other users are allowed to lock the row until the first lock is released.

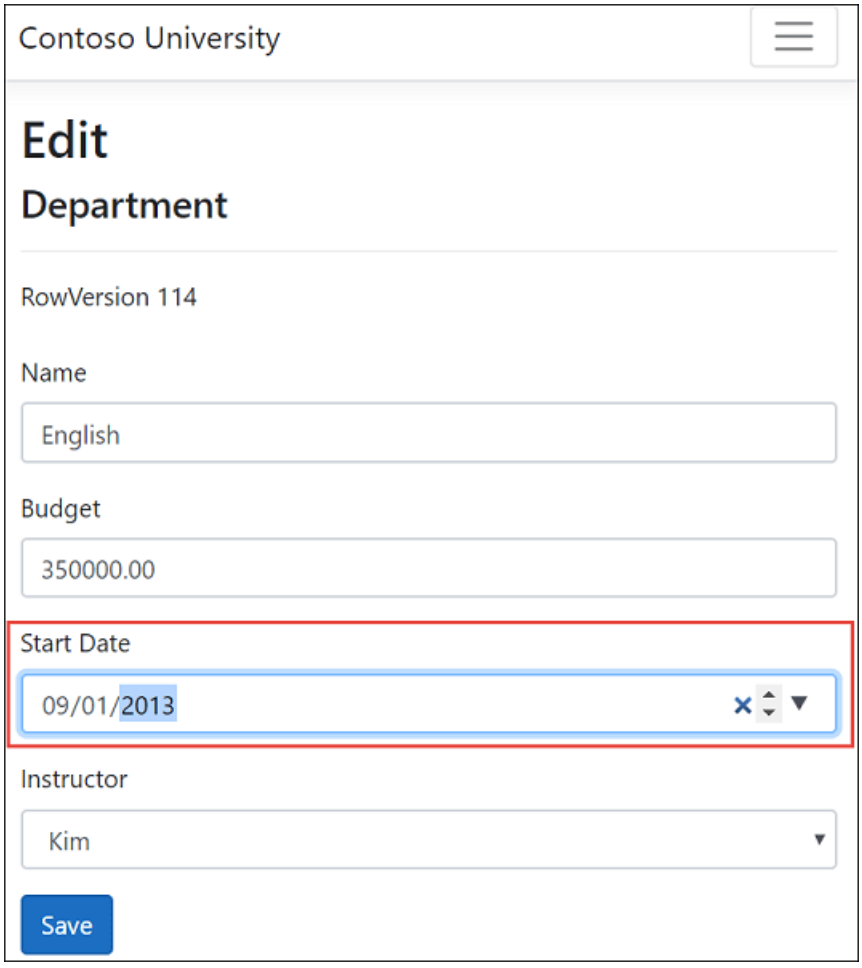
Managing locks has disadvantages. It can be complex to program and can cause performance problems as the number of users increases. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show how to implement it.

Optimistic concurrency

Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change take effect, since the browser displays the Index page with zero as the

Budget amount.

John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

  In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach has some disadvantages:

  - Can't avoid data loss if competing changes are made to the same property.
  - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.
  - Can increase app complexity compared to concurrency detection on an entity.
- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the fetched $350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) If you don't do any coding for concurrency handling, Client Wins happens automatically.
- You can prevent John's change from being updated in the database. Typically, the app would:

  - Display an error message.
  - Show the current state of the data.
  - Allow the user to reapply the changes.

  This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Conflict detection in EF Core

EF Core throws `DbConcurrencyException` exceptions when it detects conflicts. The data model has to be configured to enable conflict detection. Options for enabling conflict detection include the following:

- Configure EF Core to include the original values of columns configured as concurrency tokens in the Where clause of Update and Delete commands.

  When `SaveChanges` is called, the Where clause looks for the original values of any properties annotated with the ConcurrencyCheckAttribute attribute. The update statement won't find a row to update if any of the concurrency token properties changed since the row was first read. EF Core interprets that as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require large amounts of state. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.
- In the database table, include a tracking column that can be used to determine when a row has been changed.

  In a SQL Server database, the data type of the tracking column is `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version number). If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value. In that case, the Update or Delete statement can't find the row to update because

of the Where clause. EF Core throws a concurrency exception when no rows are affected by an Update or Delete command.

Add a tracking property

In *Models/Department.cs*, add a tracking property named RowVersion:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The TimestampAttribute attribute is what identifies the column as a concurrency tracking column. The fluent API is an alternative way to specify the tracking property:

```
modelBuilder.Entity<Department>()
  .Property<byte[]>("RowVersion")
  .IsRowVersion();
```

- Visual Studio
- Visual Studio Code

For a SQL Server database, the `[Timestamp]` attribute on an entity property defined as byte array:

- Causes the column to be included in DELETE and UPDATE WHERE clauses.
- Sets the column type in the database to rowversion.

The database generates a sequential row version number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the fetched row version value. If the row being updated has changed since it was fetched:

- The current row version value doesn't match the fetched value.

- The `Update` or `Delete` commands don't find a row because the `Where` clause looks for the fetched row version value.
- A `DbUpdateConcurrencyException` is thrown.

The following code shows a portion of the T-SQL generated by EF Core when the Department name is updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

The preceding highlighted code shows the `WHERE` clause containing `RowVersion`. If the database `RowVersion` doesn't equal the `RowVersion` parameter (`@p2`), no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT returns the number of rows affected by the last statement. If no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

Update the database

Adding the `RowVersion` property changes the data model, which requires a migration.

Build the project.

- Visual Studio
- Visual Studio Code

- Run the following command in the PMC:

  ```
  Add-Migration RowVersion
  ```

This command:

- Creates the *Migrations/{time stamp}_RowVersion.cs* migration file.
- Updates the *Migrations/SchoolContextModelSnapshot.cs* file. The update adds the following highlighted code to the `BuildModel` method:

```
modelBuilder.Entity("ContosoUniversity.Models.Department", b =>
    {
        b.Property<int>("DepartmentID")
            .ValueGeneratedOnAdd()
            .HasAnnotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn);

        b.Property<decimal>("Budget")
            .HasColumnType("money");

        b.Property<int?>("InstructorID");

        b.Property<string>("Name")
            .HasMaxLength(50);

        b.Property<byte[]>("RowVersion")
            .IsConcurrencyToken()
            .ValueGeneratedOnAddOrUpdate();

        b.Property<DateTime>("StartDate");

        b.HasKey("DepartmentID");

        b.HasIndex("InstructorID");

        b.ToTable("Department");
    });
```

- Visual Studio
- Visual Studio Code
- Run the following command in the PMC:

```
Update-Database
```

Scaffold Department pages

- Visual Studio
- Visual Studio Code
- Follow the instructions in Scaffold Student pages with the following exceptions:
- Create a *Pages/Departments* folder.
- Use `Department` for the model class.
  - Use the existing context class instead of creating a new one.

Build the project.

Update the Index page

The scaffolding tool created a `RowVersion` column for the Index page, but that field wouldn't be displayed in a production app. In this tutorial, the last byte of the `RowVersion` is displayed to help show how concurrency handling works. The last byte isn't guaranteed to be unique by itself.

Update *Pages\Departments\Index.cshtml* page:

- Replace Index with Departments.

- Change the code containing `RowVersion` to show just the last byte of the byte array.
- Replace FirstMidName with FullName.

The following code shows the updated page:

```
@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Name)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Budget)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].StartDate)
                </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Administrator)
            </th>
            <th>
                RowVersion
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Department)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    @item.RowVersion[7]
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-page="./Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-page="./Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the Edit page model

Update *Pages\Departments\Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (departmentToUpdate == null)
            {
                return HandleDeletedDepartment();
            }

            _context.Entry(departmentToUpdate)
                .Property("RowVersion").OriginalValue = Department.RowVersion;

            if (
```

```
            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
            {
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues = (Department)exceptionEntry.Entity;
                    var databaseEntry = exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable to save. " +
                            "The department was deleted by another user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await setDbErrorMessage(dbValues, clientValues, _context);

                    // Save the current RowVersion so next postback
                    // matches unless an new concurrency issue happens.
                    Department.RowVersion = (byte[])dbValues.RowVersion;
                    // Clear the model error for the next postback.
                    ModelState.Remove("Department.RowVersion");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

        private IActionResult HandleDeletedDepartment()
        {
            var deletedDepartment = new Department();
            // ModelState contains the posted data because of the deletion error
            // and will overide the Department instance values when displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another user.");
            InstructorNameSL = new SelectList(_context.Instructors, "ID", "FullName",
Department.InstructorID);
            return Page();
        }

        private async Task setDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
```

```
            }
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in the database "
              + "have been displayed. If you still want to edit this record, click "
              + "the Save button again.");
        }
    }
}
```

The OriginalValue is updated with the `rowVersion` value from the entity when it was fetched in the `OnGet` method. EF Core generates a SQL UPDATE command with a WHERE clause containing the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), a `DbUpdateConcurrencyException` exception is thrown.

```
public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        return HandleDeletedDepartment();
    }

    _context.Entry(departmentToUpdate)
        .Property("RowVersion").OriginalValue = Department.RowVersion;
```

In the preceding highlighted code:

- The value in `Department.RowVersion` is what was in the entity when it was originally fetched in the Get request for the Edit page. The value is provided to the `OnPost` method by a hidden field in the Razor page that displays the entity to be edited. The hidden field value is copied to `Department.RowVersion` by the model binder.
- `OriginalValue` is what EF Core will use in the Where clause. Before the highlighted line of code executes, `OriginalValue` has the value that was in the database when `FirstOrDefaultAsync` was called in this method, which might be different from what was displayed on the Edit page.
- The highlighted code makes sure that EF Core uses the original `RowVersion` value from the displayed `Department` entity in the SQL UPDATE statement's Where clause.

When a concurrency error happens, the following highlighted code gets the client values (the values posted to this method) and the database values.

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await setDbErrorMessage(dbValues, clientValues, _context);

        // Save the current RowVersion so next postback
        // matches unless an new concurrency issue happens.
        Department.RowVersion = (byte[])dbValues.RowVersion;
        // Clear the model error for the next postback.
        ModelState.Remove("Department.RowVersion");
    }
```

The following code adds a custom error message for each column that has database values different from what was posted to `OnPostAsync`:

```
private async Task setDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database "
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

The following highlighted code sets the `RowVersion` value to the new value retrieved from the database. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await setDbErrorMessage(dbValues, clientValues, _context);

        // Save the current RowVersion so next postback
        // matches unless an new concurrency issue happens.
        Department.RowVersion = (byte[])dbValues.RowVersion;
        // Clear the model error for the next postback.
        ModelState.Remove("Department.RowVersion");
    }
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Edit page

Update *Pages/Departments/Edit.cshtml* with the following code:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.RowVersion" />
            <div class="form-group">
                <label>RowVersion</label>
                @Model.Department.RowVersion[7]
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label"></label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label"></label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label"></label>
                <input asp-for="Department.StartDate" class="form-control" />
                <span asp-validation-for="Department.StartDate" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
                <select asp-for="Department.InstructorID" class="form-control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `RowVersion` must be added so postback binds the value.
- Displays the last byte of `RowVersion` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

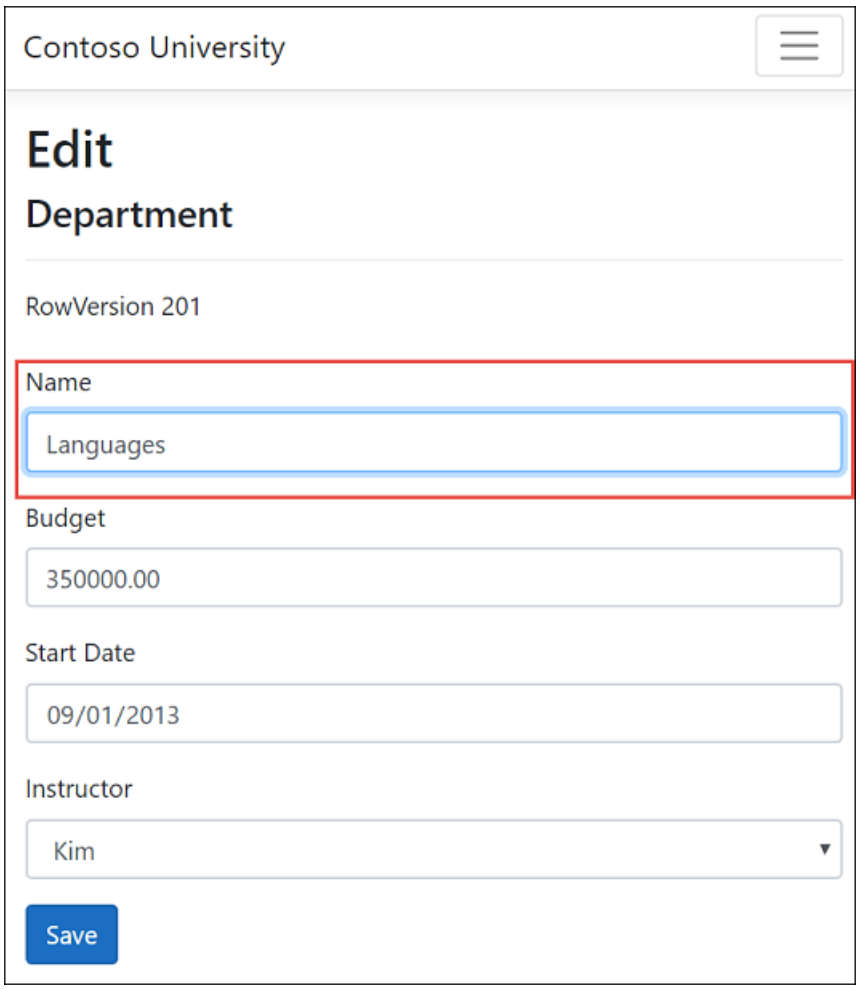
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

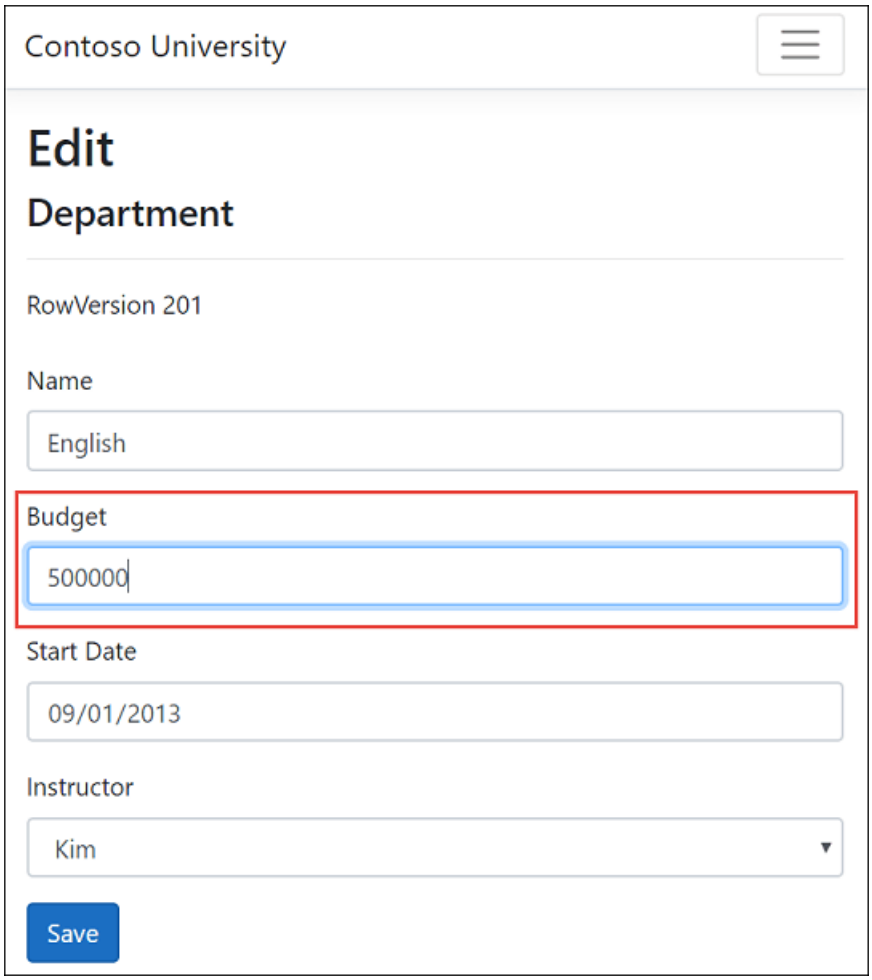
The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.

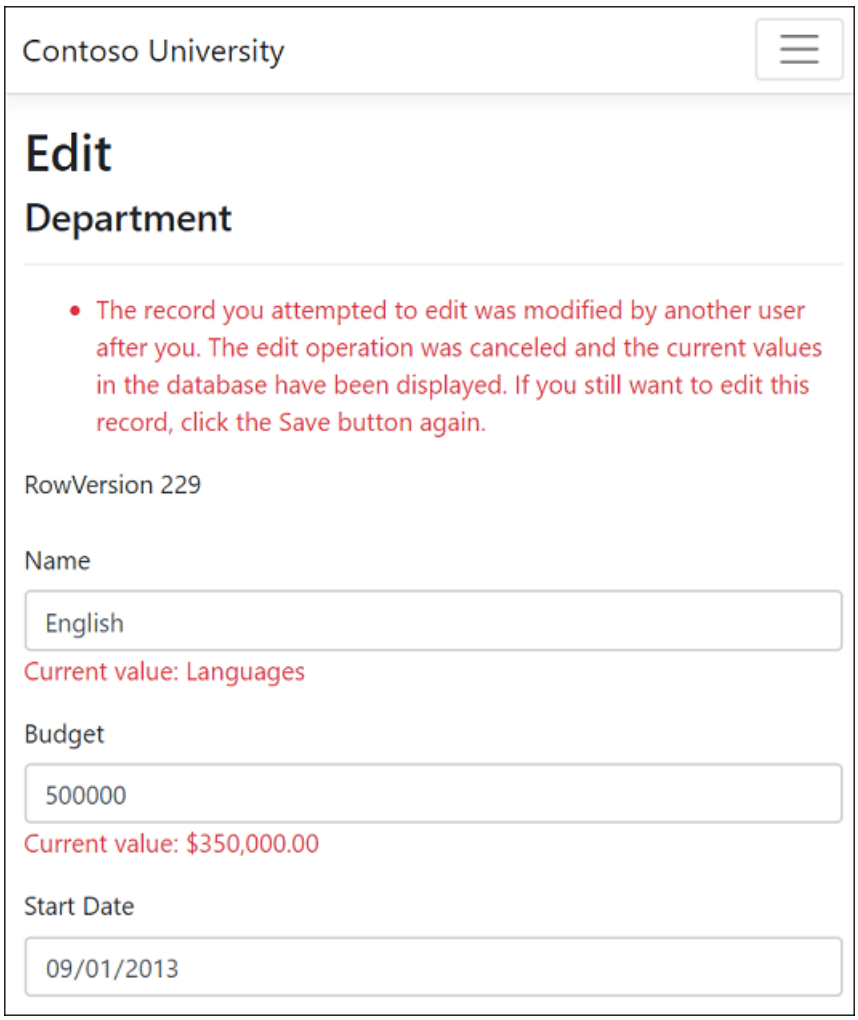


The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

Change a different field in the second browser tab.



Click **Save**. You see error messages for all fields that don't match the database values:



This browser window didn't intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.

Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page model

Update *Pages/Departments/Delete.cshtml.cs* with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool? concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                 return NotFound();
            }

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to delete "
                  + "was modified by another user after you selected delete. "
                  + "The delete operation was canceled and the current values in the "
                  + "database have been displayed. If you still want to delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.rowVersion value is from when the entity
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
```

```
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched. `Department.RowVersion` is the row version when the entity was fetched. When EF Core creates the SQL DELETE command, it includes a WHERE clause with `RowVersion`. If the SQL DELETE command results in zero rows affected:

- The `RowVersion` in the SQL DELETE command doesn't match `RowVersion` in the database.
- A DbUpdateConcurrencyException exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete page

Update *Pages/Departments/Delete.cshtml* with the following code:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.RowVersion)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.RowVersion[7])
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-danger" /> |
            <a asp-page="./Index">Back to List</a>
        </div>
</form>
</div>
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Changes `RowVersion` to display the last byte.

- Adds a hidden row version. `RowVersion` must be added so postback binds the value.

Test concurrency conflicts

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the database. Clicking **Delete** deletes the entity, unless `RowVersion` has been updated.

Additional resources

- Concurrency Tokens in EF Core
- Handle concurrency in EF Core
- Debugging ASP.NET Core 2.x source

Next steps

This is the last tutorial in the series. Additional topics are covered in the MVC version of this tutorial series.

PREVIOUS TUTORIAL

This tutorial shows how to handle conflicts when multiple users update an entity concurrently (at the same time). If you run into problems you can't solve, download or view the completed app. Download instructions.

Concurrency conflicts

A concurrency conflict occurs when:

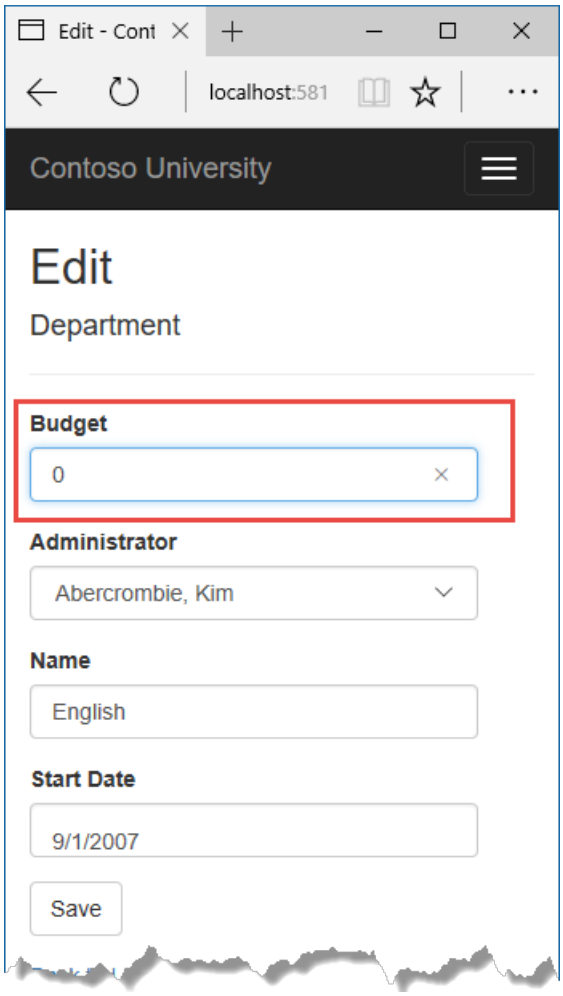
- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the DB.

If concurrency detection isn't enabled, when concurrent updates occur:

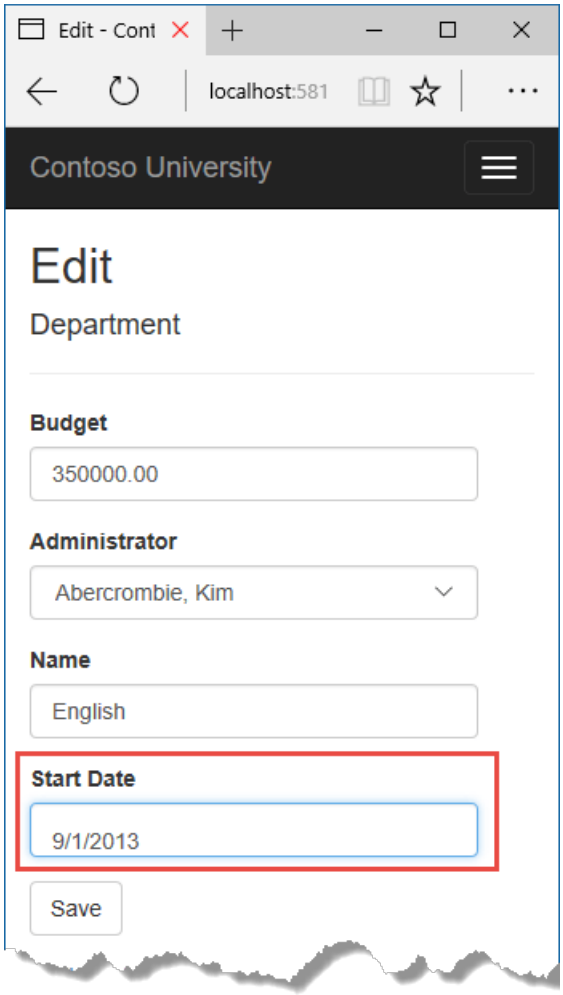
- The last update wins. That is, the last update values are saved to the DB.
- The first of the current updates are lost.

Optimistic concurrency

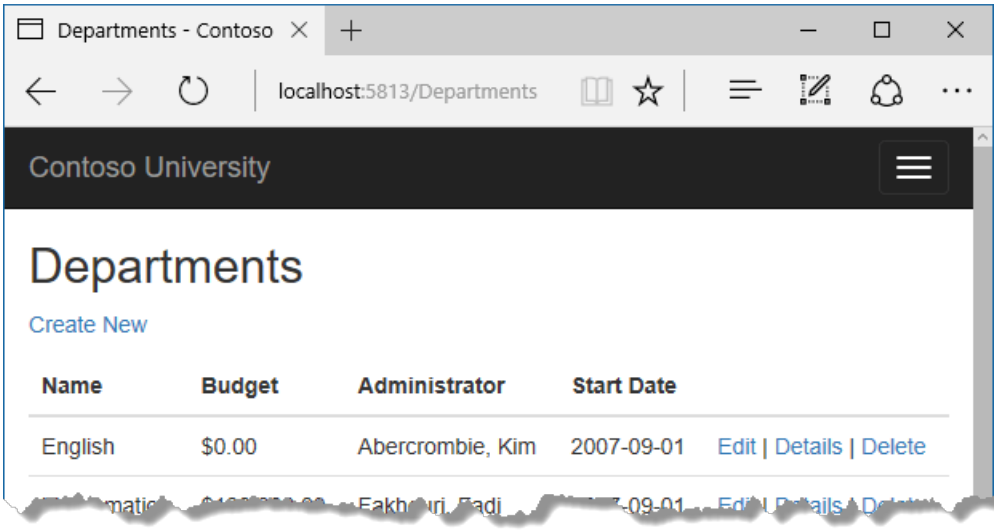
Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser displays the Index page.



John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Optimistic concurrency includes the following options:

- You can keep track of which property a user has modified and update only the corresponding columns in the DB.

  In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach:

  - Can't avoid data loss if competing changes are made to the same property.
  - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.
  - Can increase app complexity compared to concurrency detection on an entity.
- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the fetched $350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) If you don't do any coding for concurrency handling, Client Wins happens automatically.
- You can prevent John's change from being updated in the DB. Typically, the app would:

  - Display an error message.
  - Show the current state of the data.
  - Allow the user to reapply the changes.

  This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Handling concurrency

When a property is configured as a concurrency token:

- EF Core verifies that property has not been modified after it was fetched. The check occurs when SaveChanges or SaveChangesAsync is called.
- If the property has been changed after it was fetched, a DbUpdateConcurrencyException is thrown.

The DB and data model must be configured to support throwing `DbUpdateConcurrencyException`.

Detecting concurrency conflicts on a property

Concurrency conflicts can be detected at the property level with the ConcurrencyCheck attribute. The attribute can be applied to multiple properties on the model. For more information, see Data Annotations-ConcurrencyCheck.

The `[ConcurrencyCheck]` attribute isn't used in this tutorial.

Detecting concurrency conflicts on a row

To detect concurrency conflicts, a rowversion tracking column is added to the model. `rowversion`:

- Is SQL Server specific. Other databases may not provide a similar feature.
- Is used to determine that an entity has not been changed since it was fetched from the DB.

The DB generates a sequential `rowversion` number that's incremented each time the row is updated. In an `Update` or `Delete` command, the `Where` clause includes the fetched value of `rowversion`. If the row being updated has changed:

- `rowversion` doesn't match the fetched value.
- The `Update` or `Delete` commands don't find a row because the `Where` clause includes the fetched `rowversion`.
- A `DbUpdateConcurrencyException` is thrown.

In EF Core, when no rows have been updated by an `Update` or `Delete` command, a concurrency exception is thrown.

Add a tracking property to the Department entity

In *Models/Department.cs*, add a tracking property named RowVersion:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Timestamp attribute specifies that this column is included in the `Where` clause of `Update` and `Delete` commands. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` type replaced it.

The fluent API can also specify the tracking property:

```
modelBuilder.Entity<Department>()
  .Property<byte[]>("RowVersion")
  .IsRowVersion();
```

The following code shows a portion of the T-SQL generated by EF Core when the Department name is updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

The preceding highlighted code shows the `WHERE` clause containing `RowVersion`. If the DB `RowVersion` doesn't equal the `RowVersion` parameter (`@p2`), no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

```
SET NOCOUNT ON;
UPDATE [Department] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [RowVersion] = @p2;
SELECT [RowVersion]
FROM [Department]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT returns the number of rows affected by the last statement. In no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

You can see the T-SQL EF Core generates in the output window of Visual Studio.

Update the DB

Adding the `RowVersion` property changes the DB model, which requires a migration.

Build the project. Enter the following in a command window:

```
dotnet ef migrations add RowVersion
dotnet ef database update
```

The preceding commands:

- Adds the *Migrations/{time stamp}_RowVersion.cs* migration file.
- Updates the *Migrations/SchoolContextModelSnapshot.cs* file. The update adds the following highlighted code to the `BuildModel` method:

  ```
  
  ```

- Runs migrations to update the DB.

Scaffold the Departments model

- Visual Studio
- Visual Studio Code

Follow the instructions in Scaffold the student model and use `Department` for the model class.

The preceding command scaffolds the `Department` model. Open the project in Visual Studio.

Build the project.

Update the Departments Index page

The scaffolding engine created a `RowVersion` column for the Index page, but that field shouldn't be displayed. In this tutorial, the last byte of the `RowVersion` is displayed to help understand concurrency. The last byte isn't guaranteed to be unique. A real app wouldn't display `RowVersion` or the last byte of `RowVersion`.

Update the Index page:

- Replace Index with Departments.
- Replace the markup containing `RowVersion` with the last byte of `RowVersion`.
- Replace FirstMidName with FullName.

The following markup shows the updated page:

```
@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Name)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].Budget)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.Department[0].StartDate)
                </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Administrator)
            </th>
            <th>
                RowVersion
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Department) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Budget)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.StartDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Administrator.FullName)
            </td>
            <td>
                @item.RowVersion[7]
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.DepartmentID">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.DepartmentID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Update the Edit page model

Update *Pages\Departments\Edit.cshtml.cs* with the following code:

```
using ContosoUniversity.Data;
```

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            // null means Department was deleted by another user.
            if (departmentToUpdate == null)
            {
                return HandleDeletedDepartment();
            }

            // Update the RowVersion to the value when this entity was
            // fetched. If the entity has been updated after it was
            // fetched, RowVersion won't match the DB RowVersion and
            // a DbUpdateConcurrencyException is thrown.
            // A second postback will make them match, unless a new
            // concurrency issue happens.
            _context.Entry(departmentToUpdate)
                .Property("RowVersion").OriginalValue = Department.RowVersion;
```

```
            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
            {
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues = (Department)exceptionEntry.Entity;
                    var databaseEntry = exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable to save. " +
                            "The department was deleted by another user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await setDbErrorMessage(dbValues, clientValues, _context);

                    // Save the current RowVersion so next postback
                    // matches unless an new concurrency issue happens.
                    Department.RowVersion = (byte[])dbValues.RowVersion;
                    // Must clear the model error for the next postback.
                    ModelState.Remove("Department.RowVersion");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

        private IActionResult HandleDeletedDepartment()
        {
            var deletedDepartment = new Department();
            // ModelState contains the posted data because of the deletion error and will overide the
Department instance values when displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another user.");
            InstructorNameSL = new SelectList(_context.Instructors, "ID", "FullName",
Department.InstructorID);
            return Page();
        }

        private async Task setDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
```

```
            }
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in the database "
              + "have been displayed. If you still want to edit this record, click "
              + "the Save button again.");
        }
    }
}
```

To detect a concurrency issue, the OriginalValue is updated with the `rowVersion` value from the entity it was fetched. EF Core generates a SQL UPDATE command with a WHERE clause containing the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), a `DbUpdateConcurrencyException` exception is thrown.

```
public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    // null means Department was deleted by another user.
    if (departmentToUpdate == null)
    {
        return HandleDeletedDepartment();
    }

    // Update the RowVersion to the value when this entity was
    // fetched. If the entity has been updated after it was
    // fetched, RowVersion won't match the DB RowVersion and
    // a DbUpdateConcurrencyException is thrown.
    // A second postback will make them match, unless a new
    // concurrency issue happens.
    _context.Entry(departmentToUpdate)
        .Property("RowVersion").OriginalValue = Department.RowVersion;
```

In the preceding code, `Department.RowVersion` is the value when the entity was fetched. `OriginalValue` is the value in the DB when `FirstOrDefaultAsync` was called in this method.

The following code gets the client values (the values posted to this method) and the DB values:

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless an new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
```

The following code adds a custom error message for each column that has DB values different from what was posted to `OnPostAsync`:

```
private async Task setDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database "
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

The following highlighted code sets the `RowVersion` value to the new value retrieved from the DB. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

```
try
{
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
catch (DbUpdateConcurrencyException ex)
{
    var exceptionEntry = ex.Entries.Single();
    var clientValues = (Department)exceptionEntry.Entity;
    var databaseEntry = exceptionEntry.GetDatabaseValues();
    if (databaseEntry == null)
    {
        ModelState.AddModelError(string.Empty, "Unable to save. " +
            "The department was deleted by another user.");
        return Page();
    }

    var dbValues = (Department)databaseEntry.ToObject();
    await setDbErrorMessage(dbValues, clientValues, _context);

    // Save the current RowVersion so next postback
    // matches unless an new concurrency issue happens.
    Department.RowVersion = (byte[])dbValues.RowVersion;
    // Must clear the model error for the next postback.
    ModelState.Remove("Department.RowVersion");
}
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Edit page

Update *Pages/Departments/Edit.cshtml* with the following markup:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.RowVersion" />
            <div class="form-group">
                <label>RowVersion</label>
                @Model.Department.RowVersion[7]
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label"></label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label"></label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label"></label>
                <input asp-for="Department.StartDate" class="form-control" />
                <span asp-validation-for="Department.StartDate" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
                <select asp-for="Department.InstructorID" class="form-control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID" class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding markup:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `RowVersion` must be added so post back binds the value.
- Displays the last byte of `RowVersion` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

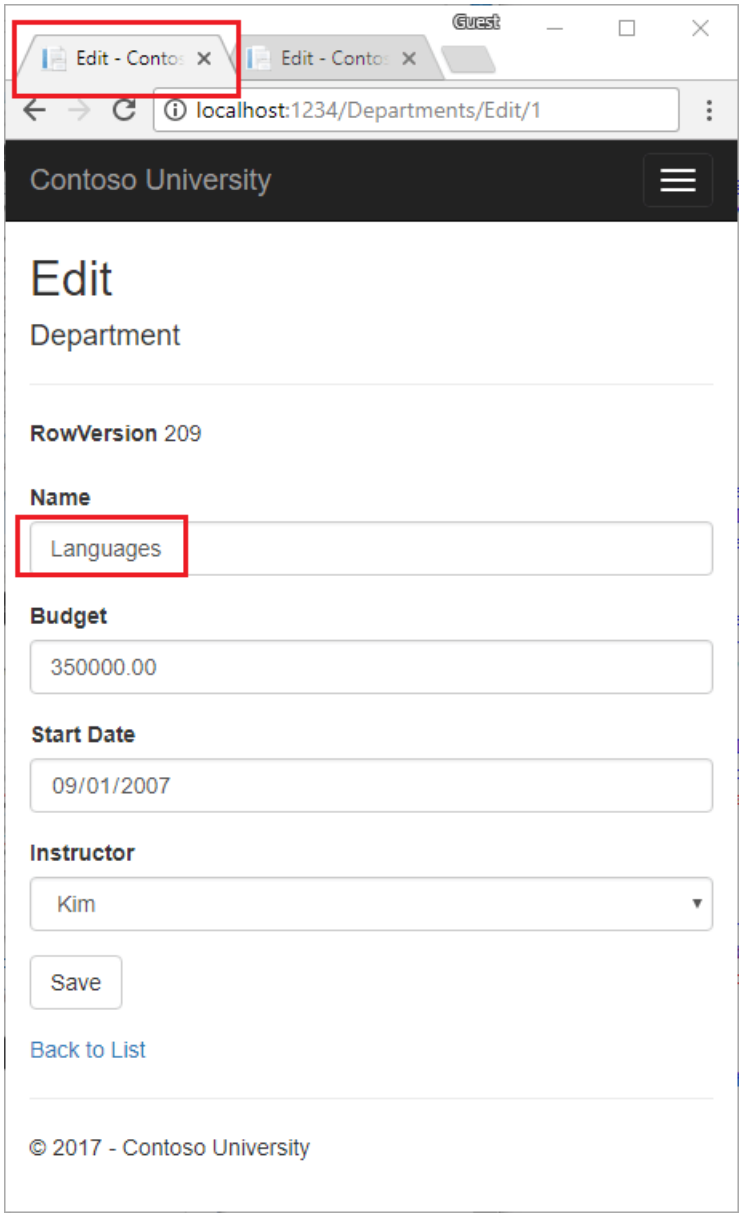
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

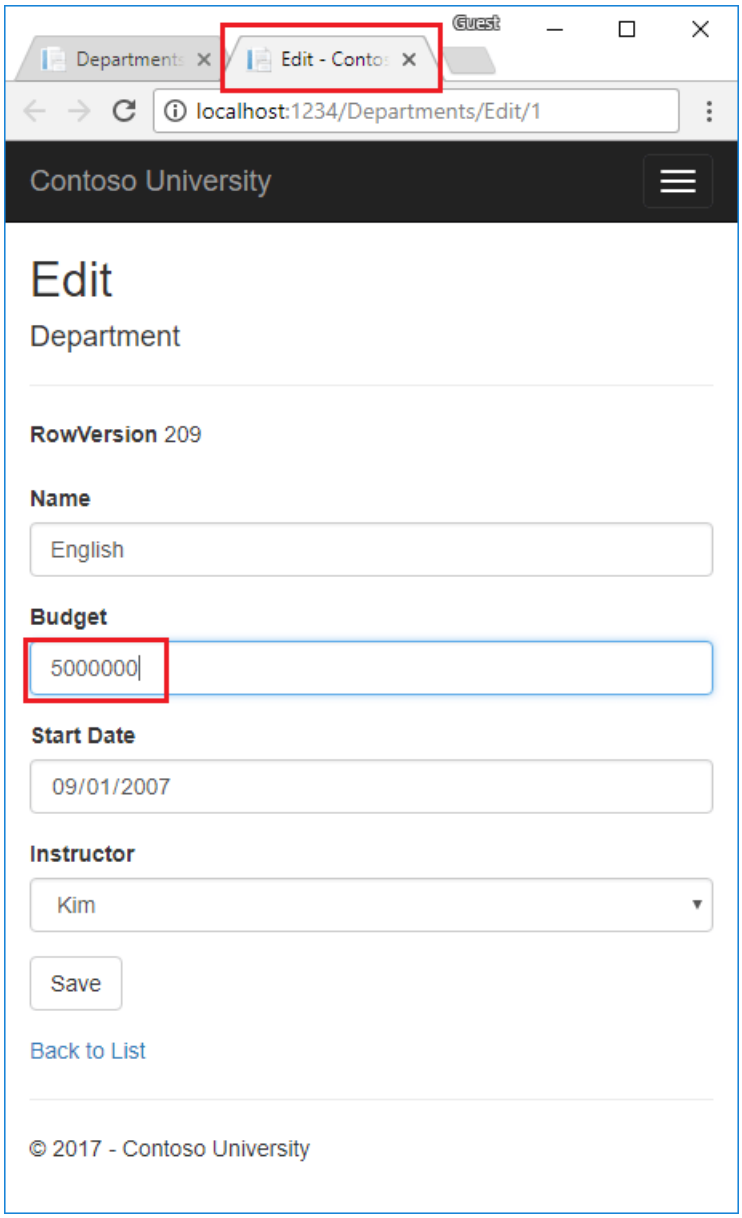
The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.

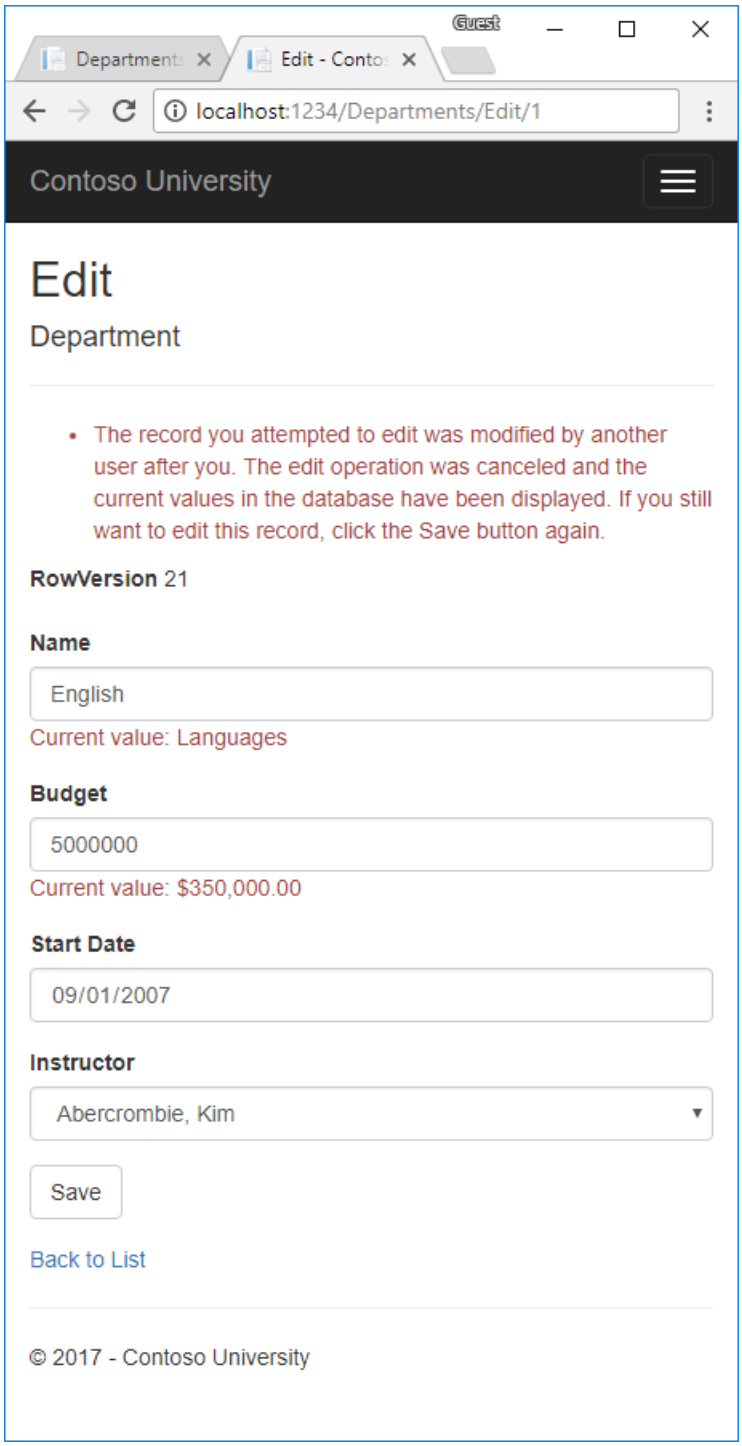


The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

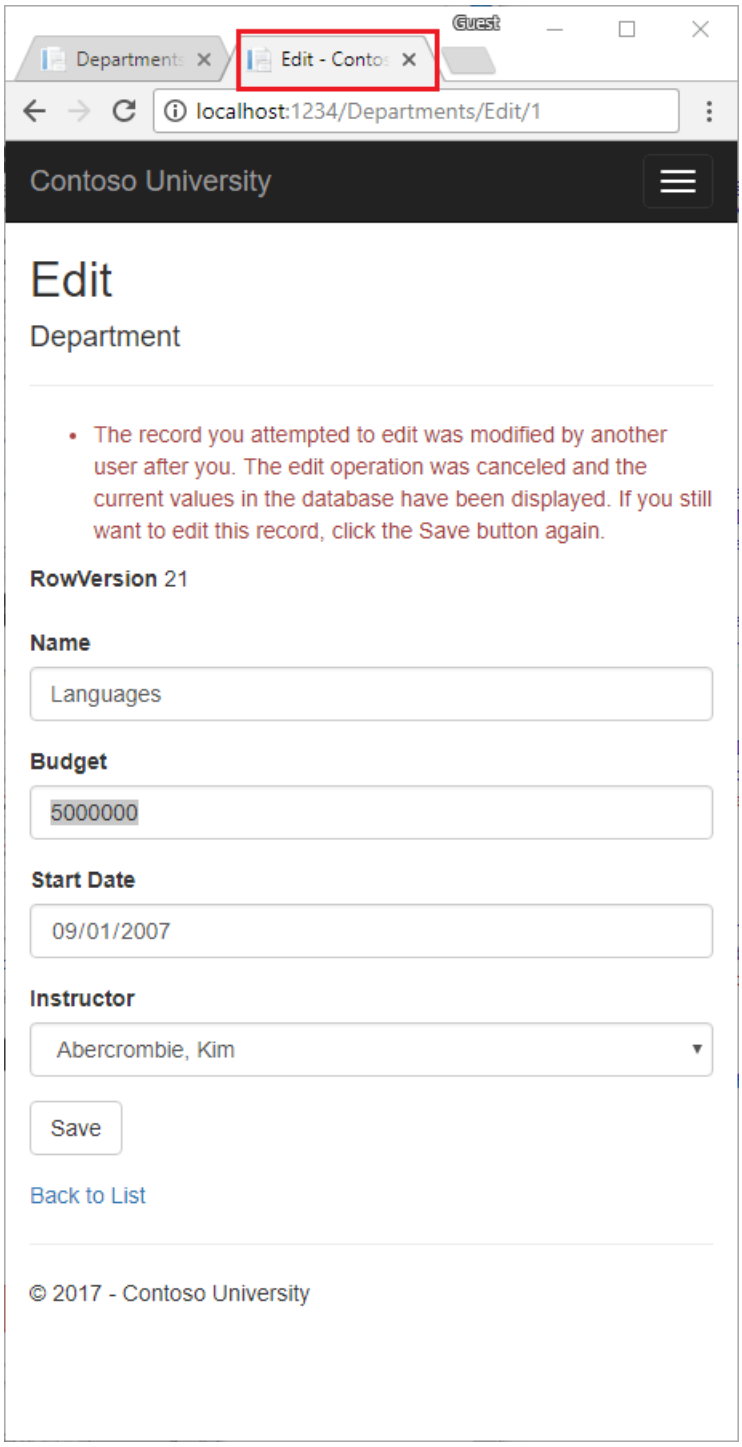
Change a different field in the second browser tab.



Click **Save**. You see error messages for all fields that don't match the DB values:



This browser window didn't intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page

Update the Delete page model with the following code:

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
```

```
        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool? concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                 return NotFound();
            }

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to delete "
                  + "was modified by another user after you selected delete. "
                  + "The delete operation was canceled and the current values in the "
                  + "database have been displayed. If you still want to delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.rowVersion value is from when the entity
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched. `Department.RowVersion` is the row version when the entity was fetched. When EF Core creates the SQL DELETE command, it includes a WHERE clause with `RowVersion`. If the SQL DELETE command results in zero rows affected:

- The `RowVersion` in the SQL DELETE command doesn't match `RowVersion` in the DB.
- A DbUpdateConcurrencyException exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete page

Update *Pages/Departments/Delete.cshtml* with the following code:

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.RowVersion)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.RowVersion[7])
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-page="./Index">Back to List</a>
        </div>
</form>
</div>
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.

- Replaces FirstMidName with FullName in the **Administrator** field.
- Changes `RowVersion` to display the last byte.
- Adds a hidden row version. `RowVersion` must be added so post back binds the value.

Test concurrency conflicts with the Delete page

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated rowVersion indicator. Note the updated rowVersion indicator, it's displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the DB. Clicking **Delete** deletes the entity, unless `RowVersion` has been updated.

See Inheritance on how to inherit a data model.

Additional resources

- Concurrency Tokens in EF Core
- Handle concurrency in EF Core
- YouTube version of this tutorial(Handling Concurrency Conflicts)
- YouTube version of this tutorial(Part 2)
- YouTube version of this tutorial(Part 3)

ASP.NET Core MVC with EF Core - tutorial series

9/22/2020 • 2 minutes to read • Edit Online

This tutorial has **not** been updated to ASP.NET Core 3.0. The Razor Pages version has been updated. For information on when this might be updated, see this GitHub issue.

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model that was introduced in ASP.NET Core 2.0. For new development, we recommend Razor Pages over MVC with controllers and views. There is a Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- A version available for ASP.NET Core 3.0

1. Get started
2. Create, Read, Update, and Delete operations
3. Sorting, filtering, paging, and grouping
4. Migrations
5. Create a complex data model
6. Reading related data
7. Updating related data
8. Handle concurrency conflicts
9. Inheritance
10. Advanced topics

Tutorial: Get started with EF Core in an ASP.NET MVC web app

9/22/2020 • 21 minutes to read • Edit Online

This tutorial has **not** been updated to ASP.NET Core 3.0. The Razor Pages version has been updated. Most of the code changes for the ASP.NET Core 3.0 and later version of this tutorial:

- Are in the *Startup.cs* and *Program.cs* files.
- Can be found in the Razor Pages version.

For information on when this might be updated, see this GitHub issue.

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model that was introduced in ASP.NET Core 2.0. For new development, we recommend Razor Pages over MVC with controllers and views. There is a Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- A version available for ASP.NET Core 3.0

The Contoso University sample web application demonstrates how to create ASP.NET Core 2.2 MVC web applications using Entity Framework (EF) Core 2.2 and Visual Studio 2017 or 2019.

The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This is the first in a series of tutorials that explain how to build the Contoso University sample application from scratch.

In this tutorial, you:

- Create an ASP.NET Core MVC web app
- Set up the site style
- Learn about EF Core NuGet packages
- Create the data model
- Create the database context
- Register the context for dependency injection
- Initialize the database with test data
- Create a controller and views
- View the database

Prerequisites

- .NET Core SDK 2.2
- Visual Studio 2019 with the following workloads:

- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload

Troubleshooting

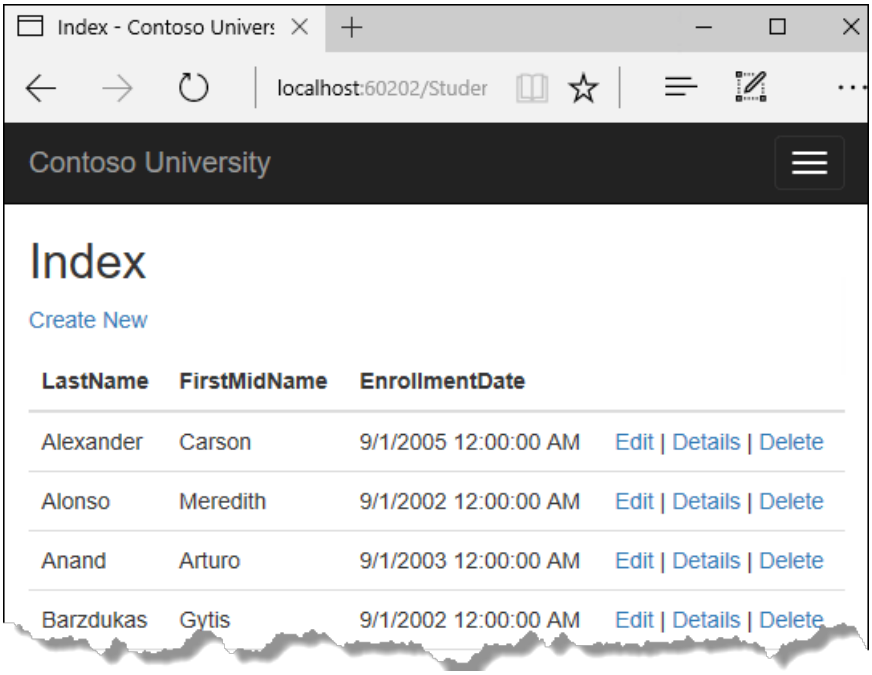
If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the completed project. For a list of common errors and how to solve them, see the Troubleshooting section of the last tutorial in the series. If you don't find what you need there, you can post a question to StackOverflow.com for ASP.NET Core or EF Core.

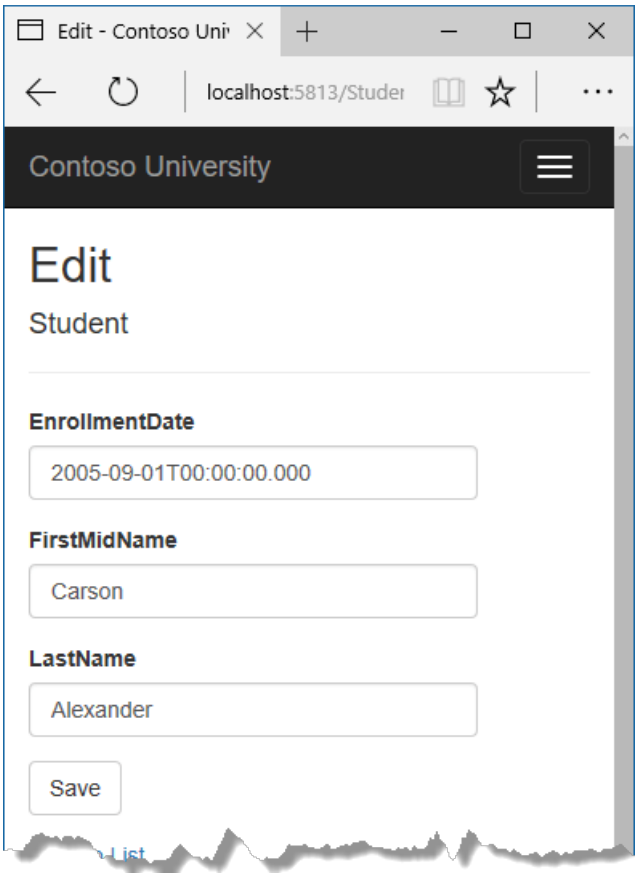
> **TIP**
> This is a series of 10 tutorials, each of which builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. Then if you run into problems, you can start over from the previous tutorial instead of going back to the beginning of the whole series.

Contoso University web app

The application you'll be building in these tutorials is a simple university web site.

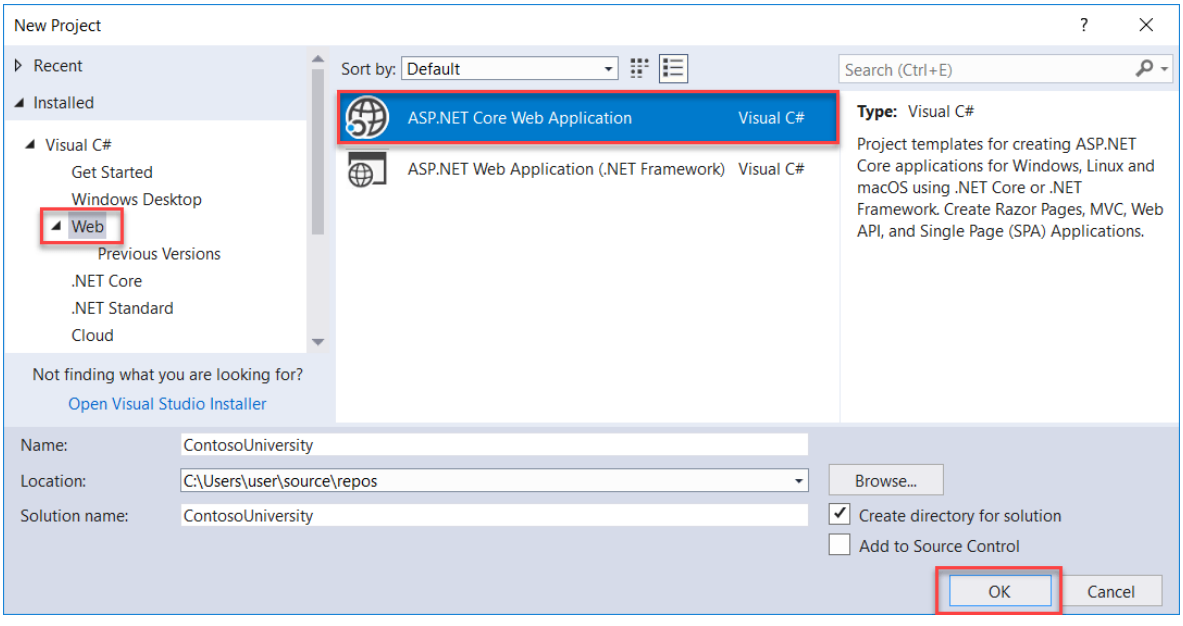
Users can view and update student, course, and instructor information. Here are a few of the screens you'll create.





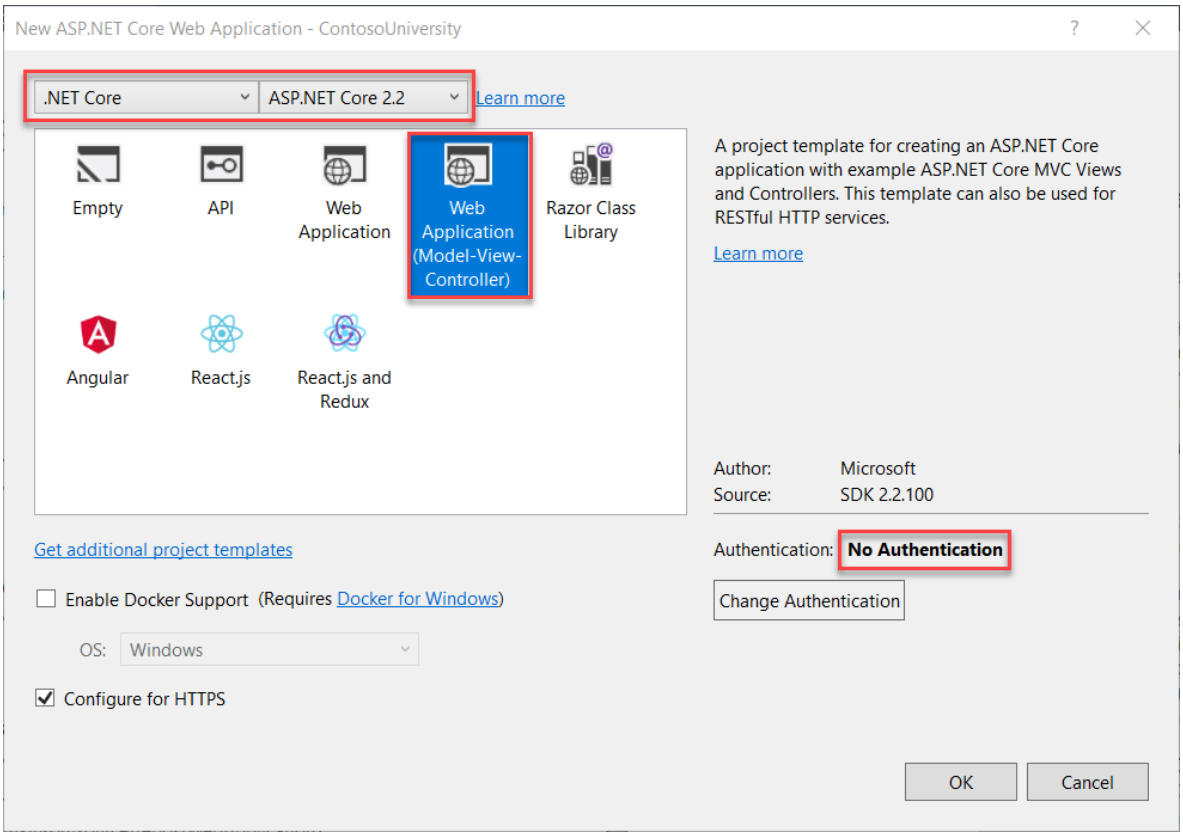
Create web app

- Open Visual Studio.
- From the **File** menu, select **New > Project**.
- From the left pane, select **Installed > Visual C# > Web**.
- Select the **ASP.NET Core Web Application** project template.
- Enter **ContosoUniversity** as the name and click **OK**.



- Wait for the **New ASP.NET Core Web Application** dialog to appear.
- Select **.NET Core**, **ASP.NET Core 2.2** and the **Web Application (Model-View-Controller)** template.

- Make sure **Authentication** is set to **No Authentication**.
- Select **OK**



Set up the site style

A few simple changes will set up the site menu, layout, and home page.

Open *Views/Shared/_Layout.cshtml* and make the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Add menu entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Privacy** menu entry.

The changes are highlighted.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute"
              crossorigin="anonymous"
              integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE="/>
    </environment>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
```

```
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">Contoso
University</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
action="About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Students" asp-
action="Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Courses" asp-
action="Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Instructors" asp-
action="Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Departments" asp-
action="Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - Contoso University - <a asp-area="" asp-controller="Home" asp-
action="Privacy">Privacy</a>
        </div>
    </footer>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">
        </script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-
```

```
bootstrap/4.1.3/js/bootstrap.bundle.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha256-E/V4cWE4qvAeO5MOhjtGtqDzPndRO1LBk8lJ/PR7CA4=">
        </script>
    </environment>
    <script src="~/js/site.js" asp-append-version="true"></script>

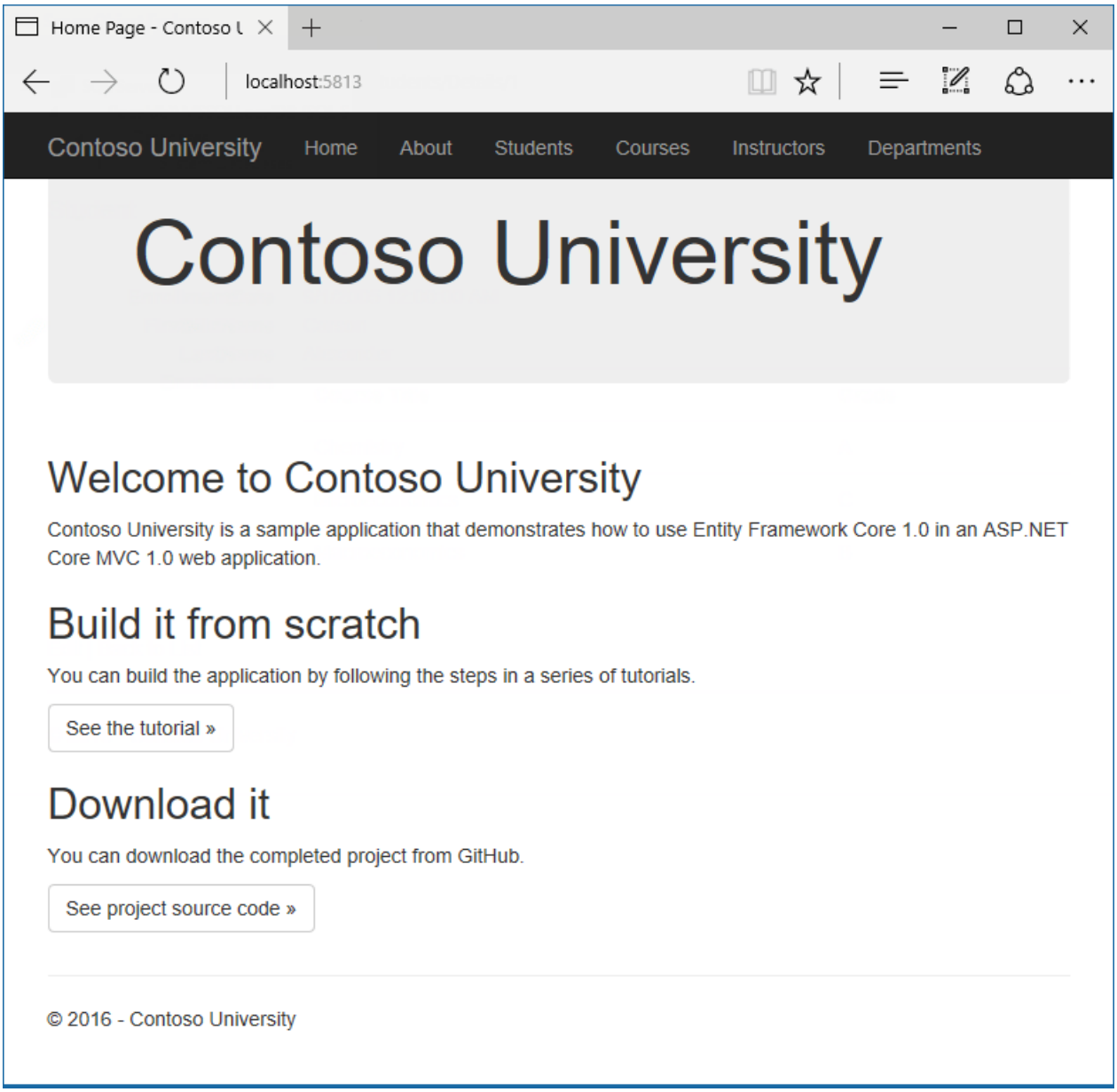
    @RenderSection("Scripts", required: false)
</body>
</html>
```

In *Views/Home/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this application:

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of tutorials.</p>
        <p><a class="btn btn-default" href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
href="https://github.com/dotnet/AspNetCore.Docs/tree/master/aspnetcore/data/ef-mvc/intro/samples/cu-
final">See project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu. You see the home page with tabs for the pages you'll create in these tutorials.



About EF Core NuGet packages

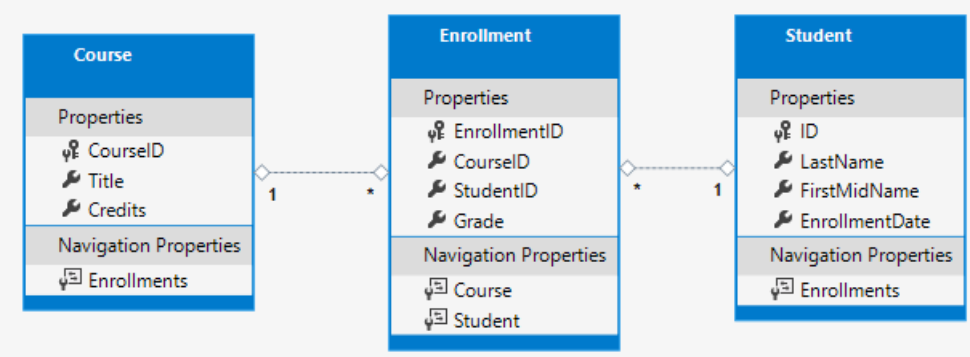
To add EF Core support to a project, install the database provider that you want to target. This tutorial uses SQL Server, and the provider package is Microsoft.EntityFrameworkCore.SqlServer. This package is included in the Microsoft.AspNetCore.App metapackage, so you don't need to reference the package.

The EF SQL Server package and its dependencies (`Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`) provide runtime support for EF. You'll add a tooling package later, in the Migrations tutorial.

For information about other database providers that are available for Entity Framework Core, see Database providers.

Create the data model

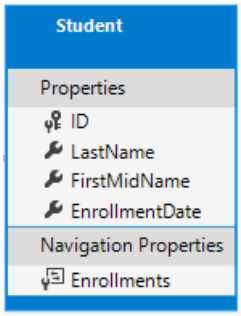
Next you'll create entity classes for the Contoso University application. You'll start with the following three entities.



There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections you'll create a class for each one of these entities.

The Student entity



In the *Models* folder, create a class file named *Student.cs* and replace the template code with the following code.

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

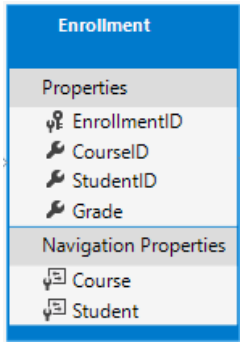
        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or `classnameID` as the primary key.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student entity` will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given Student row in the database has two related Enrollment rows (rows that contain that student's primary key value in their StudentID foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection<T>`. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The Enrollment entity



In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property will be the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself as you saw in the `Student` entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a later tutorial, you'll see how using ID without classname makes it easier to implement inheritance in the data model.

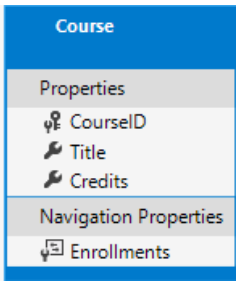
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a foreign key property if it's named `<navigation property name><primary key property name>` (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply `<primary key property name>` (for example, `CourseID` since the `Course` entity's primary key is `CourseID`).

The Course entity



In the *Models* folder, create *Course.cs* and replace the existing code with the following code:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the `DatabaseGenerated` attribute in a later tutorial in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the database context

The main class that coordinates Entity Framework functionality for a given data model is the database context class. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. In your code you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named *Data*.

In the *Data* folder create a new class file named *SchoolContext.cs*, and replace the template code with the following code:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

This code creates a `DbSet` property for each entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.

You could've omitted the `DbSet<Enrollment>` and `DbSet<Course>` statements and it would work the same. The Entity Framework would include them implicitly because the `Student` entity references the `Enrollment` entity and the `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural (Students rather than Student), but developers disagree about whether table names should be pluralized or not. For these tutorials you'll override the default behavior by specifying singular table names in the DbContext. To do that, add the following highlighted code after the last DbSet property.

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Register the SchoolContext

ASP.NET Core implements dependency injection by default. Services (such as the EF database context) are registered with dependency injection during application startup. Components that require these services (such as MVC controllers) are provided these services via constructor parameters. You'll see the controller constructor code that gets a context instance later in this tutorial.

To register `SchoolContext` as a service, open *Startup.cs*, and add the highlighted lines to the `ConfigureServices` method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<SchoolContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddMvc();
}
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the ASP.NET Core configuration system reads the connection string from the *appsettings.json* file.

Add `using` statements for `ContosoUniversity.Data` and `Microsoft.EntityFrameworkCore` namespaces, and then build the project.

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Http;
```

Open the *appsettings.json* file and add a connection string as shown in the following example.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

SQL Server Express LocalDB

The connection string specifies a SQL Server LocalDB database. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for application development, not production use. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB creates *.mdf* database files in the `C:/Users/<user>` directory.

Initialize DB with test data

The Entity Framework will create an empty database for you. In this section, you write a method that's called after the database is created in order to populate it with test data.

Here you'll use the `EnsureCreated` method to automatically create the database. In a later tutorial you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the *Data* folder, create a new class file named *DbInitializer.cs* and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.Parse("2005-09-
01")},
            new Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Parse("2002-09-
01")},
            new Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse("2003-09-01")},
            new Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Parse("2002-09-
01")},
            new Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002-09-01")},
            new Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Parse("2001-09-01")},
            new Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse("2003-09-01")},
            new Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Parse("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
```

```
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollments.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database, and if not, it assumes the database is new and needs to be seeded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

In *Program.cs*, modify the `Main` method to do the following on application startup:

- Get a database context instance from the dependency injection container.
- Call the seed method, passing to it the context.
- Dispose the context when the seed method is done.

```
public static void Main(string[] args)
{
     var host = CreateWebHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the database.");
        }
    }

    host.Run();
}
```

Add `using` statements:

```
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Data;
```

In older tutorials, you may see similar code in the `Configure` method in *Startup.cs*. We recommend that you use the `Configure` method only to set up the request pipeline. Application startup code belongs in the `Main` method.

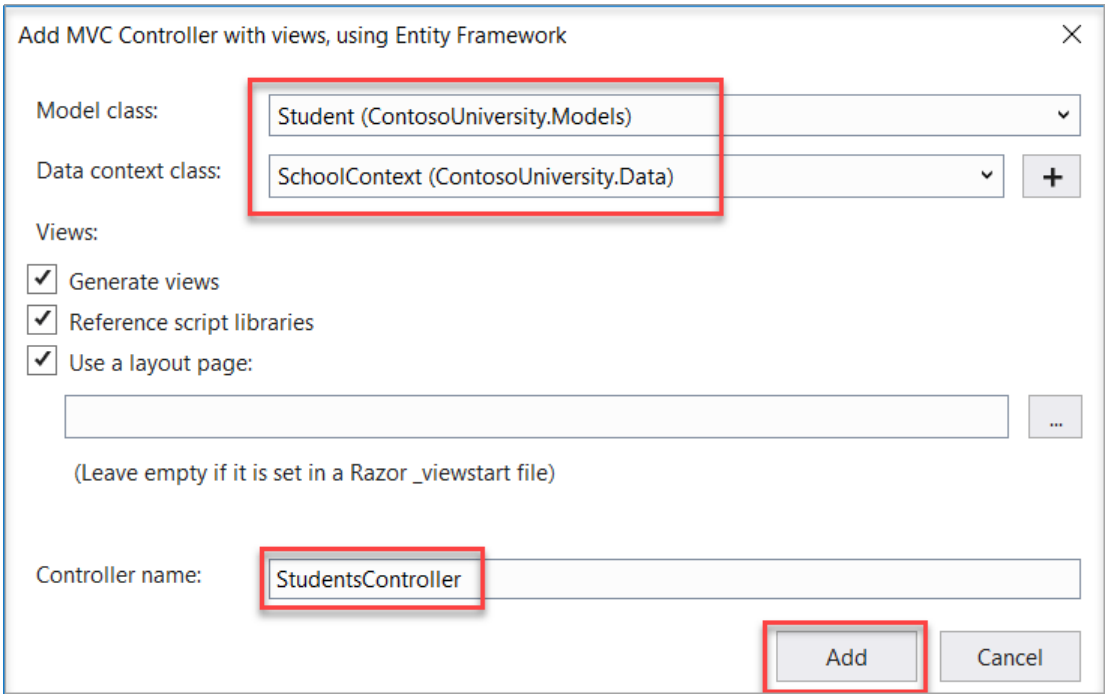
Now the first time you run the application, the database will be created and seeded with test data. Whenever you change your data model, you can delete the database, update your seed method, and start afresh with a new database the same way. In later tutorials, you'll see how to modify the database when the data model changes, without deleting and re-creating it.

Create controller and views

Next, you'll use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding. Scaffolding differs from code generation in that the scaffolded code is a starting point that you can modify to suit your own requirements, whereas you typically don't modify generated code. When you need to customize generated code, you use partial classes or you regenerate the code when things change.

- Right-click the **Controllers** folder in **Solution Explorer** and select **Add > New Scaffolded Item**.
- In the **Add Scaffold** dialog box:
  - Select **MVC controller with views, using Entity Framework**.
  - Click **Add**. The **Add MVC Controller with views, using Entity Framework** dialog box appears.



  - In **Model class** select **Student**.
  - In **Data context class** select **SchoolContext**.
  - Accept the default **StudentsController** as the name.
  - Click **Add**.

  When you click **Add**, the Visual Studio scaffolding engine creates a *StudentsController.cs* file and a set of views (*.cshtml* files) that work with the controller.

(The scaffolding engine can also create the database context for you if you don't create it manually first as you did earlier for this tutorial. You can specify a new context class in the **Add Controller** box by clicking the plus sign to the right of **Data context class**. Visual Studio will then create your `DbContext` class as well as the controller and views.)

You'll notice that the controller takes a `SchoolContext` as a constructor parameter.

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
```

ASP.NET Core dependency injection takes care of passing an instance of `SchoolContext` into the controller. You configured that in the *Startup.cs* file earlier.

The controller contains an `Index` action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the `Students` property of the database context instance:

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

You'll learn about the asynchronous programming elements in this code later in the tutorial.

The *Views/Students/Index.cshtml* view displays this list in a table:

```
@model IEnumerable<ContosoUniversity.Models.Student>

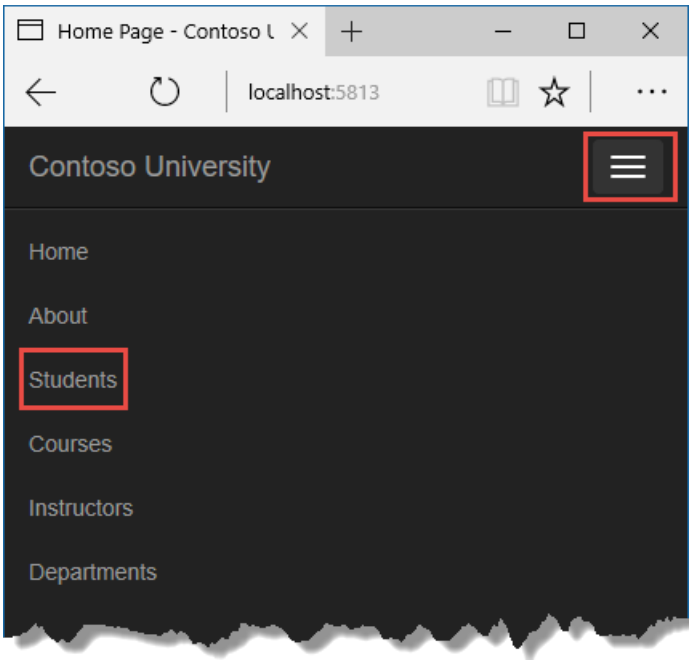
@{
    ViewData["Title"] = "Index";
}

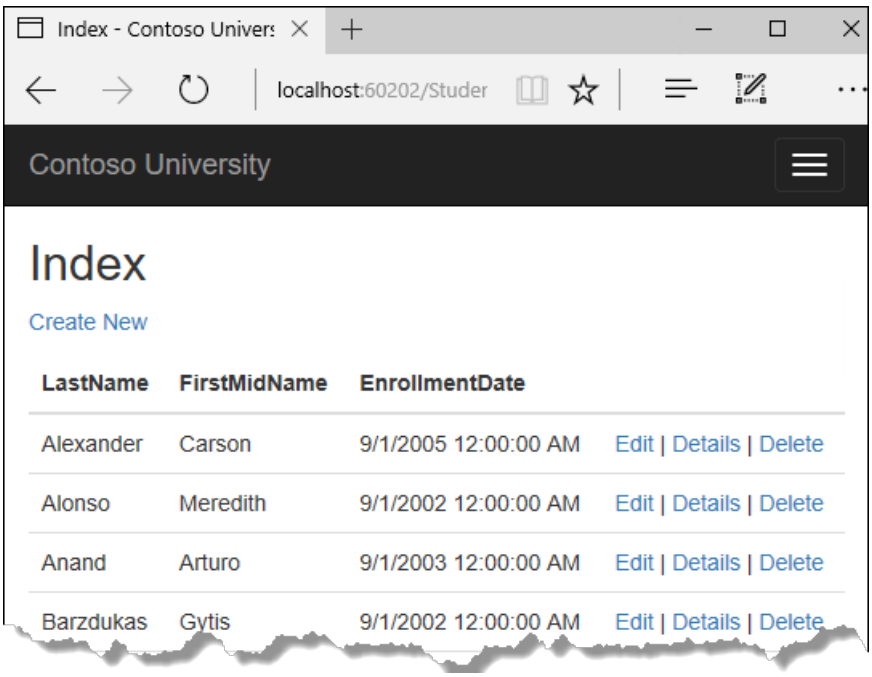
<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.LastName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.EnrollmentDate)
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Press CTRL+F5 to run the project or choose **Debug > Start Without Debugging** from the menu.

Click the Students tab to see the test data that the `DbInitializer.Initialize` method inserted. Depending on how narrow your browser window is, you'll see the `Students` tab link at the top of the page or you'll have to click the navigation icon in the upper right corner to see the link.





View the database

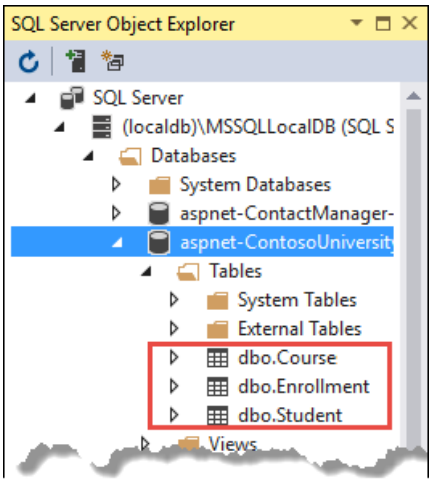
When you started the application, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database and so it created one, then the remainder of the `Initialize` method code populated the database with data. You can use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio.

Close the browser.

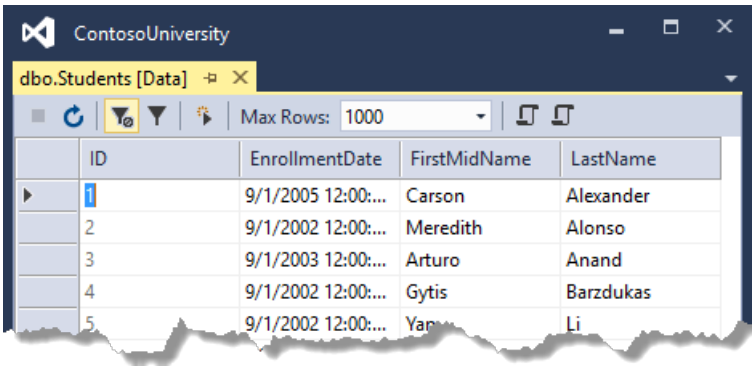
If the SSOX window isn't already open, select it from the **View** menu in Visual Studio.

In SSOX, click **(localdb)\MSSQLLocalDB > Databases**, and then click the entry for the database name that's in the connection string in your *appsettings.json* file.

Expand the **Tables** node to see the tables in your database.



Right-click the **Student** table and click **View Data** to see the columns that were created and the rows that were inserted into the table.



The *.mdf* and *.ldf* database files are in the *C:\Users\<yourusername>* folder.

Because you're calling `EnsureCreated` in the initializer method that runs on app start, you could now make a change to the `Student` class, delete the database, run the application again, and the database would automatically be re-created to match your change. For example, if you add an `EmailAddress` property to the `Student` class, you'll see a new `EmailAddress` column in the re-created table.

Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of conventions, or assumptions that the Entity Framework makes.

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named ID or classnameID are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named *<navigation property name><primary key property name>* (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply *<primary key property name>* (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

Conventional behavior can be overridden. For example, you can explicitly specify table names, as you saw earlier in this tutorial. And you can set column names and set any property as primary key or foreign key, as you'll see in a later tutorial in this series.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the `async` keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that's returned.
- The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when you are writing asynchronous code that uses the Entity Framework:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes, for example, `ToListAsync`, `SingleOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include, for example, statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF context isn't thread safe: don't try to do multiple operations in parallel. When you call any async EF method, always use the `await` keyword.
- If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

For more information about asynchronous programming in .NET, see Async Overview.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Created ASP.NET Core MVC web app

- Set up the site style
- Learned about EF Core NuGet packages
- Created the data model
- Created the database context
- Registered the SchoolContext
- Initialized DB with test data
- Created controller and views
- Viewed the database

In the following tutorial, you'll learn how to perform basic CRUD (create, read, update, delete) operations.

Advance to the next tutorial to learn how to perform basic CRUD (create, read, update, delete) operations.

Implement basic CRUD functionality

Tutorial: Implement CRUD Functionality - ASP.NET MVC with EF Core

9/22/2020 • 19 minutes to read • Edit Online

In the previous tutorial, you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial, you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

> **NOTE**
>
> It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about repositories with EF, see the last tutorial in this series.

In this tutorial, you:

- Customize the Details page
- Update the Create page
- Update the Edit page
- Update the Delete page
- Close database connections

Prerequisites

- Get started with EF Core and ASP.NET Core MVC

Customize the Details page

The scaffolded code for the Students Index page left out the `Enrollments` property, because that property holds a collection. In the **Details** page, you'll display the contents of the collection in an HTML table.

In *Controllers/StudentsController.cs*, the action method for the Details view uses the `SingleOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls `Include`. `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .Include(s => s.Enrollments)
            .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. You'll learn more about these methods in the read related data tutorial.

The `AsNoTracking` method improves performance in scenarios where the entities returned won't be updated in the current context's lifetime. You'll learn more about `AsNoTracking` at the end of this tutorial.

Route data

The key value that's passed to the `Details` method comes from *route data*. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the `Index` method `id` parameter if you pass it as a query string value:

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the `id` parameter matches the default route, so `id` is added to the route data.

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, `studentID` doesn't match a parameter in the default route, so it's added as a query string.

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

For more information about tag helpers, see Tag Helpers in ASP.NET Core.

Add enrollments to the Details view

Open *Views/Students/Details.cshtml*. Each field is displayed using `DisplayNameFor` and `DisplayFor` helpers, as shown in the following example:

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd class="col-sm-10">
    @Html.DisplayFor(model => model.LastName)
</dd>
```

After the last field and immediately before the closing `</dl>` tag, add the following code to display a list of enrollments:

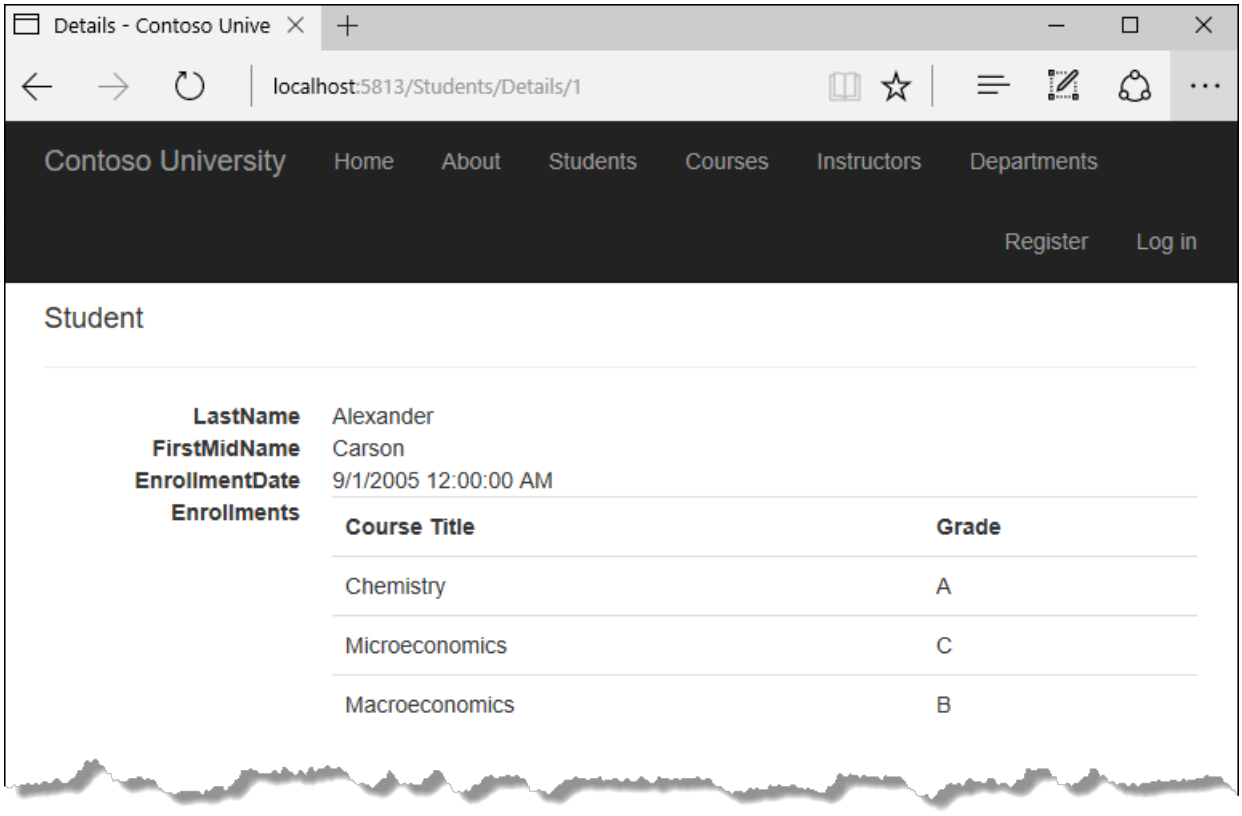
```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd class="col-sm-10">
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. You see the list of courses and

grades for the selected student:



Update the Create page

In *StudentsController.cs*, modify the HttpPost `Create` method by adding a try-catch block and removing ID from the `Bind` attribute.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

This code adds the Student entity created by the ASP.NET Core MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET Core MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed `ID` from the `Bind` attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user doesn't set the ID value.

Other than the `Bind` attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from `DbUpdateException` is caught while the changes are being saved, a generic error message is displayed. `DbUpdateException` exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in Monitoring and Telemetry (Building Real-World Cloud Apps with Azure).

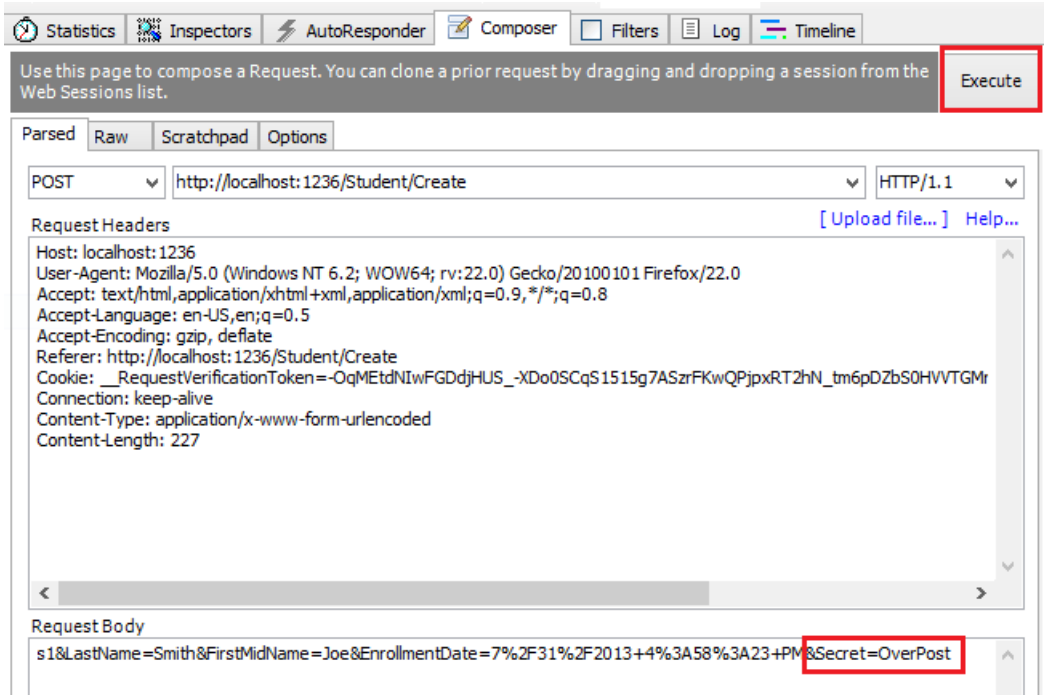
The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the FormTagHelper and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Security note about overposting

The `Bind` attribute that the scaffolded code includes on the `Create` method is one way to protect against overposting in create scenarios. For example, suppose the Student entity includes a `Secret` property that you don't want this web page to set.

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a Student instance, the model binder would pick up that `Secret` form value and use it to create the Student entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to set that property.

You can prevent overposting in edit scenarios by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That's the method used in these tutorials.

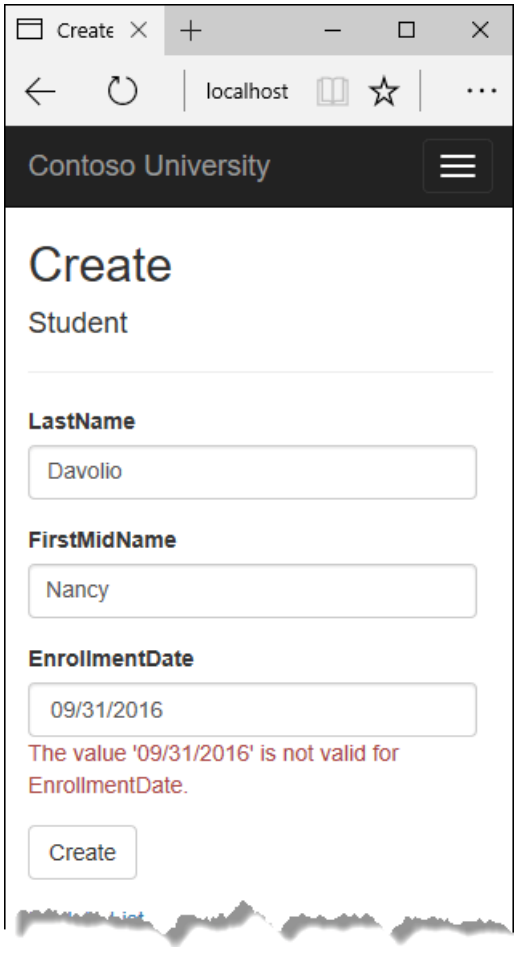
An alternative way to prevent overposting that's preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use `_context.Entry` on the entity instance to set its state to `Unchanged`, and then set `Property("PropertyName").IsModified` to true on each entity property that's included in the view model. This method works in both edit and create scenarios.

Test the Create page

The code in *Views/Students/Create.cshtml* uses `label`, `input`, and `span` (for validation messages) tag helpers for each field.

Run the app, select the **Students** tab, and click **Create New**.

Enter names and a date. Try entering an invalid date if your browser lets you do that. (Some browsers force you to use a date picker.) Then click **Create** to see the error message.



This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the `Create` method.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

Update the Edit page

In *StudentController.cs*, the HttpGet `Edit` method (the one without the `HttpPost` attribute) uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the `Details` method. You don't need to change this method.

Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.FirstOrDefaultAsync(s => s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}
```

These changes implement a security best practice to prevent overposting. The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a `Modified` flag. That code isn't recommended for many scenarios because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter.

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity based on user input in the posted form data. The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

As a best practice to prevent overposting, the fields that you want to be updateable by the **Edit** page are declared in the `TryUpdateModel` parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the HttpPost `Edit` method is the same as the HttpGet `Edit` method; therefore you've renamed the method `EditPost`.

Alternative HttpPost Edit code: Create and attach

The recommended HttpPost edit code ensures that only changed columns get updated and preserves data in properties that you don't want included for model binding. However, the read-first approach requires an extra database read, and can result in more complex code for handling concurrency conflicts. An alternative is to attach an entity created by the model binder to the EF context and mark it as modified. (Don't update your project with this code, it's only shown to illustrate an optional approach.)

```
public async Task<IActionResult> Edit(int id, [Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student
student)
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}
```

You can use this approach when the web page UI includes all of the fields in the entity and can update any of them.

The scaffolded code uses the create-and-attach approach but only catches `DbUpdateConcurrencyException` exceptions and returns 404 error codes. The example shown catches any database update exception and displays an error message.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the `Add` method, that entity's state is set to `Added`. Then when you call the `SaveChanges` method, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`. The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.
- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. You read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the HttpPost `Edit` action method is called, a new web request is made and you have a

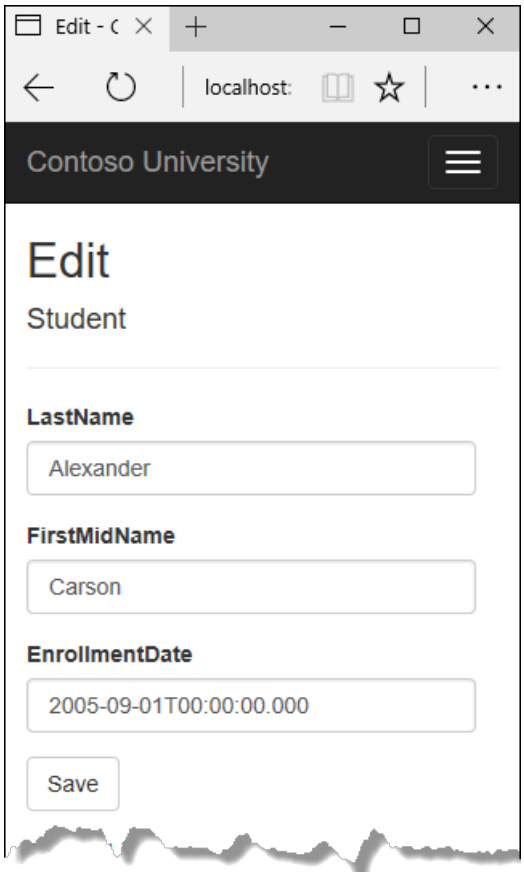
new instance of the `DbContext`. If you re-read the entity in that new context, you simulate desktop processing.

But if you don't want to do the extra read operation, you have to use the entity object created by the model binder. The simplest way to do this is to set the entity state to Modified as is done in the alternative HttpPost Edit code shown earlier. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want to avoid the read-first approach, but you also want the SQL UPDATE statement to update only the fields that the user actually changed, the code is more complex. You have to save the original values in some way (such as by using hidden fields) so that they're available when the HttpPost `Edit` method is called. Then you can create a Student entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`.

Test the Edit page

Run the app, select the **Students** tab, then click an **Edit** hyperlink.



Change some of the data and click **Save**. The **Index** page opens and you see the changed data.

Update the Delete page

In *StudentController.cs*, the template code for the HttpGet `Delete` method uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that's called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the HttpPost `Delete` method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the HttpPost `Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the HttpPost Delete method calls the HttpGet Delete method, passing it a

parameter that indicates that an error has occurred. The HttpGet Delete method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

Replace the HttpGet `Delete` action method with the following code, which manages error reporting.

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the HttpGet `Delete` method is called without a previous failure. When it's called by the HttpPost `Delete` method in response to a database update error, the parameter is true and an error message is passed to the view.

The read-first approach to HttpPost Delete

Replace the HttpPost `Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students.FindAsync(id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

This code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated.

The create-and-attach approach to HttpPost Delete

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query by instantiating a Student entity using only the primary key value and then setting the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity. (Don't put this code in your project; it's here just to illustrate an alternative.)

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id, saveChangesError = true });
    }
}
```

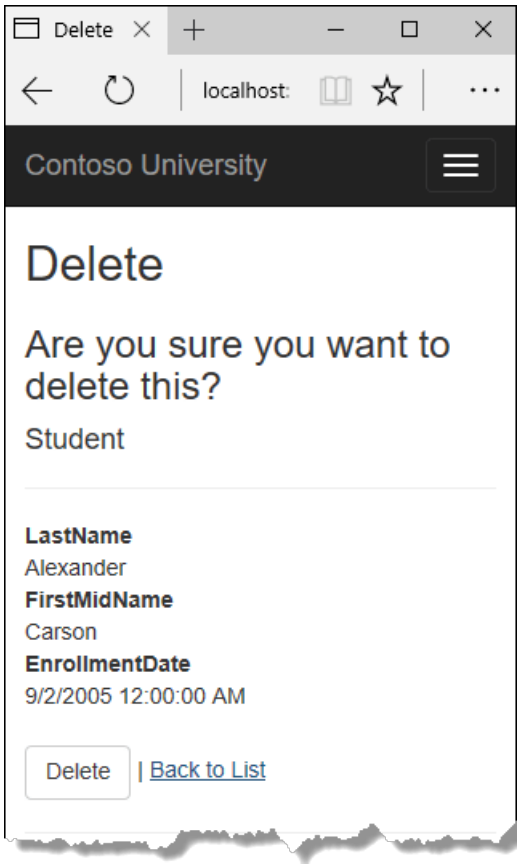
If the entity has related data that should also be deleted, make sure that cascade delete is configured in the database. With this approach to entity deletion, EF might not realize there are related entities to be deleted.

Update the Delete view

In *Views/Student/Delete.cshtml*, add an error message between the h2 heading and the h3 heading, as shown in the following example:

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the app, select the **Students** tab, and click a **Delete** hyperlink:



Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Close database connections

To free up the resources that a database connection holds, the context instance must be disposed as soon as possible when you are done with it. The ASP.NET Core built-in dependency injection takes care of that task for you.

In *Startup.cs*, you call the AddDbContext extension method to provision the `DbContext` class in the ASP.NET Core DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handle transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or they all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by calling the `AsNoTracking` method. Typical scenarios in

which you might want to do that include the following:

- During the context lifetime you don't need to update any entities, and you don't need EF to automatically load navigation properties with entities retrieved by separate queries. Frequently these conditions are met in a controller's HttpGet action methods.
- You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.
- You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see Tracking vs. No-Tracking.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Customized the Details page
- Updated the Create page
- Updated the Edit page
- Updated the Delete page
- Closed database connections

Advance to the next tutorial to learn how to expand the functionality of the **Index** page by adding sorting, filtering, and paging.

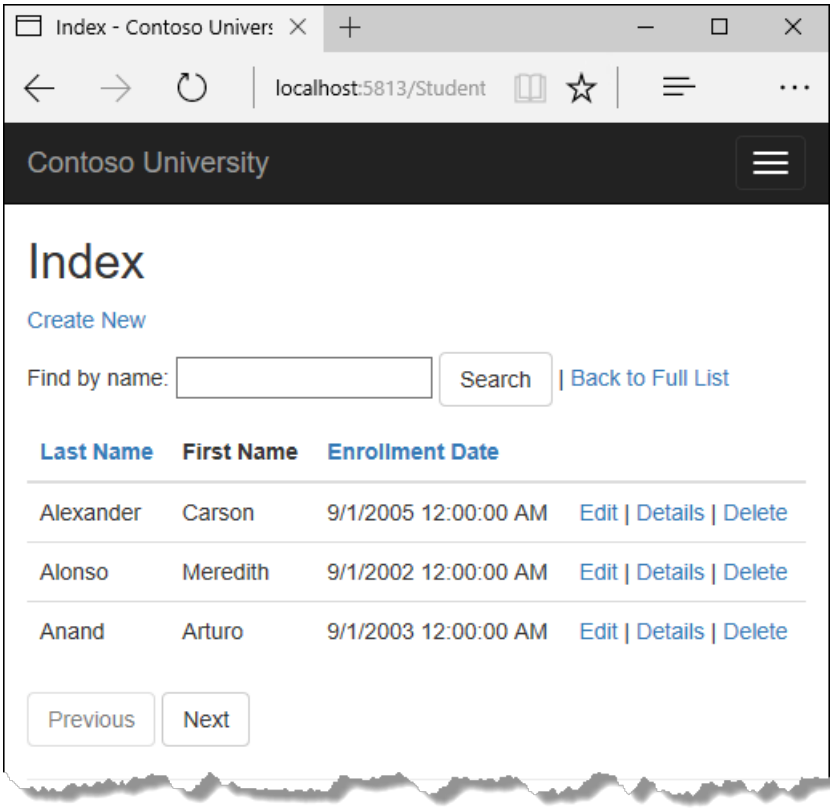
Next: Sorting, filtering, and paging

Tutorial: Add sorting, filtering, and paging - ASP.NET MVC with EF Core

9/22/2020 • 14 minutes to read • Edit Online

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for Student entities. In this tutorial you'll add sorting, filtering, and paging functionality to the Students Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



In this tutorial, you:

- Add column sort links
- Add a Search box
- Add paging to Students Index
- Add paging to Index method
- Add paging links
- Create an About page

Prerequisites

- Implement CRUD Functionality

Add column sort links

To add sorting to the Student Index page, you'll change the `Index` method of the Students controller and add

code to the Student Index view.

Add sorting Functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code:

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewData` elements (NameSortParm and DateSortParm) are used by the view to configure the column heading hyperlinks with the appropriate query string values.

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, NameSortParm should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

CURRENT SORT ORDER	LAST NAME HYPERLINK	DATE HYPERLINK
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an `IQueryable` variable before the switch statement, modifies it in the switch statement, and calls the `ToListAsync` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query isn't executed until you convert the `IQueryable` object into a collection by calling a method such as `ToListAsync`. Therefore, this code results in a single query that's not executed until the `return View` statement.

This code could get verbose with a large number of columns. The last tutorial in this series shows how to write code that lets you pass the name of the `OrderBy` column in a string variable.

Add column heading hyperlinks to the Student Index view

Replace the code in *Views/Students/Index.cshtml*, with the following code to add column heading hyperlinks. The changed lines are highlighted.

```
@model IEnumerable<ContosoUniversity.Models.Student>

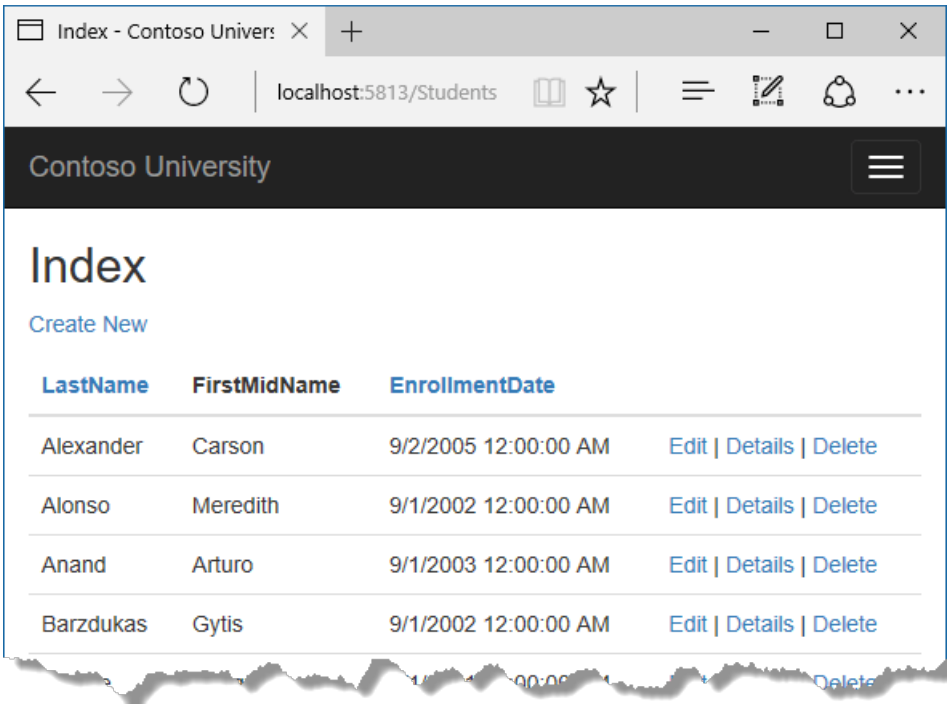
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model => model.LastName)</a>
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model => model.EnrollmentDate)</a>
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

This code uses the information in `ViewData` properties to set up hyperlinks with the appropriate query string values.

Run the app, select the **Students** tab, and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Add a Search box

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code (the changes are highlighted).

```
public async Task<IActionResult> Index(string sortOrder, string searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

You've added a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

> **NOTE**
>
> Here you are calling the `Where` method on an `IQueryable` object, and the filter will be processed on the server. In some scenarios you might be calling the `Where` method as an extension method on an in-memory collection. (For example, suppose you change the reference to `_context.Students` so that instead of an EF `DbSet` it references a repository method that returns an `IEnumerable` collection.) The result would normally be the same but in some cases may be different.
>
> For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there's a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

In *Views/Student/Index.cshtml*, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

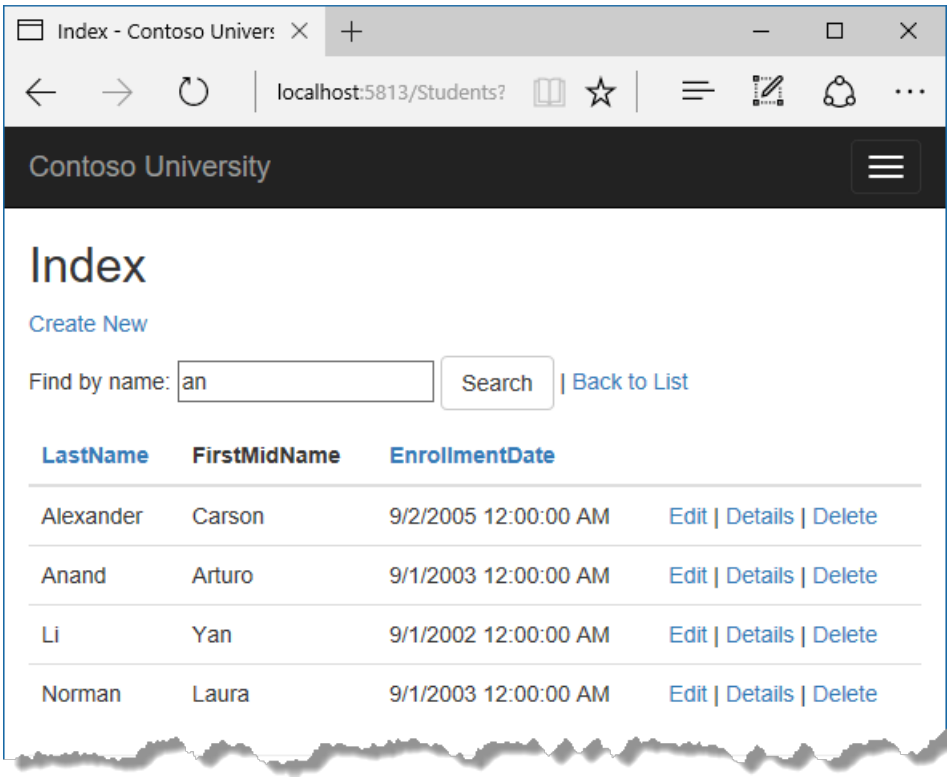
```
<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
```

This code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action doesn't result in an update.

Run the app, select the **Students** tab, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

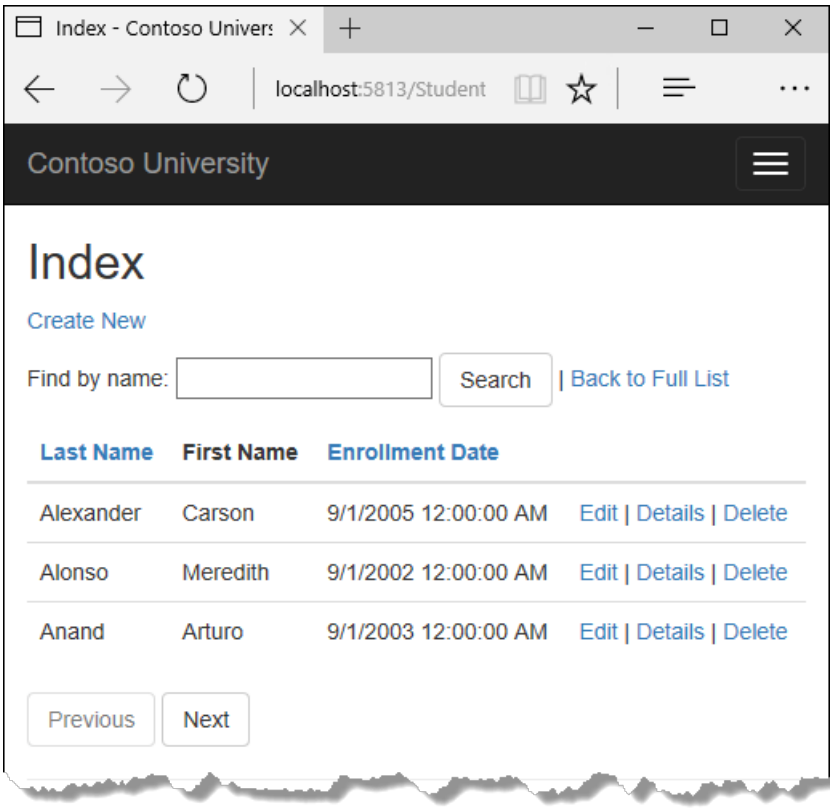
```
http://localhost:5813/Students?SearchString=an
```

If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding `method="get"` to the `form` tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the **Search** box. You'll fix that in the next section.

Add paging to Students Index

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs`, and then replace the template code with the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage
        {
            get
            {
                return (PageIndex > 1);
            }
        }

        public bool HasNextPage
        {
            get
            {
                return (PageIndex < TotalPages);
            }
        }

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int pageIndex, int
pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) * pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in this code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it will return a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` can be used to enable or disable **Previous** and **Next** paging buttons.

A `CreateAsync` method is used instead of a constructor to create the `PaginatedList<T>` object because constructors can't run asynchronous code.

Add paging to Index method

In *StudentsController.cs*, replace the `Index` method with the following code.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
        pageNumber = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1,
pageSize));
}
```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The `ViewData` element named CurrentSort provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The `ViewData` element named CurrentFilter provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the `searchString` parameter isn't null.

```
if (searchString != null)
{
    pageNumber = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the `Index` method, the `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

```
return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1, pageSize));
```

The `PaginatedList.CreateAsync` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(pageNumber ?? 1)` means return the value of `pageNumber` if it has a value, or return 1 if `pageNumber` is null.

Add paging links

In *Views/Students/Index.cshtml*, replace the existing code with the following code. The changes are highlighted.

```
@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString" value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
```

```
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex + 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

The `@model` statement at the top of the page specifies that the view now gets a `PaginatedList<T>` object instead of a `List<T>` object.

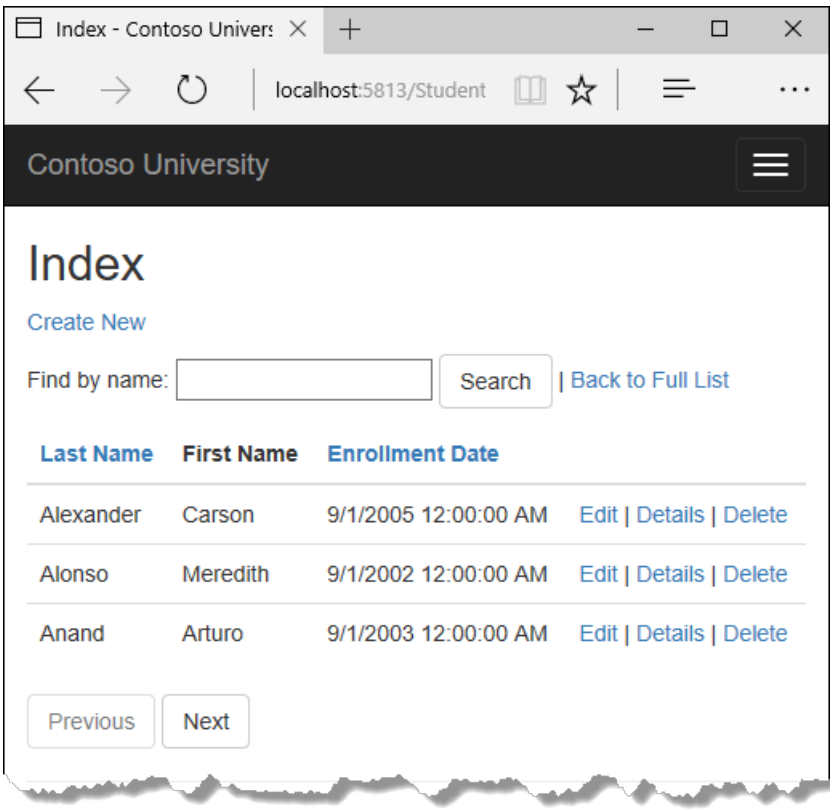
The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter
="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

The paging buttons are displayed by tag helpers:

```
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
   Previous
</a>
```

Run the app and go to the Students page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

Create an About page

For the Contoso University website's **About** page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Create the About method in the Home controller.
- Create the About view.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the new folder, add a class file *EnrollmentDateGroup.cs* and replace the template code with the following code:

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Modify the Home Controller

In *HomeController.cs*, add the following using statements at the top of the file:

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;

    public HomeController(SchoolContext context)
    {
        _context = context;
    }
```

Add an `About` method with the following code:

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Create the About View

Add a *Views/Home/About.cshtml* file with the following code:

```
@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and go to the About page. The count of students for each enrollment date is displayed in a table.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Added column sort links
- Added a Search box
- Added paging to Students Index
- Added paging to Index method
- Added paging links
- Created an About page

Advance to the next tutorial to learn how to handle data model changes by using migrations.

Next: Handle data model changes

Tutorial: Using the migrations feature - ASP.NET MVC with EF Core

9/22/2020 • 7 minutes to read • Edit Online

In this tutorial, you start using the EF Core migrations feature for managing data model changes. In later tutorials, you'll add more migrations as you change the data model.

In this tutorial, you:

- Learn about migrations
- Change the connection string
- Create an initial migration
- Examine Up and Down methods
- Learn about the data model snapshot
- Apply the migration

Prerequisites

- Sorting, filtering, and paging

About migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You started these tutorials by configuring the Entity Framework to create the database if it doesn't exist. Then each time you change the data model -- add, remove, or change entity classes or change your DbContext class -- you can delete the database and EF creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it's usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The EF Core Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.

To work with migrations, you can use the **Package Manager Console** (PMC) or the CLI. These tutorials show how to use CLI commands. Information about the PMC is at the end of this tutorial.

Change the connection string

In the *appsettings.json* file, change the name of the database in the connection string to ContosoUniversity2 or some other name that you haven't used on the computer you're using.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

This change sets up the project so that the first migration will create a new database. This isn't required to get started with migrations, but you'll see later why it's a good idea.

NOTE

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

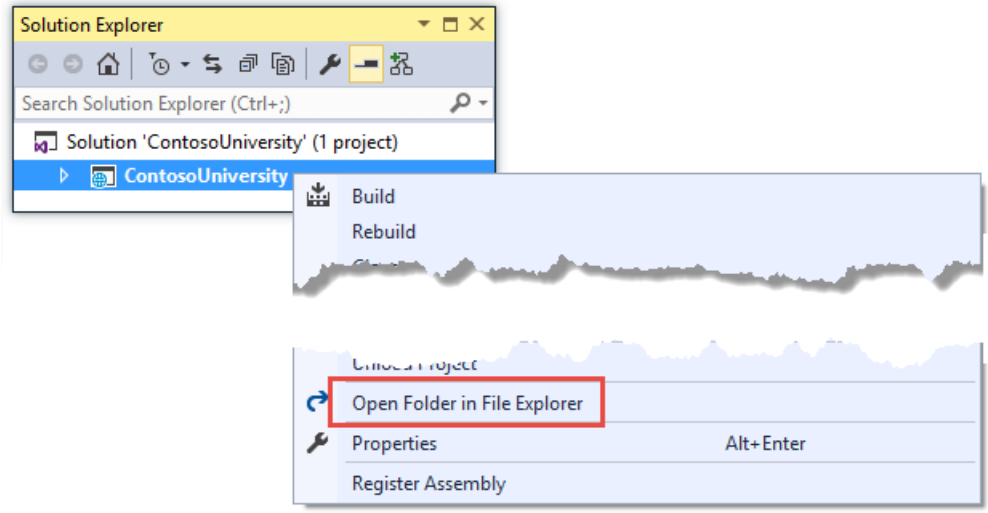
```
dotnet ef database drop
```

The following section explains how to run CLI commands.

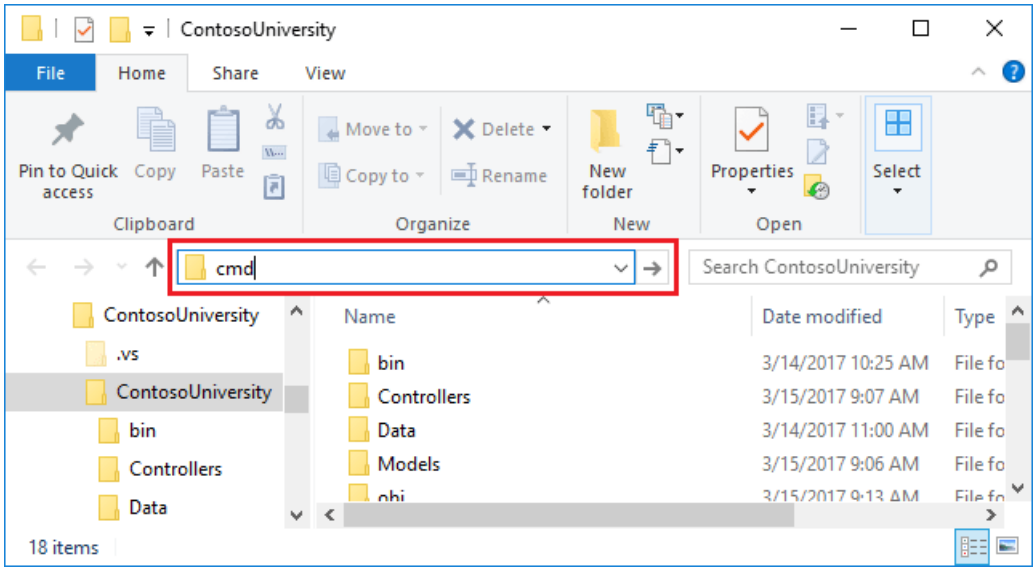
Create an initial migration

Save your changes and build the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

- In **Solution Explorer**, right-click the project and choose **Open Folder in File Explorer** from the context menu.



- Enter "cmd" in the address bar and press Enter.



Enter the following command in the command window:

```
dotnet tool install --global dotnet-ef
dotnet ef migrations add InitialCreate
```

`dotnet tool install --global dotnet-ef` installs `dotnet ef` as a global tool.

In the preceding commands, output similar to the following is displayed:

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.2.0-rtm-35687 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

If you see an error message "*cannot access the file ... ContosoUniversity.dll because it is being used by another process.*", find the IIS Express icon in the Windows System Tray, and right-click it, then click **ContosoUniversity > Stop Site**.

Examine Up and Down methods

When you executed the `migrations add` command, EF generated the code that will create the database from scratch. This code is in the *Migrations* folder, in the file named *<timestamp>_InitialCreate.cs*. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");
        // Additional code not shown
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the `migrations add InitialCreate` command. The migration name parameter ("InitialCreate" in the example) is used for the file name and can be whatever you want. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a

good idea to test it first. That's why you changed the name of the database in the connection string earlier -- so that migrations can create a new one from scratch.

The data model snapshot

Migrations creates a *snapshot* of the current database schema in *Migrations/SchoolContextModelSnapshot.cs*. When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

Use the dotnet ef migrations remove command to remove a migration. `dotnet ef migrations remove` deletes the migration and ensures the snapshot is correctly reset. If `dotnet ef migrations remove` fails, use `dotnet ef migrations remove -v` to get more information on the failure.

See EF Core Migrations in Team Environments for more information about how the snapshot file is used.

Apply the migration

In the command window, enter the following command to create the database and tables in it.

```
dotnet ef database update
```

The output from the command is similar to the `migrations add` command, except that you see logs for the SQL commands that set up the database. Most of the logs are omitted in the following sample output. If you prefer not to see this level of detail in log messages, you can change the log level in the *appsettings.Development.json* file. For more information, see Logging in .NET Core and ASP.NET Core.

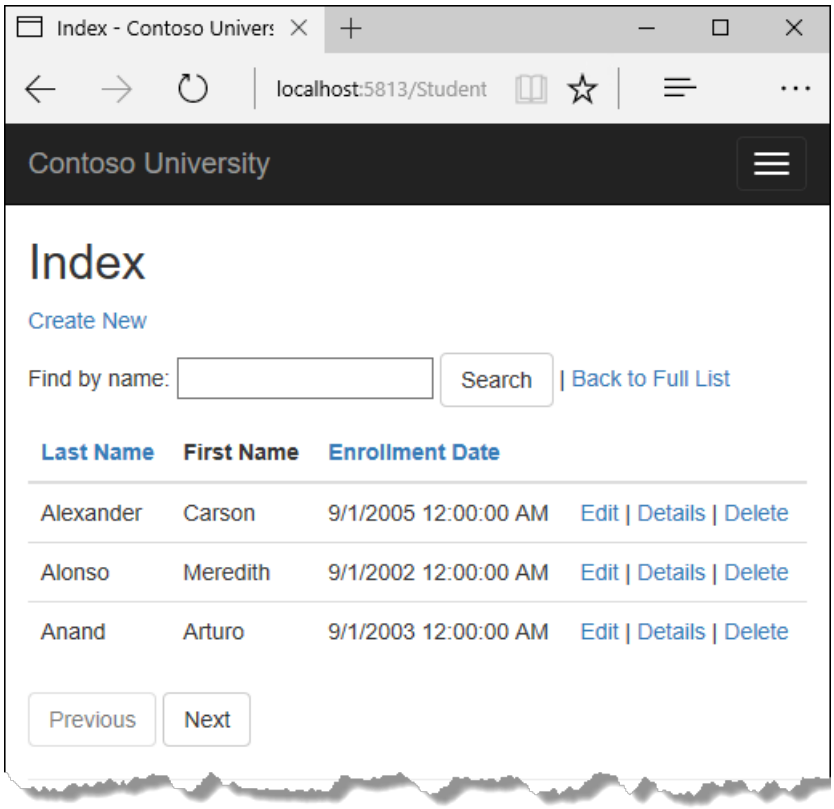
```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 2.2.0-rtm-35687 initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (274ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (60ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [ContosoUniversity2] SET READ_COMMITTED_SNAPSHOT ON;
      END;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );

<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20190327172701_InitialCreate', N'2.2.0-rtm-35687');
Done.
```

Use **SQL Server Object Explorer** to inspect the database as you did in the first tutorial. You'll notice the addition of an __EFMigrationsHistory table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one row for the first migration. (The last log in the preceding CLI output example shows the INSERT statement that creates this row.)

Run the application to verify that everything still works the same as before.



Compare CLI and PMC

The EF tooling for managing migrations is available from .NET Core CLI commands or from PowerShell cmdlets in the Visual Studio **Package Manager Console** (PMC) window. This tutorial shows how to use the CLI, but you can use the PMC if you prefer.

The EF commands for the PMC commands are in the Microsoft.EntityFrameworkCore.Tools package. This package is included in the Microsoft.AspNetCore.App metapackage, so you don't need to add a package reference if your app has a package reference for `Microsoft.AspNetCore.App`.

Important: This isn't the same package as the one you install for the CLI by editing the *.csproj* file. The name of this one ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see .NET Core CLI.

For more information about the PMC commands, see Package Manager Console (Visual Studio).

Get the code

Download or view the completed application.

Next step

In this tutorial, you:

- Learned about migrations
- Learned about NuGet migration packages
- Changed the connection string
- Created an initial migration
- Examined Up and Down methods
- Learned about the data model snapshot
- Applied the migration

Advance to the next tutorial to begin looking at more advanced topics about expanding the data model. Along the way you'll create and apply additional migrations.

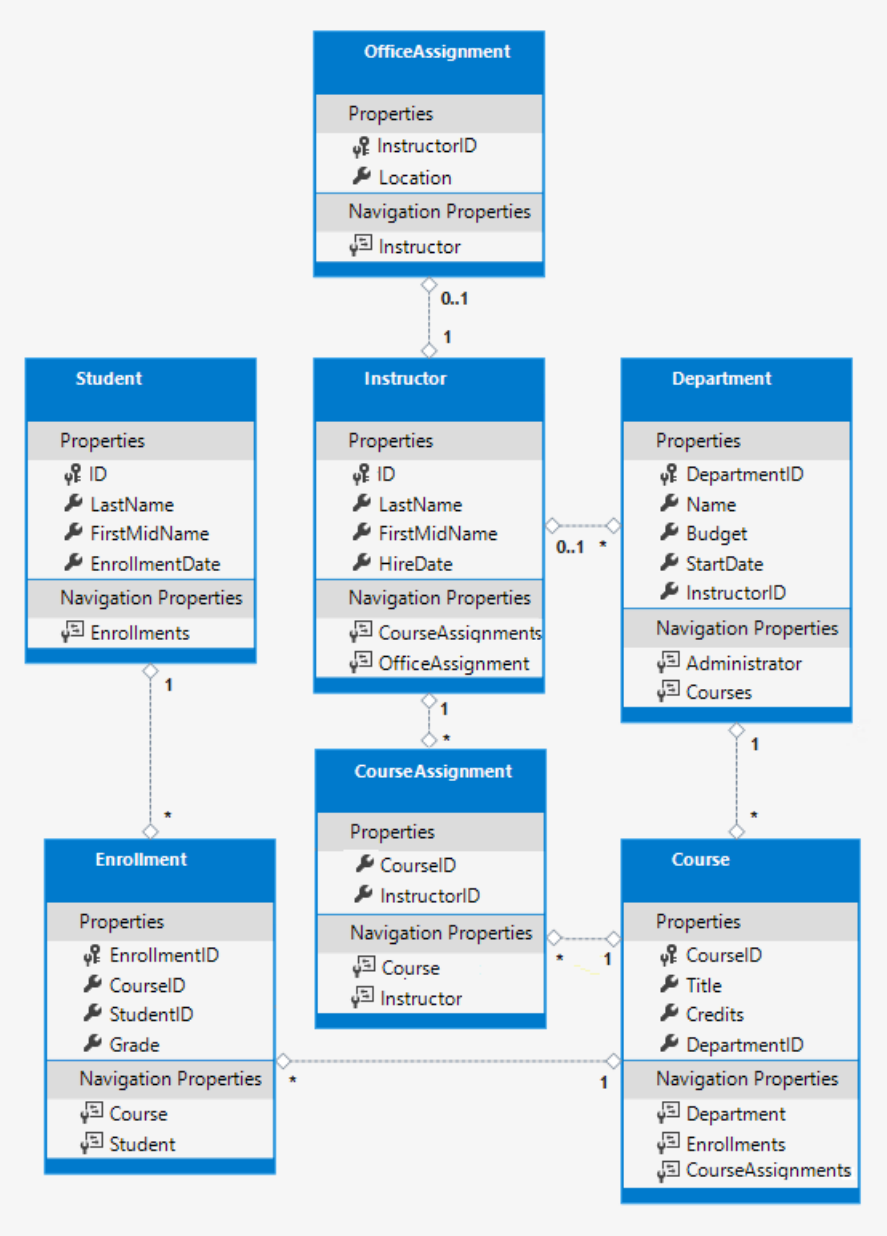
Create and apply additional migrations

Tutorial: Create a complex data model - ASP.NET MVC with EF Core

9/22/2020 • 30 minutes to read • Edit Online

In the previous tutorials, you worked with a simple data model that was composed of three entities. In this tutorial, you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



In this tutorial, you:

- Customize the Data model
- Make changes to Student entity
- Create Instructor entity
- Create OfficeAssignment entity

- Modify Course entity
- Create Department entity
- Modify Enrollment entity
- Update the database context
- Seed database with test data
- Add a migration
- Change the connection string
- Update the database

Prerequisites

- Using EF Core migrations

Customize the Data model

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete School data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In *Models/Student.cs*, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes don't provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

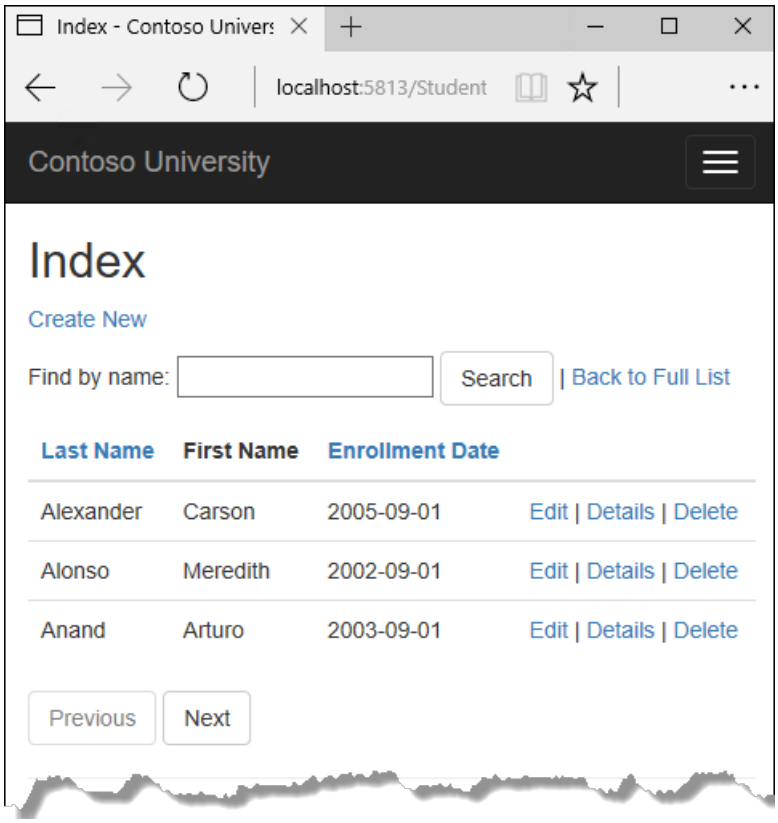
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).
- By default, the browser will render data using the correct format based on your locale.

For more information, see the <input> tag helper documentation.

Run the app, go to the Students Index page and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the Student model.



The StringLength attribute

You can also specify data validation rules and validation error messages using attributes. The `StringLength` attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET Core MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `StringLength` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `StringLength` attribute won't prevent a user from entering white space for a name. You can use the `RegularExpression` attribute to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

The `MaxLength` attribute provides functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

```
dotnet ef migrations add MaxLengthOnNames
```

```
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations creates a file named *<timeStamp>_MaxLengthOnNames.cs*. This file contains code in the `Up` method that will update the database to match the current data model. The `database update` command ran that code.

The timestamp prefixed to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the update-database command, and then all of the migrations are applied in the order in which they were created.

Run the app, select the **Students** tab, click **Create New**, and try to enter either name longer than 50 characters. The application should prevent you from doing this.

The Column attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name.

But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they're given the same name as the property name.

In the *Student.cs* file, add a `using` statement for `System.ComponentModel.DataAnnotations.Schema` and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

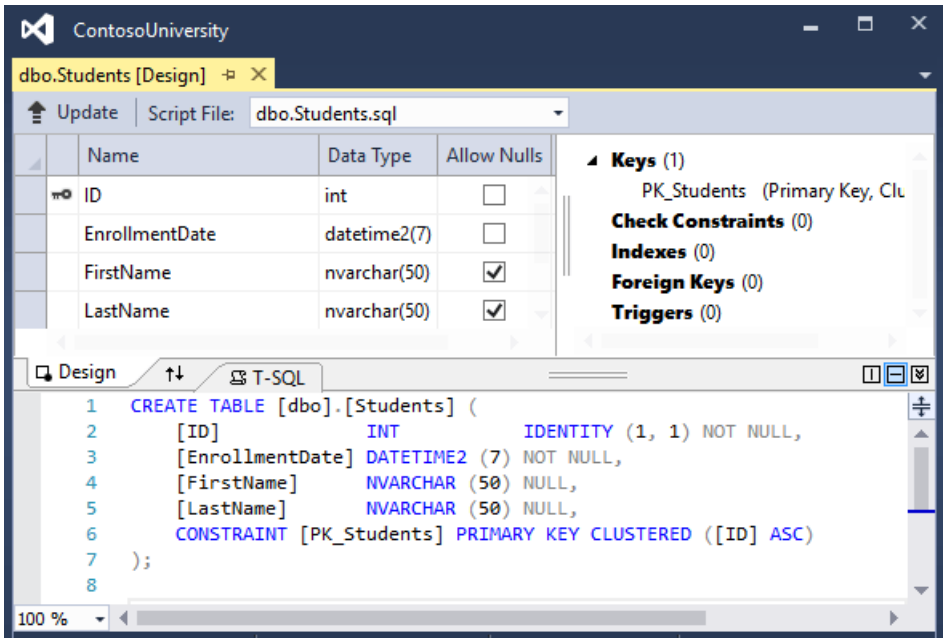
The addition of the `Column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

```
dotnet ef migrations add ColumnFirstName
```

```
dotnet ef database update
```

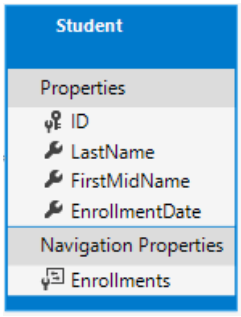
In **SQL Server Object Explorer**, open the Student table designer by double-clicking the **Student** table.



Before you applied the first two migrations, the name columns were of type nvarchar(MAX). They're now nvarchar(50) and the column name has changed from FirstMidName to FirstName.

> **NOTE**
> If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Changes to Student entity



In *Models/Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50)]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

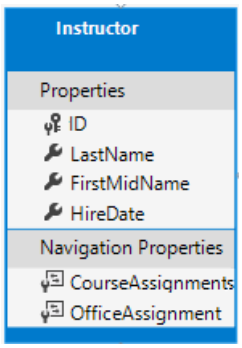
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a get accessor, and no `FullName` column will be generated in the database.

Create Instructor entity



Create *Models/Instructor.cs*, replacing the template code with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the Student and Instructor entities. In the Implementing Inheritance tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the `HireDate` attributes as follows:

```
[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

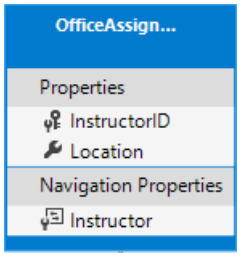
If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single OfficeAssignment entity (which may be null if no office is assigned).

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create OfficeAssignment entity



Create *Models/OfficeAssignment.cs* with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

There's a one-to-zero-or-one relationship between the Instructor and the OfficeAssignment entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the Instructor entity. But the Entity Framework can't automatically recognize InstructorID as the primary key of this entity because its name doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify it as the key:

```
[Key]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something other than classnameID or ID.

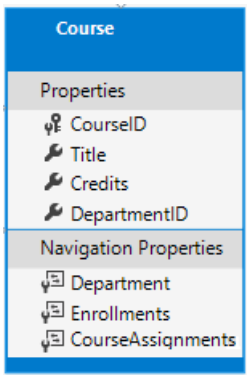
By default, EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The Instructor entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the OfficeAssignment entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an Instructor entity has a related OfficeAssignment entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the `InstructorID` foreign key (which is also the key to this table) is non-nullable.

Modify Course entity



In *Models/Course.cs*, replace the code you added earlier with the following code. The changes are highlighted.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related Department entity and it has a `Department` navigation property.

The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they're needed and creates shadow properties for them. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the Department entity is null if you don't load it, so when you update the course entity, you would have to first fetch the Department entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the Department entity before you update.

The DatabaseGenerated attribute

The `DatabaseGenerated` attribute with the `None` parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for Course entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key properties and navigation properties in the Course entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

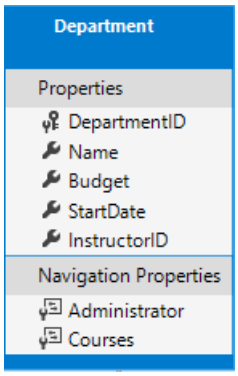
A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained later):

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create Department entity



Create *Models/Department.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Earlier you used the `Column` attribute to change column name mapping. In the code for the Department entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the Instructor entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation property is named `Administrator` but holds an Instructor entity:

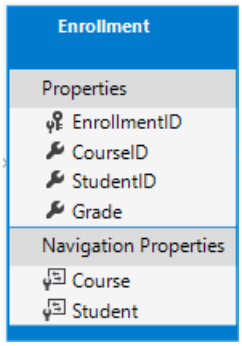
```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

A department may have many courses, so there's a Courses navigation property:

```
public ICollection<Course> Courses { get; set; }
```

> **NOTE**
> By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the Department.InstructorID property as nullable, EF would configure a cascade delete rule to delete the department when you delete the instructor, which isn't what you want to have happen. If your business rules required the `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:
>
> ```
> modelBuilder.Entity<Department>()
>    .HasOne(d => d.Administrator)
>    .WithMany()
>    .OnDelete(DeleteBehavior.Restrict)
> ```

Modify Enrollment entity



In *Models/Enrollment.cs*, replace the code you added earlier with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The foreign key properties and navigation properties reflect the following relationships:

An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

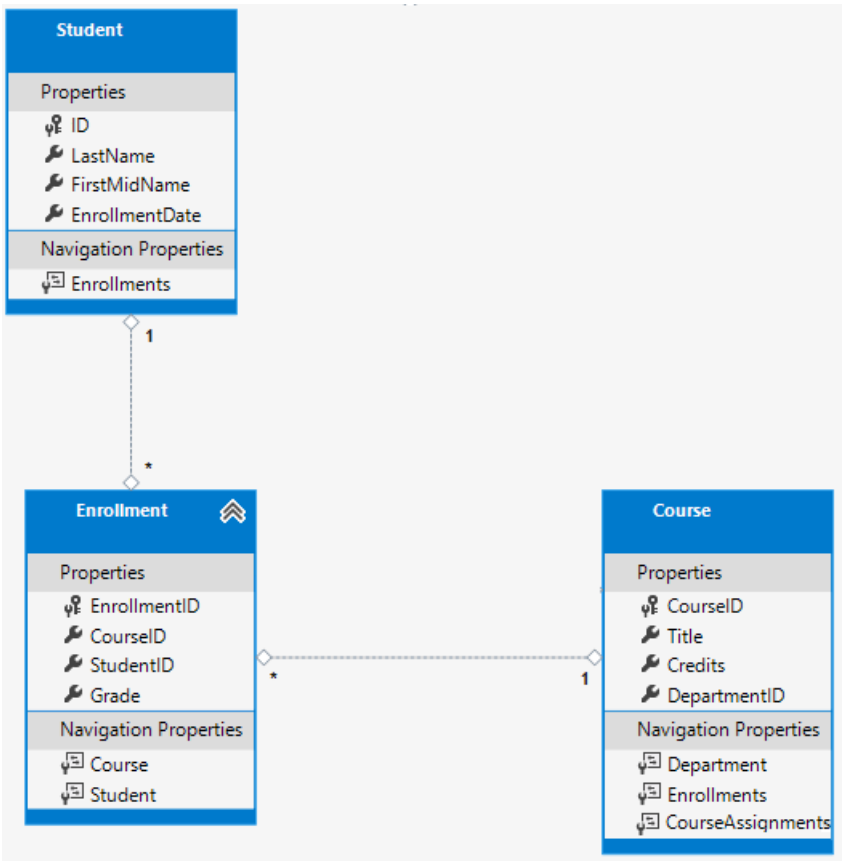
An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many relationships

There's a many-to-many relationship between the Student and Course entities, and the Enrollment entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the Enrollment table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a Grade property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

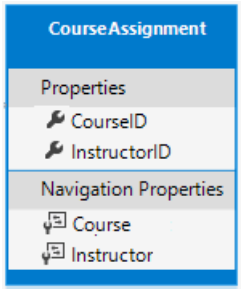


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the Enrollment table didn't include grade information, it would only need to contain the two foreign keys CourseID and StudentID. In that case, it would be a many-to-many join table without payload (or a pure join table) in the database. The Instructor and Course entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

(EF 6.x supports implicit join tables for many-to-many relationships, but EF Core doesn't. For more information, see the discussion in the EF Core GitHub repository.)

The CourseAssignment entity



Create *Models/CourseAssignment.cs* with the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there's no need for a separate primary key. The *InstructorID* and *CourseID* properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Add the following highlighted code to the *Data/SchoolContext.cs* file:

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

About a fluent API alternative

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the EF Core documentation:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

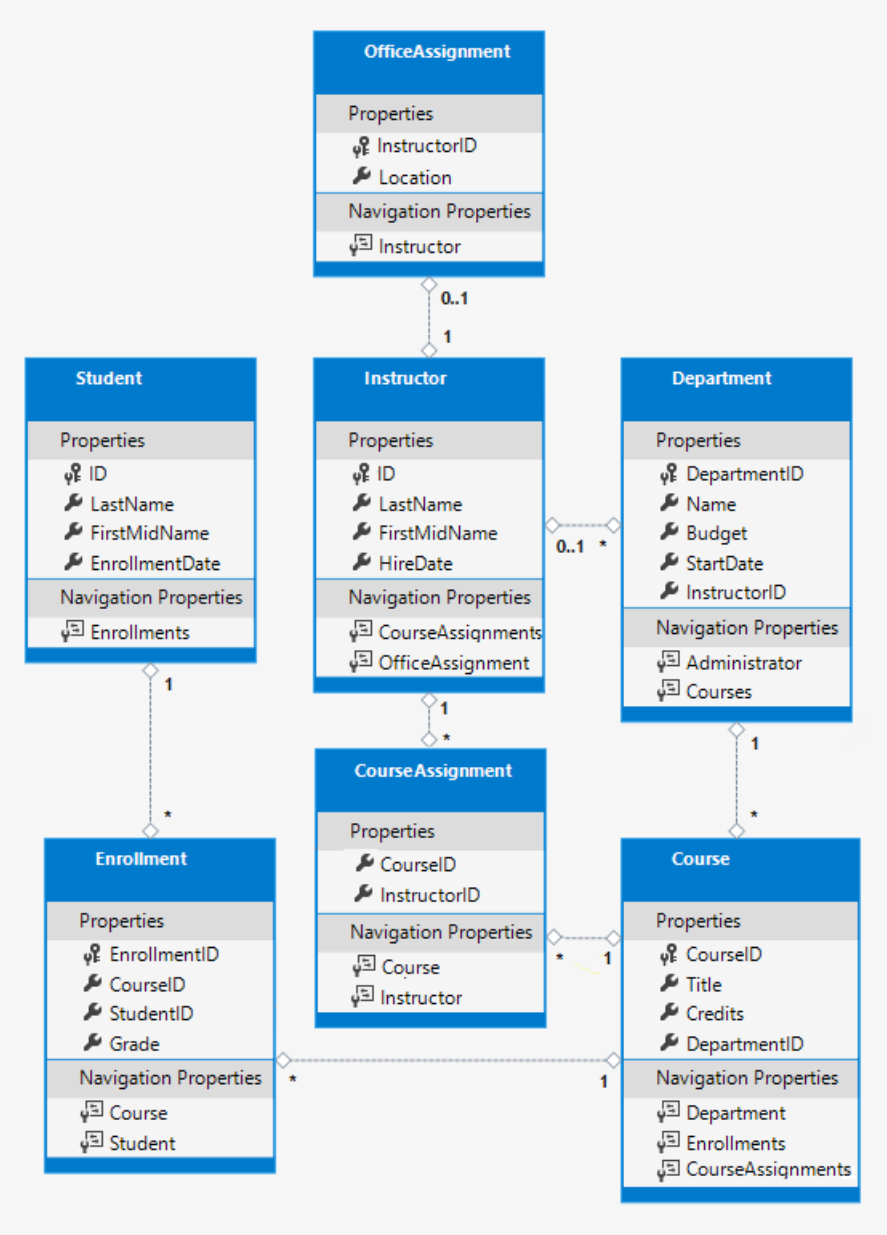
In this tutorial, you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible. If you do use both, note that wherever there's a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the Instructor and OfficeAssignment entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Seed database with test data

Replace the code in the *Data/DbInitializer.cs* file with the following code in order to provide seed data for the new entities you've created.

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
```

```
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName = "Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName = "Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName = "Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName = "Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName = "Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            foreach (Instructor i in instructors)
            {
                context.Instructors.Add(i);
            }
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName == "Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
```

```
            InstructorID  = instructors.Single( i => i.LastName == "Harui").ID },
        new Department { Name = "Economics",   Budget = 100000,
            StartDate = DateTime.Parse("2007-09-01"),
            InstructorID  = instructors.Single( i => i.LastName == "Kapoor").ID }
    };

    foreach (Department d in departments)
    {
        context.Departments.Add(d);
    }
    context.SaveChanges();

    var courses = new Course[]
    {
        new Course {CourseID = 1050, Title = "Chemistry",      Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Engineering").DepartmentID
        },
        new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "Economics").DepartmentID
        },
        new Course {CourseID = 1045, Title = "Calculus",       Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 3141, Title = "Trigonometry",   Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "Mathematics").DepartmentID
        },
        new Course {CourseID = 2021, Title = "Composition",    Credits = 3,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
        new Course {CourseID = 2042, Title = "Literature",     Credits = 4,
            DepartmentID = departments.Single( s => s.Name == "English").DepartmentID
        },
    };

    foreach (Course c in courses)
    {
        context.Courses.Add(c);
    }
    context.SaveChanges();

    var officeAssignments = new OfficeAssignment[]
    {
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Fakhouri").ID,
            Location = "Smith 17" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Harui").ID,
            Location = "Gowan 27" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName == "Kapoor").ID,
            Location = "Thompson 304" },
    };

    foreach (OfficeAssignment o in officeAssignments)
    {
        context.OfficeAssignments.Add(o);
    }
    context.SaveChanges();

    var courseInstructors = new CourseAssignment[]
    {
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
            },
        new CourseAssignment {
```

```
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Harui").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Fakhouri").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Harui").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
                },
            new CourseAssignment {
                CourseID = courses.Single(c => c.Title == "Literature" ).CourseID,
                InstructorID = instructors.Single(i => i.LastName == "Abercrombie").ID
                },
        };

        foreach (CourseAssignment ci in courseInstructors)
        {
            context.CourseAssignments.Add(ci);
        }
        context.SaveChanges();

        var enrollments = new Enrollment[]
        {
            new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alexander").ID,
                CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
                Grade = Grade.A
            },
                new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alexander").ID,
                CourseID = courses.Single(c => c.Title == "Microeconomics" ).CourseID,
                Grade = Grade.C
                },
                new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alexander").ID,
                CourseID = courses.Single(c => c.Title == "Macroeconomics" ).CourseID,
                Grade = Grade.B
                },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alonso").ID,
                CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
                Grade = Grade.B
                },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Alonso").ID,
                CourseID = courses.Single(c => c.Title == "Trigonometry" ).CourseID,
                Grade = Grade.B
                },
                new Enrollment {
                StudentID = students.Single(s => s.LastName == "Alonso").ID,
                CourseID = courses.Single(c => c.Title == "Composition" ).CourseID,
                Grade = Grade.B
                },
                new Enrollment {
```

```
                    StudentID = students.Single(s => s.LastName == "Anand").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Anand").ID,
                    CourseID = courses.Single(c => c.Title == "Microeconomics").CourseID,
                    Grade = Grade.B
                    },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title == "Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Justice").ID,
                    CourseID = courses.Single(c => c.Title == "Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID == e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

```
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If you tried to run the `database update` command at this point (don't do it yet), you would get the following error:

> The ALTER TABLE statement conflicted with the FOREIGN KEY constraint

> "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints. The generated code in the `Up` method adds a non-nullable DepartmentID foreign key to the Course table. If there are already rows in the Course table when the code runs, the `AddColumn` operation fails because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing Course rows will all be related to the "Temp" department after the `Up` method runs.

- Open the *{timestamp}_ComplexDataModel.cs* file.
- Comment out the line of code that adds the DepartmentID column to the Course table.

```
migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

- Add the following highlighted code after the code that creates the Department table:

```
migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00,
GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the Course.DepartmentID column.

Save your changes and build the project.

Change the connection string

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in *appsettings.json* to ContosoUniversity3 or some other name that you haven't used on the computer you're using.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
```

Save your change to *appsettings.json*.

NOTE

As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:

```
dotnet ef database drop
```

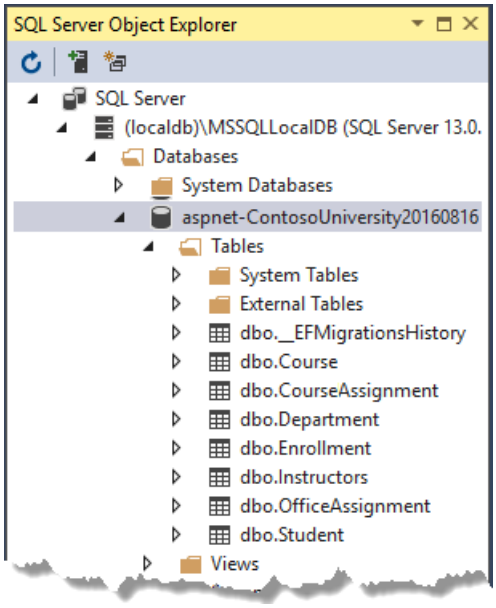
Update the database

After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

```
dotnet ef database update
```

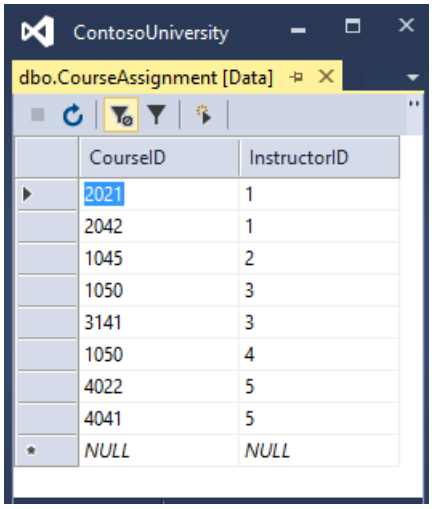
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

Open the database in SSOX as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have SSOX open from the earlier time, click the **Refresh** button.)



Run the app to trigger the initializer code that seeds the database.

Right-click the **CourseAssignment** table and select **View Data** to verify that it has data in it.



Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Customized the Data model
- Made changes to Student entity
- Created Instructor entity
- Created OfficeAssignment entity
- Modified Course entity
- Created Department entity
- Modified Enrollment entity
- Updated the database context
- Seeded database with test data
- Added a migration
- Changed the connection string
- Updated the database

Advance to the next tutorial to learn more about how to access related data.

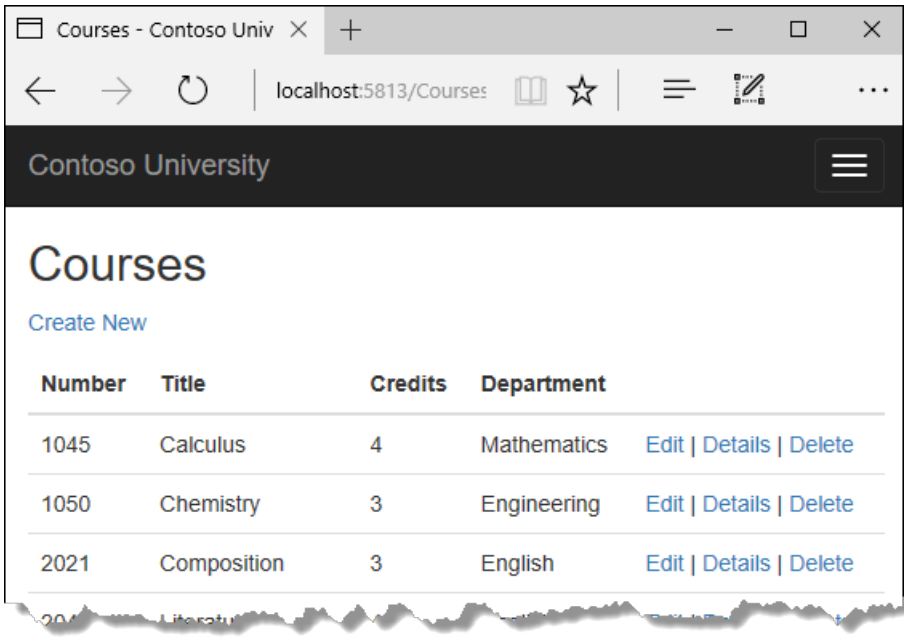
Next: Access related data

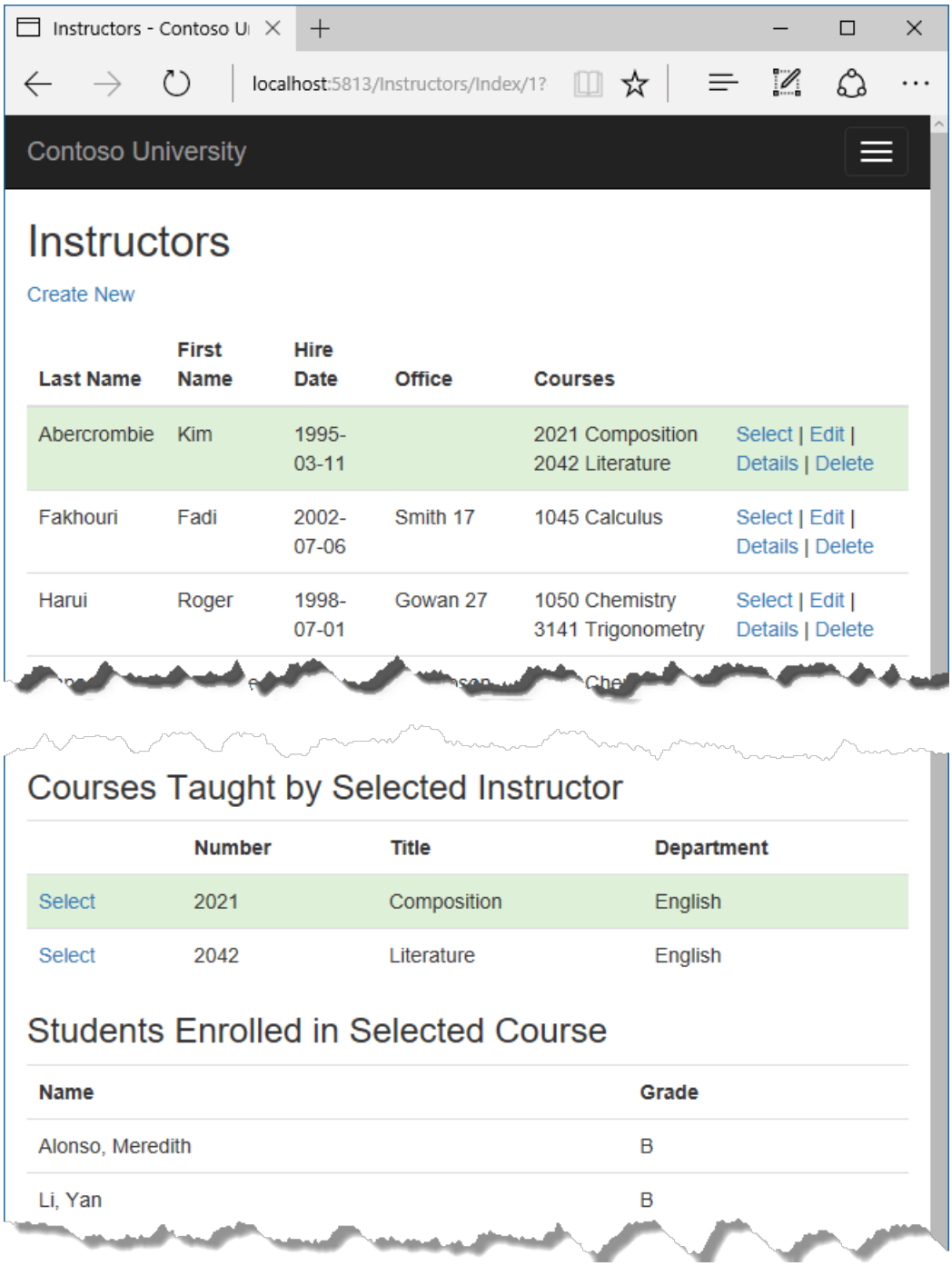
Tutorial: Read related data - ASP.NET MVC with EF Core

9/22/2020 • 14 minutes to read • Edit Online

In the previous tutorial, you completed the School data model. In this tutorial, you'll read and display related data -- that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.





In this tutorial, you:

- Learn how to load related data
- Create a Courses page
- Create an Instructors page
- Learn about explicit loading

Prerequisites

- Create a complex data model

Learn how to load related data

There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

- Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a

single join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of previously retrieved entities. For the query that retrieves related data, you can use the `Load` method instead of a method that returns a list or object, such as `ToList` or `Single`.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Explicit loading. When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed. As in the case of eager loading with separate queries, explicit loading results in multiple queries sent to the database. The difference is that with explicit loading, the code specifies the navigation properties to be loaded. In Entity Framework Core 1.1 you can use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Lazy loading. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. A query is sent to the database each time you try to get data from a navigation property for the first time. Entity Framework Core 1.0 doesn't support lazy loading.

Performance considerations

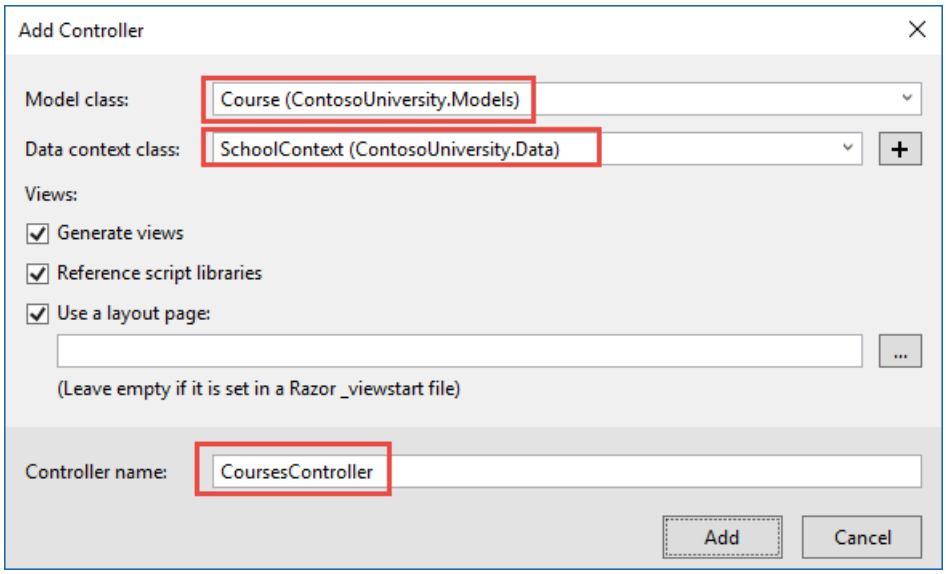
If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager loading of all related data would result in just a single (join) query and a single round trip to the database. A separate query for courses for each department would result in eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, separate queries might perform better because eager loading of everything up front would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Create a Courses page

The Course entity includes a navigation property that contains the Department entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the Name property from the Department entity that's in the `Course.Department` navigation property.

Create a controller named CoursesController for the Course entity type, using the same options for the **MVC Controller with views, using Entity Framework** scaffolder that you did earlier for the Students controller, as shown in the following illustration:



Open *CoursesController.cs* and examine the `Index` method. The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Replace the `Index` method with the following code that uses a more appropriate name for the `IQueryable` that returns Course entities (`courses` instead of `schoolContext`):

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Open *Views/Courses/Index.cshtml* and replace the template code with the following code. The changes are highlighted:

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewData["Title"] = "Courses";
}

<h2>Courses</h2>

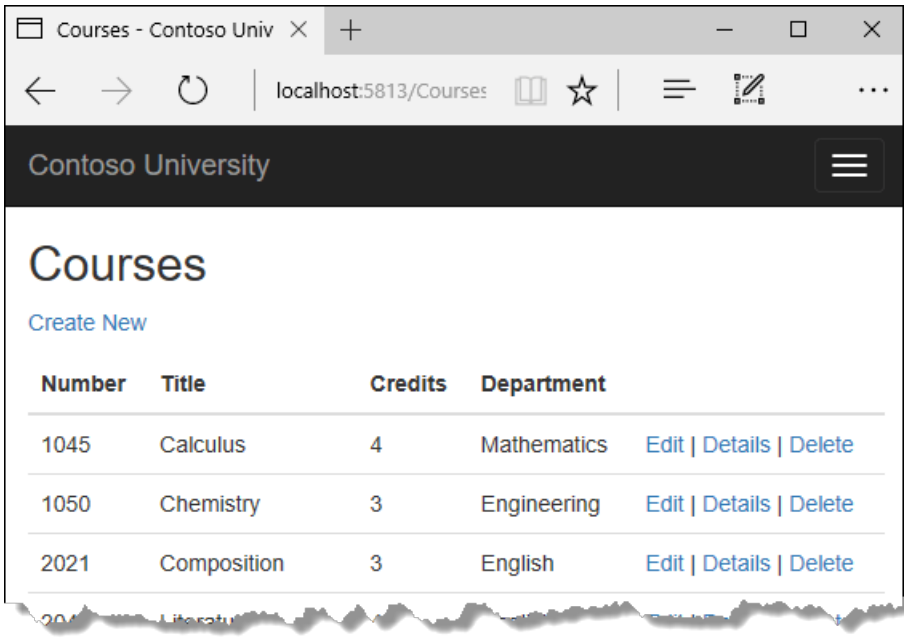
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

You've made the following changes to the scaffolded code:

- Changed the heading from Index to Courses.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the Department entity that's loaded into the `Department` navigation property:

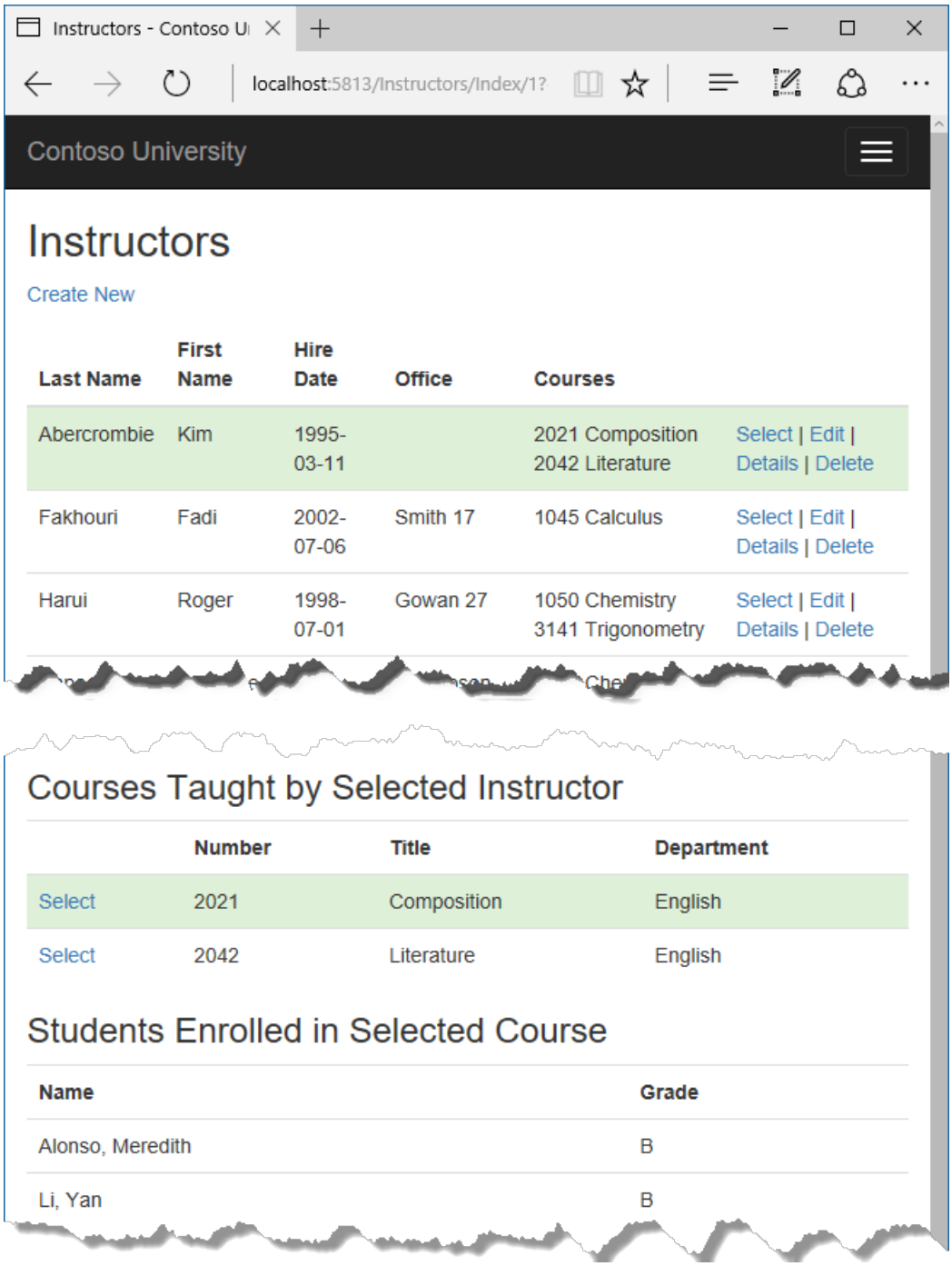
```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Create an Instructors page

In this section, you'll create a controller and view for the Instructor entity in order to display the Instructors page:



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the OfficeAssignment entity. The Instructor and OfficeAssignment entities are in a one-to-zero-or-one relationship. You'll use eager loading for the OfficeAssignment entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.
- When the user selects an instructor, related Course entities are displayed. The Instructor and Course entities are in a many-to-many relationship. You'll use eager loading for the Course entities and their related Department entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the Enrollments entity set is displayed. The Course and Enrollment entities are in a one-to-many relationship. You'll use separate queries for Enrollment entities and their related Student entities.

Create a view model for the Instructor Index view

The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

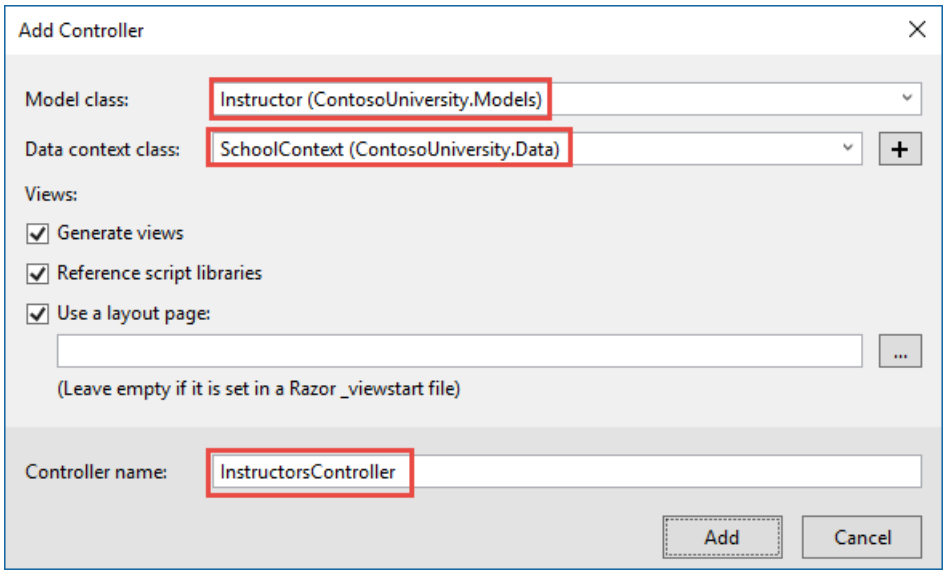
In the *SchoolViewModels* folder, create *InstructorIndexData.cs* and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:



Open *InstructorsController.cs* and add a using statement for the ViewModels namespace:

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Since the view always requires the OfficeAssignment entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string

of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `Enrollment.Student`.

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `Course.Enrollments`.

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the Course entities from that instructor's `CourseAssignments` navigation property.

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single Instructor entity being returned. The `Single` method converts the collection into a single Instructor entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The Single method throws an exception if the collection passed to it's empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the Where condition instead of calling the `Where` method separately:

```
.Single(i => i.ID == id.Value)
```

Instead of:

```
.Where(i => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Modify the Instructor Index view

In *Views/Instructors/Index.cshtml*, replace the template code with the following code. The changes are highlighted.

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.CourseAssignments)
                        {
                            @course.Course.CourseID @:  @course.Course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
          }
    </tbody>
</table>
```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

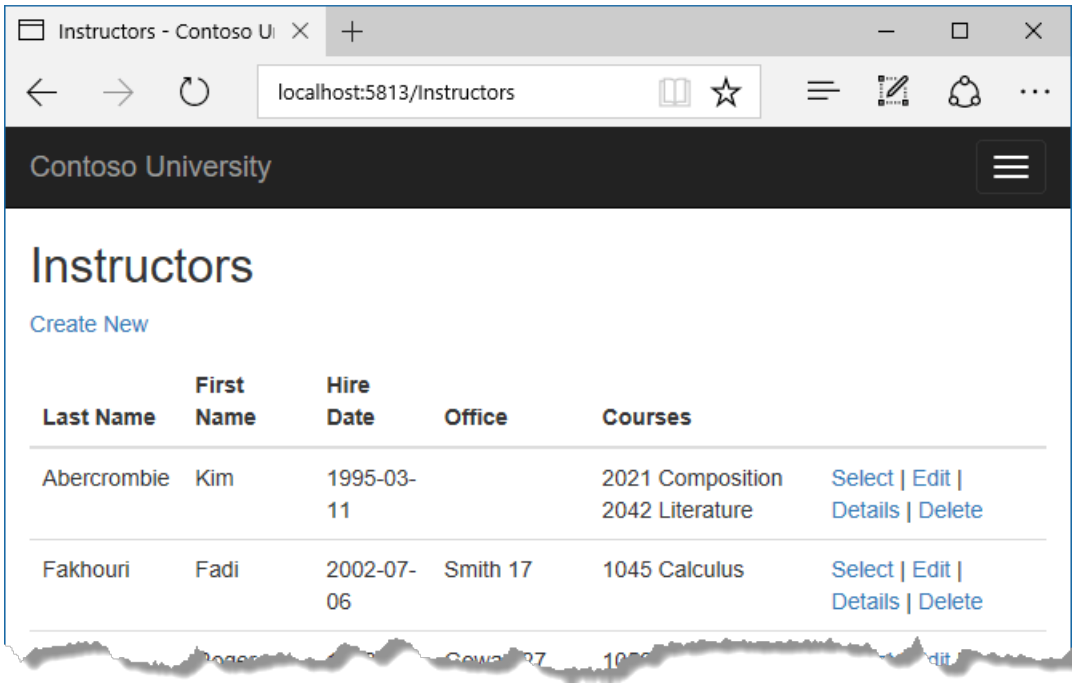
- Added a **Courses** column that displays courses taught by each instructor. For more information, see the Explicit line transition section of the Razor syntax article.
- Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
<tr class="@selectedRow">
```

- Added a new hyperlink labeled **Select** immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.



In the *Views/Instructors/Index.cshtml* file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

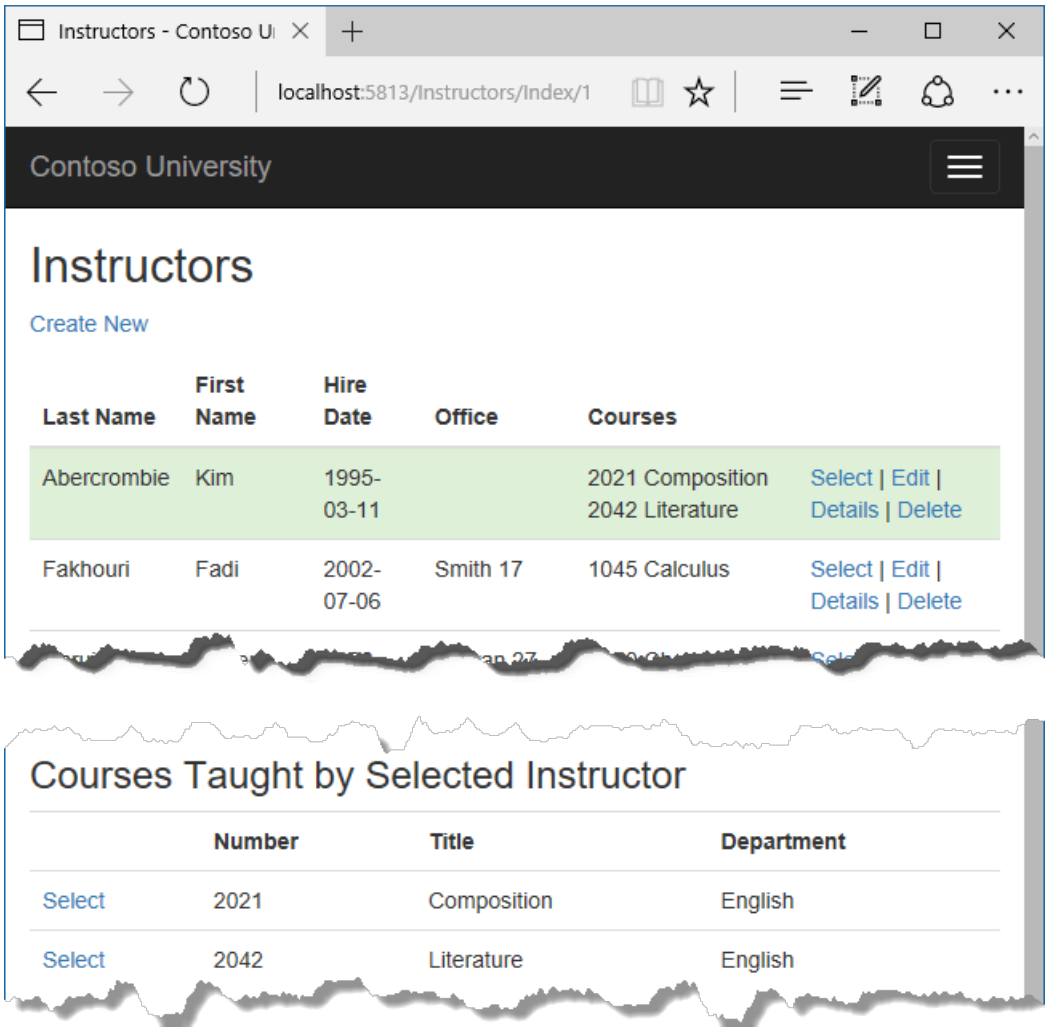
```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID = item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a **Select** hyperlink that sends the ID of the selected course to the `Index` action method.

Refresh the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



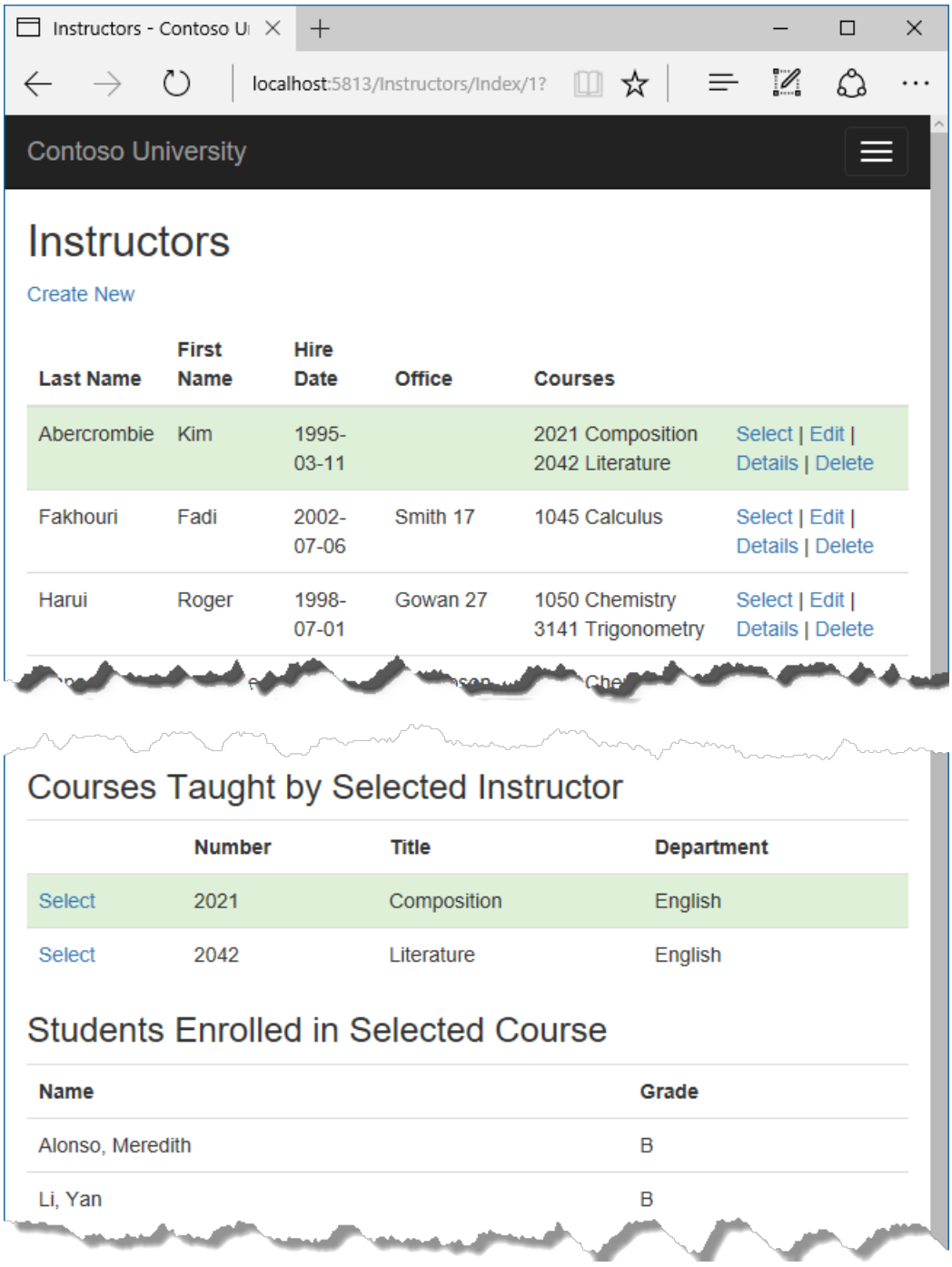
After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the Enrollments property of the view model in order to display a list of students enrolled in the course.

Refresh the page again and select an instructor. Then select a course to see the list of enrolled students and their

grades.



About explicit loading

When you retrieved the list of instructors in *InstructorsController.cs*, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for Enrollments and loads that property explicitly. The code changes are highlighted.

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID == courseID).Single();
        await _context.Entry(selectedCourse).Collection(x => x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

The new code drops the *ThenInclude* method calls for enrollment data from the code that retrieves instructor entities. It also drops `AsNoTracking`. If an instructor and course are selected, the highlighted code retrieves Enrollment entities for the selected course, and Student entities for each Enrollment.

Run the app, go to the Instructors Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Learned how to load related data
- Created a Courses page
- Created an Instructors page
- Learned about explicit loading

Advance to the next tutorial to learn how to update related data.

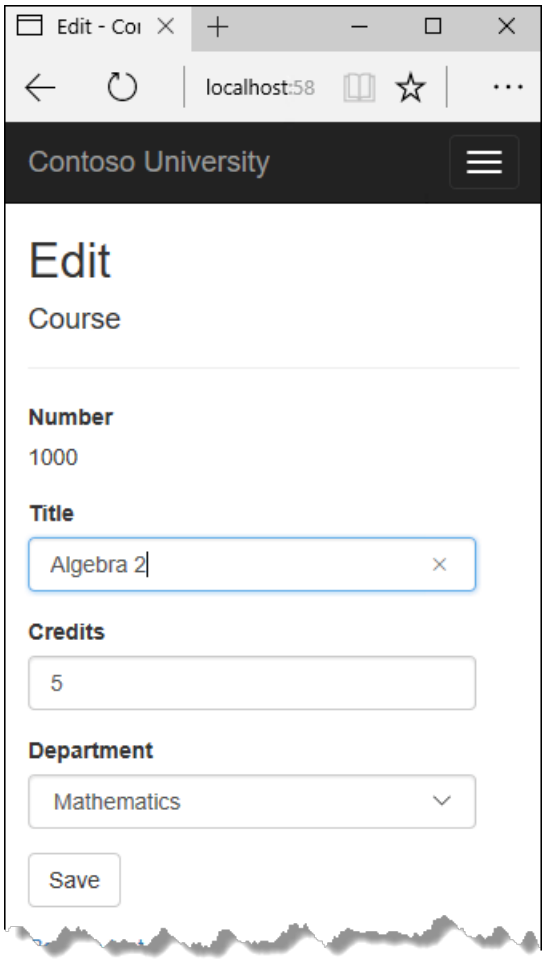
Update related data

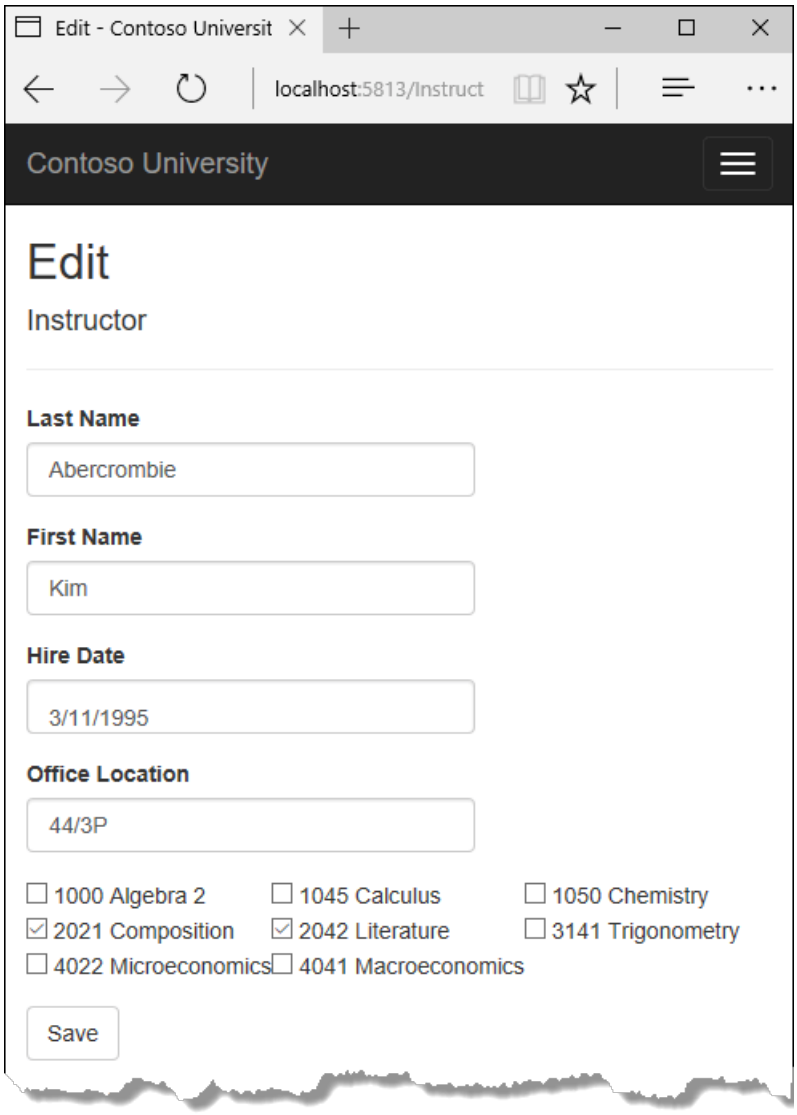
Tutorial: Update related data - ASP.NET MVC with EF Core

9/22/2020 • 18 minutes to read • Edit Online

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.





In this tutorial, you:

- Customize Courses pages
- Add Instructors Edit page
- Add courses to Edit page
- Update Delete page
- Add office location and courses to Create page

Prerequisites

- Read related data

Customize Courses pages

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate Department entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In *CoursesController.cs*, delete the four Create and Edit methods and replace them with the following code:

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .FirstOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}
```

After the `Edit` HttpPost method, create a new method that loads department info for the drop-down list.

```
private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(), "DepartmentID", "Name",
selectedDepartment);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The HttpGet `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department isn't established yet:

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The HttpGet `Edit` method sets the selected item, based on the ID of the department that's already assigned to

the course being edited:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The HttpPost methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add .AsNoTracking to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and HttpGet `Delete` methods.

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

Modify the Course views

In *Views/Courses/Create.cshtml*, add a "Select Department" option to the **Department** drop-down list, change the caption from **DepartmentID** to **Department**, and add a validation message.

```
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />
```

In *Views/Courses/Edit.cshtml*, make the same change for the Department field that you just did in *Create.cshtml*.

Also in *Views/Courses/Edit.cshtml*, add a course number field before the **Title** field. Because the course number is the primary key, it's displayed, but it can't be changed.

```
<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>
```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the **Edit** page.

In *Views/Courses/Delete.cshtml*, add a course number field at the top and change department ID to department name.

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

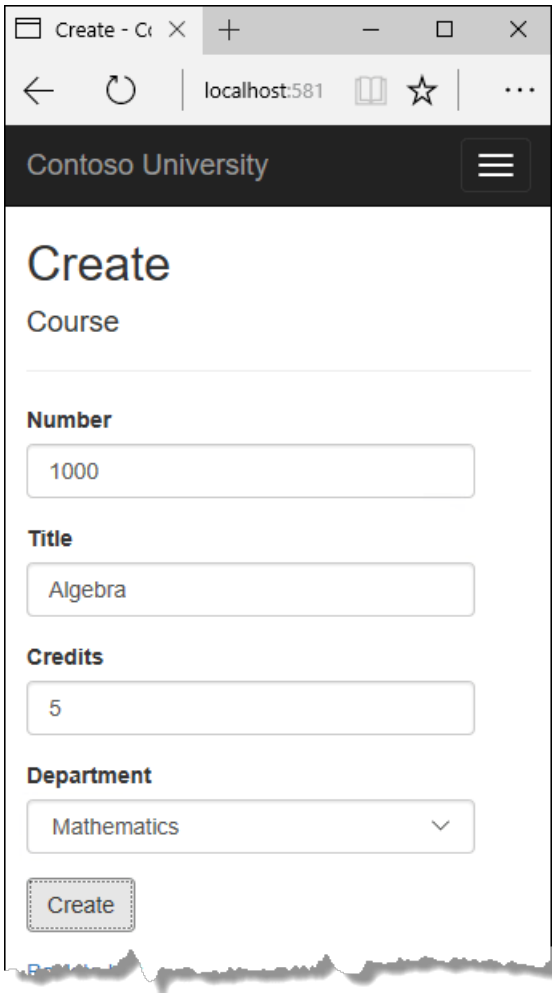
<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

In *Views/Courses/Details.cshtml*, make the same change that you just did for *Delete.cshtml*.

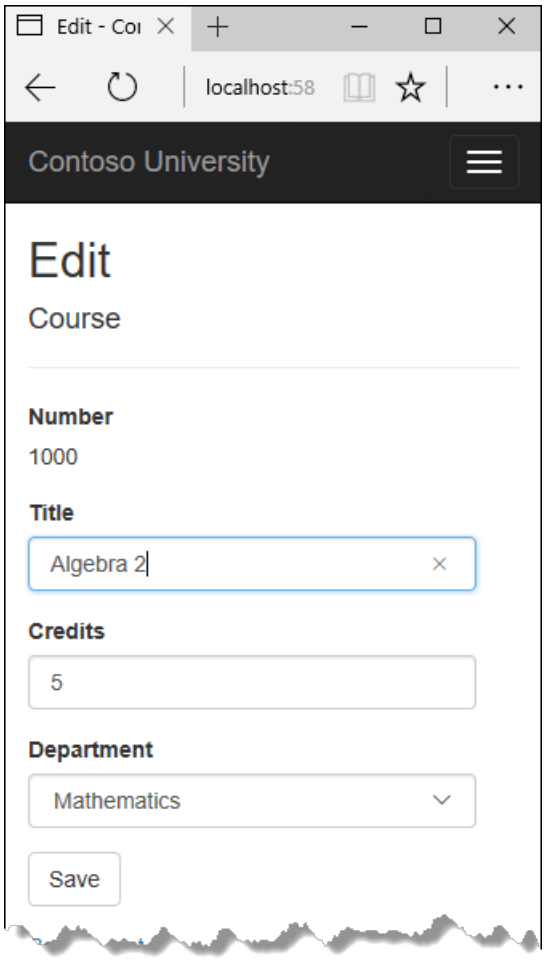
Test the Course pages

Run the app, select the **Courses** tab, click **Create New**, and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Click **Edit** on a course in the Courses Index page.



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add Instructors Edit page

When you edit an instructor record, you want to be able to update the instructor's office assignment. The Instructor entity has a one-to-zero-or-one relationship with the OfficeAssignment entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the OfficeAssignment entity.
- If the user enters an office assignment value and it originally was empty, create a new OfficeAssignment entity.
- If the user changes the value of an office assignment, change the value in an existing OfficeAssignment entity.

Update the Instructors controller

In *InstructorsController.cs*, change the code in the HttpGet `Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Replace the HttpPost `Edit` method with the following code to handle office assignment updates:

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .FirstOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}
```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the HttpGet `Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current Instructor entity from the database using eager loading for the `OfficeAssignment`

navigation property. This is the same as what you did in the HttpGet `Edit` method.

- Updates the retrieved Instructor entity with values from the model binder. The `TryUpdateModel` overload enables you to declare the properties you want to include. This prevents over-posting, as explained in the second tutorial.

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
```

- If the office location is blank, sets the Instructor.OfficeAssignment property to null so that the related row in the OfficeAssignment table will be deleted.

```
if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

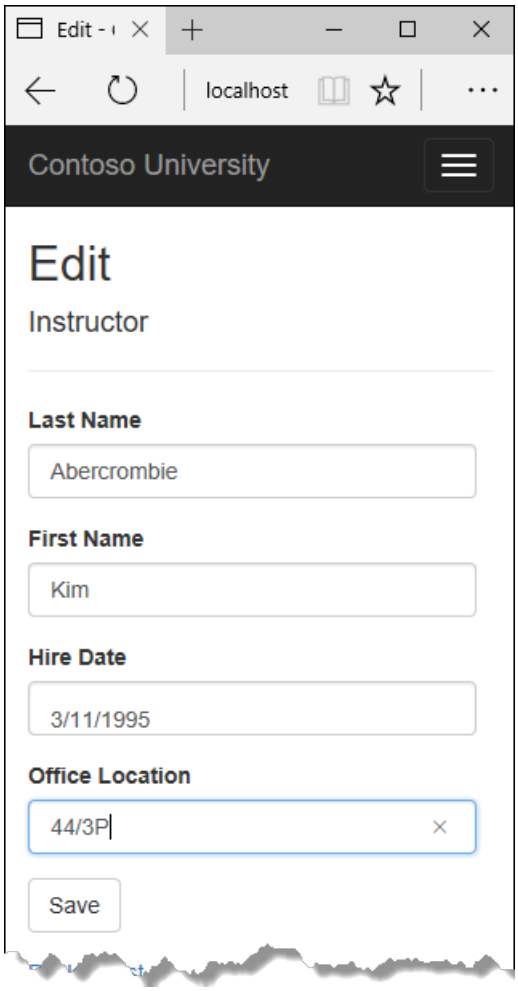
- Saves the changes to the database.

Update the Instructor Edit view

In *Views/Instructors/Edit.cshtml*, add a new field for editing the office location, at the end before the **Save** button:

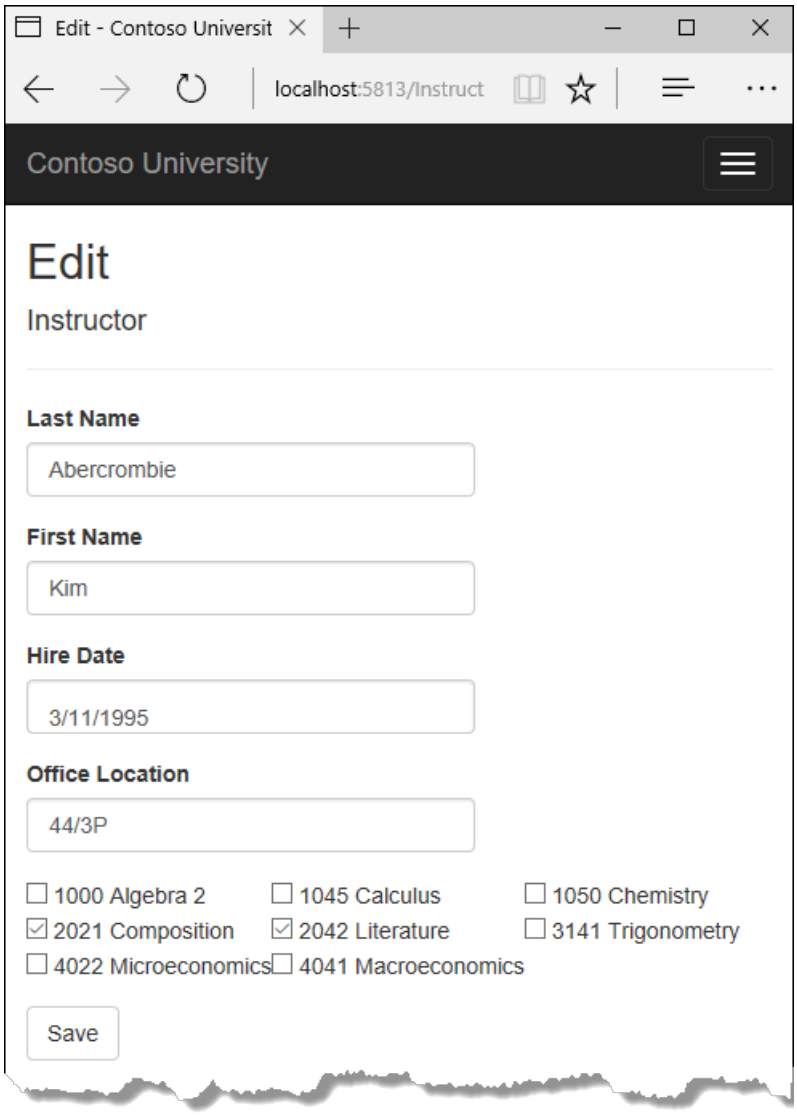
```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>
```

Run the app, select the **Instructors** tab, and then click **Edit** on an instructor. Change the **Office Location** and click **Save**.



Add courses to Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:



The relationship between the Course and Instructor entities is many-to-many. To add and remove relationships, you add and remove entities to and from the CourseAssignments join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of check boxes, you'll use a view model class.

Create *AssignedCourseData.cs* in the *SchoolViewModels* folder and replace the existing code with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In *InstructorsController.cs*, replace the HttpGet `Edit` method with the following code. The changes are highlighted.

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all Course entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set

to true for courses the instructor is assigned to. The view will use this property to determine which check boxes must be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
        .FirstOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i => i.OfficeAssignment))
    {
        if (String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    UpdateInstructorCourses(selectedCourses, instructorToUpdate);
    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}
```

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The method signature is now different from the HttpGet `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore, you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload that requires explicit approval and `CourseAssignments` isn't in the include list.

If no check boxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment { InstructorID =
instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove = instructorToUpdate.CourseAssignments.FirstOrDefault(i =>
i.CourseID == course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

Update the Instructor views

In *Views/Instructors/Edit.cshtml*, add a **Courses** field with an array of check boxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

> **NOTE**
>
> When you paste the code in Visual Studio, line breaks might be changed in a way that breaks the code. If the code looks different after pasting, press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab three times to line up the new code with the existing code. This problem is fixed in Visual Studio 2019.

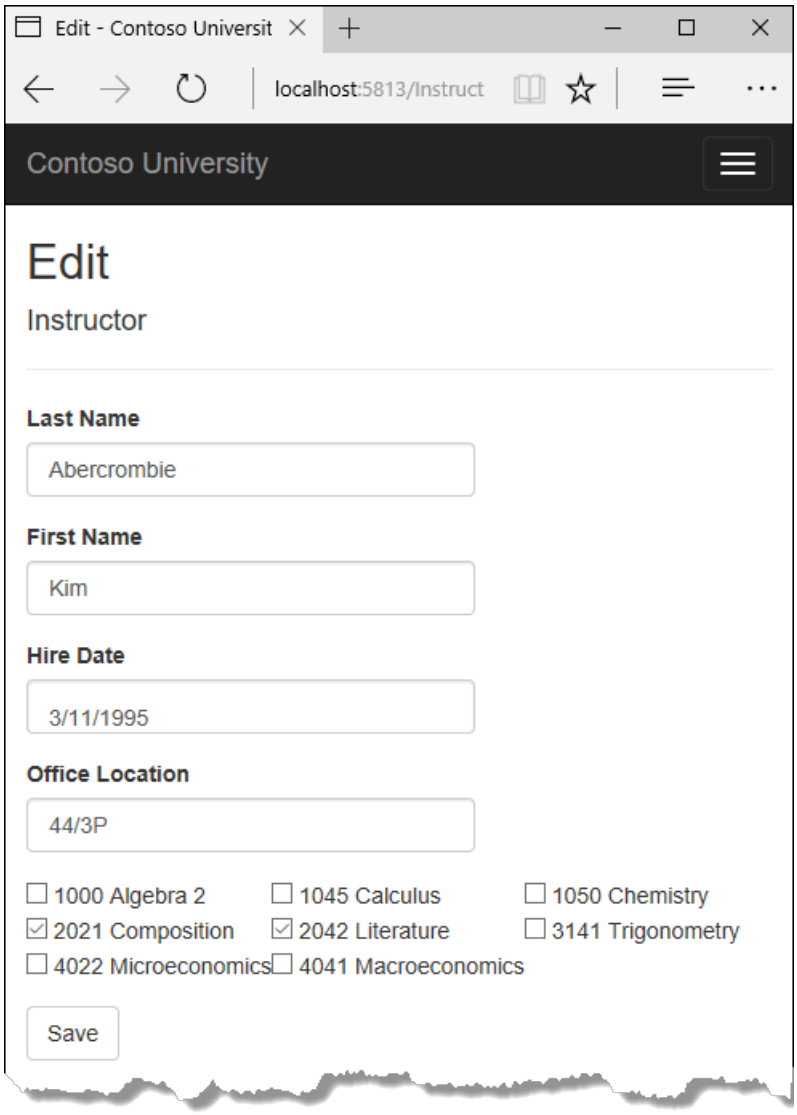
```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they're to be treated as a group. The value attribute of each check box is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the check boxes which are selected.

When the check boxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the app, select the **Instructors** tab, and click **Edit** on an instructor to see the **Edit** page.



Change some course assignments and click Save. The changes you make are reflected on the Index page.

> **NOTE**
> The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update Delete page

In *InstructorsController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to Create page

In *InstructorsController.cs*, delete the HttpGet and HttpPost `Create` methods, and then add the following code in their place:

```
public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}

// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID = instructor.ID, CourseID =
int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The HttpGet `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost `Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the Instructor model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In *Views/Instructor/Create.cshtml*, add an office location text box and check boxes for courses before the Submit button. As in the case of the Edit page, fix the formatting if Visual Studio reformats the code when you paste it.

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label"></label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;
                    List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ? "checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                            @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

Test by running the app and creating an instructor.

Handling Transactions

As explained in the CRUD tutorial, the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- Customized Courses pages
- Added Instructors Edit page
- Added courses to Edit page
- Updated Delete page
- Added office location and courses to Create page

Advance to the next tutorial to learn how to handle concurrency conflicts.

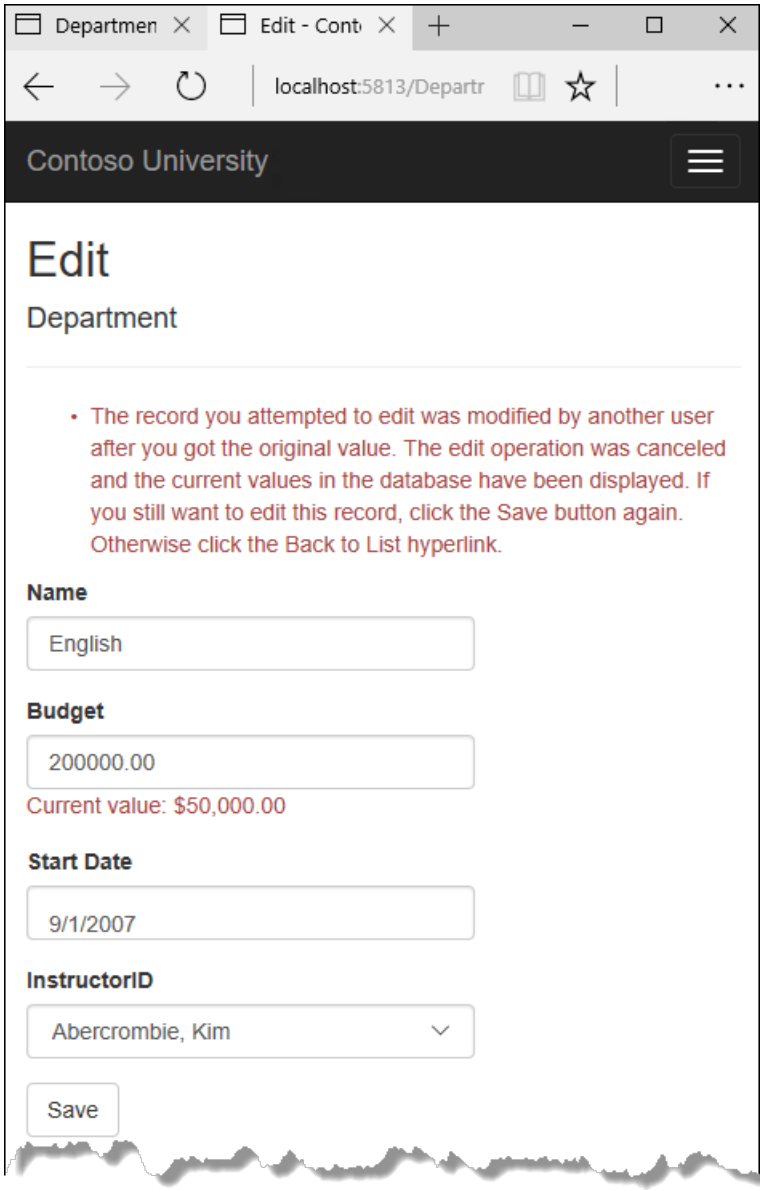
Handle concurrency conflicts

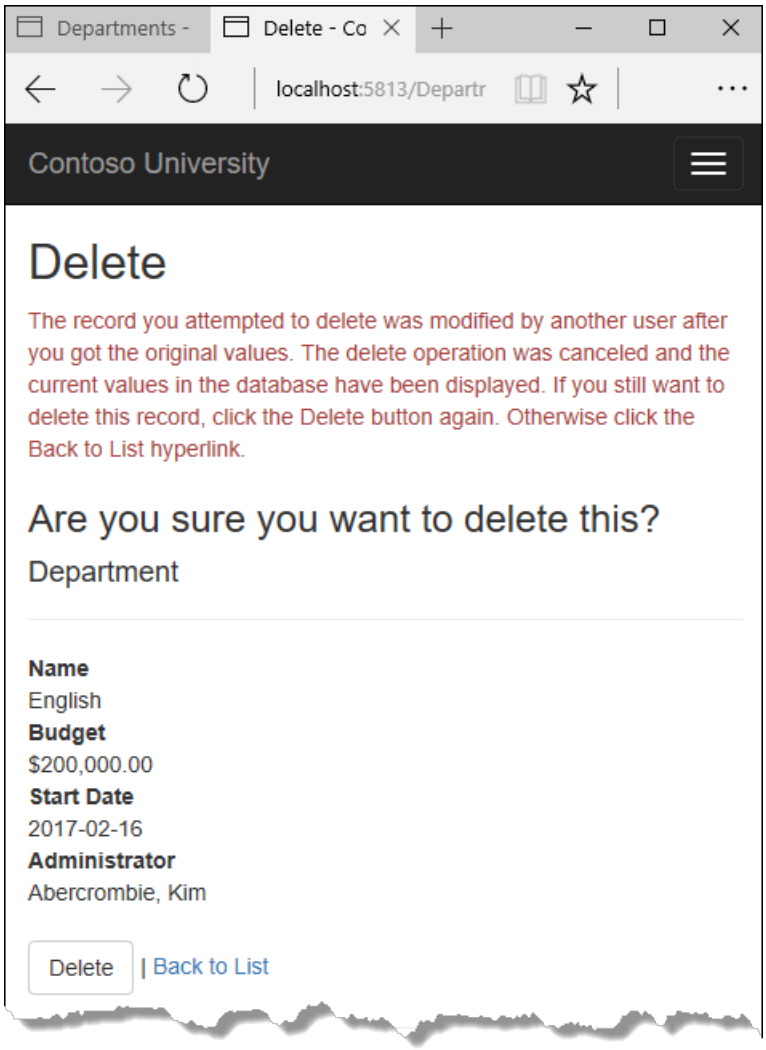
Tutorial: Handle concurrency - ASP.NET MVC with EF Core

9/22/2020 • 18 minutes to read • Edit Online

In earlier tutorials, you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the Department entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.





In this tutorial, you:

- Learn about concurrency conflicts
- Add a tracking property
- Create Departments controller and views
- Update Index view
- Update Edit methods
- Update Edit view
- Test concurrency conflicts
- Update the Delete page
- Update Details and Create views

Prerequisites

- Update related data

Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

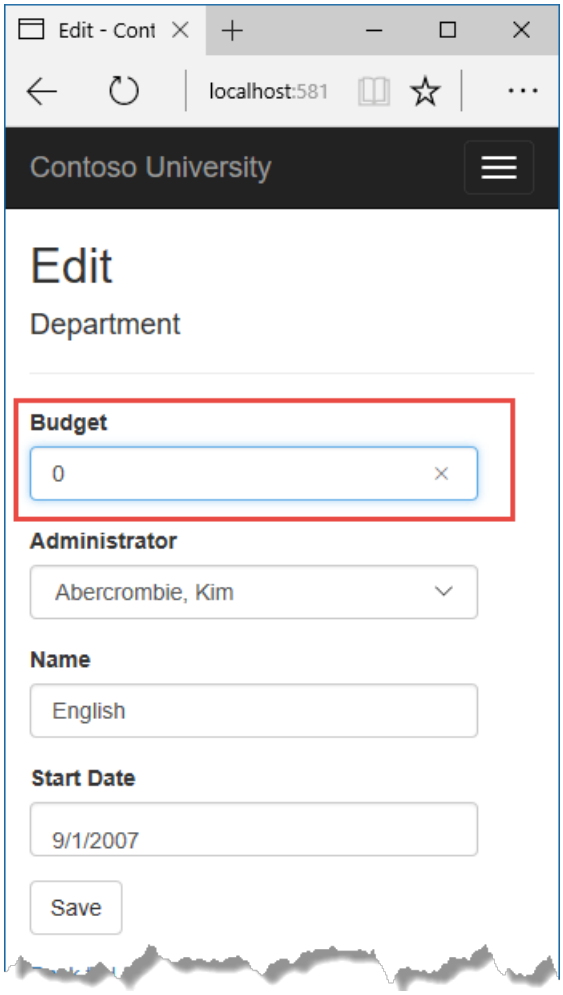
Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

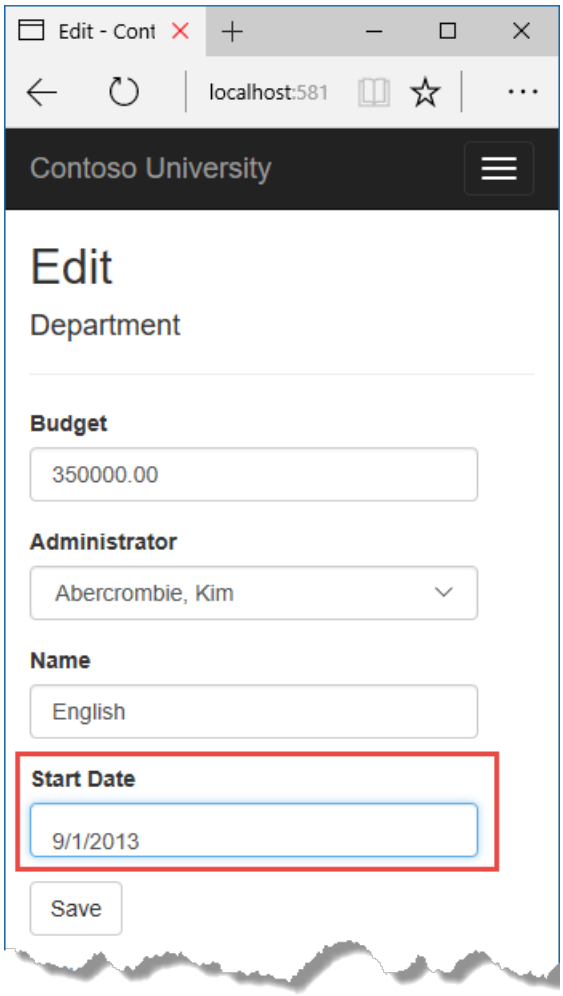
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

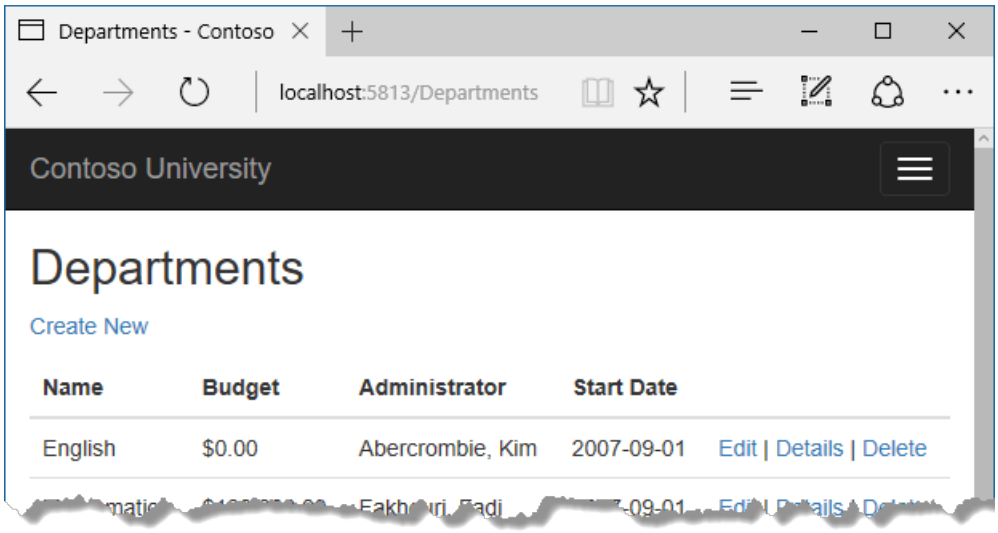
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



Then John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

  In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are

made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent John's change from being updated in the database.

  Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

  The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version) . If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

  As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

  If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a

controller and views, and test to verify that everything works correctly.

Add a tracking property

In *Models/Department.cs*, add a tracking property named RowVersion:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsConcurrencyToken` method (in *Data/SchoolContext.cs*) to specify the tracking property, as shown in the following example:

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();
```

By adding a property you changed the database model, so you need to do another migration.

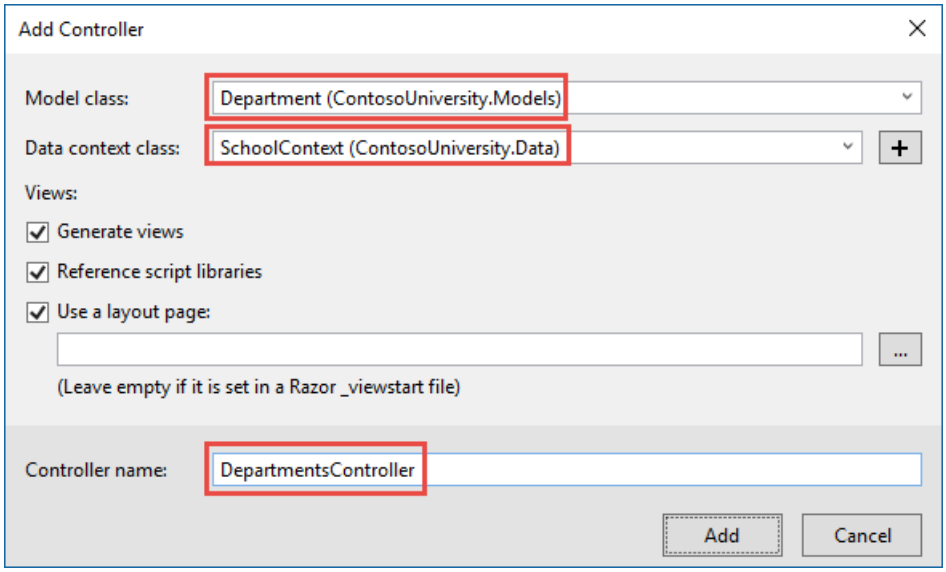
Save your changes and build the project, and then enter the following commands in the command window:

```
dotnet ef migrations add RowVersion
```

```
dotnet ef database update
```

Create Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



In the *DepartmentsController.cs* file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName", department.InstructorID);
```

Update Index view

The scaffolding engine created a RowVersion column in the Index view, but that field shouldn't be displayed.

Replace the code in *Views/Departments/Index.cshtml* with the following code.

```
@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

This changes the heading to "Departments", deletes the RowVersion column, and shows full name instead of first name for the administrator.

Update Edit methods

In both the HttpGet `Edit` method and the `Details` method, add `AsNoTracking`. In the HttpGet `Edit` method, add eager loading for the Administrator.

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .FirstOrDefaultAsync(m => m.DepartmentID == id);
```

Replace the existing code for the HttpPost `Edit` method with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i => i.Administrator).FirstOrDefaultAsync(m
=> m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value: {databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {
```

```
                    ModelState.AddModelError("StartDate", $"Current value: {databaseValues.StartDate:d}");
                }
                if (databaseValues.InstructorID != clientValues.InstructorID)
                {
                    Instructor databaseInstructor = await _context.Instructors.FirstOrDefaultAsync(i => i.ID
== databaseValues.InstructorID);
                    ModelState.AddModelError("InstructorID", $"Current value:
{databaseInstructor?.FullName}");
                }

                ModelState.AddModelError(string.Empty, "The record you attempted to edit "
                        + "was modified by another user after you got the original value. The "
                        + "edit operation was canceled and the current values in the database "
                        + "have been displayed. If you still want to edit this record, click "
                        + "the Save button again. Otherwise click the Back to List hyperlink.");
                departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
                ModelState.Remove("RowVersion");
            }
        }
    }
    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}
```

The code begins by trying to read the department to be updated. If the `FirstOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a department entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the department entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;
```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

```
var exceptionEntry = ex.Entries.Single();
```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update Edit view

In *Views/Departments/Edit.cshtml*, make the following changes:

- Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.
- Add a "Select Administrator" option to the drop-down list.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

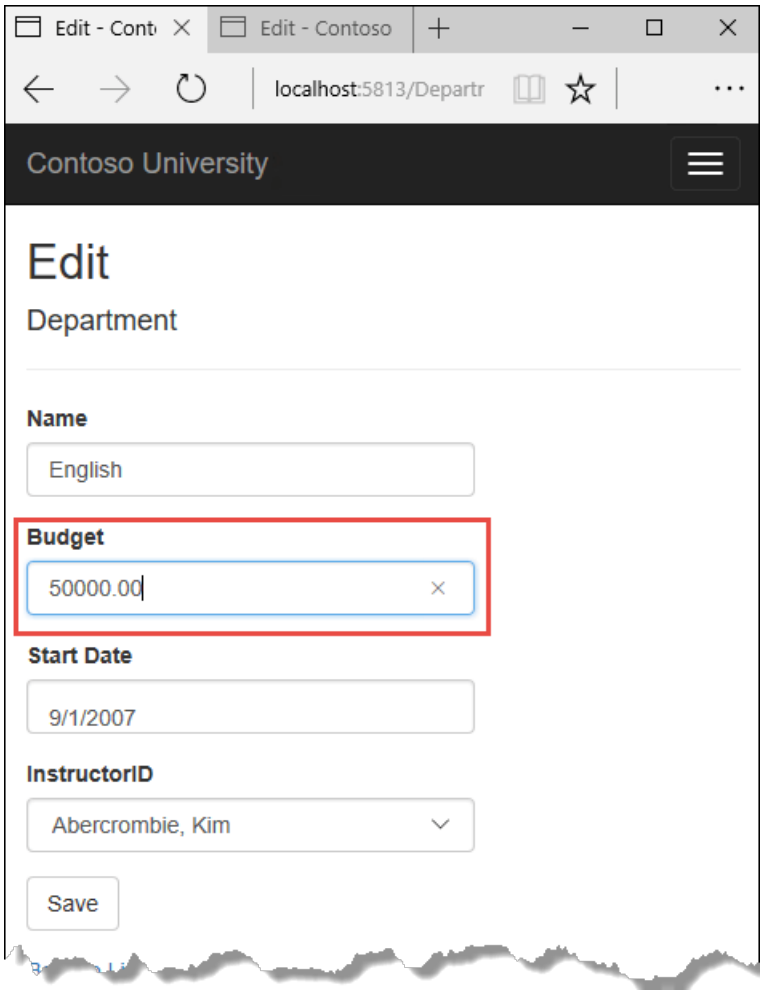
<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test concurrency conflicts

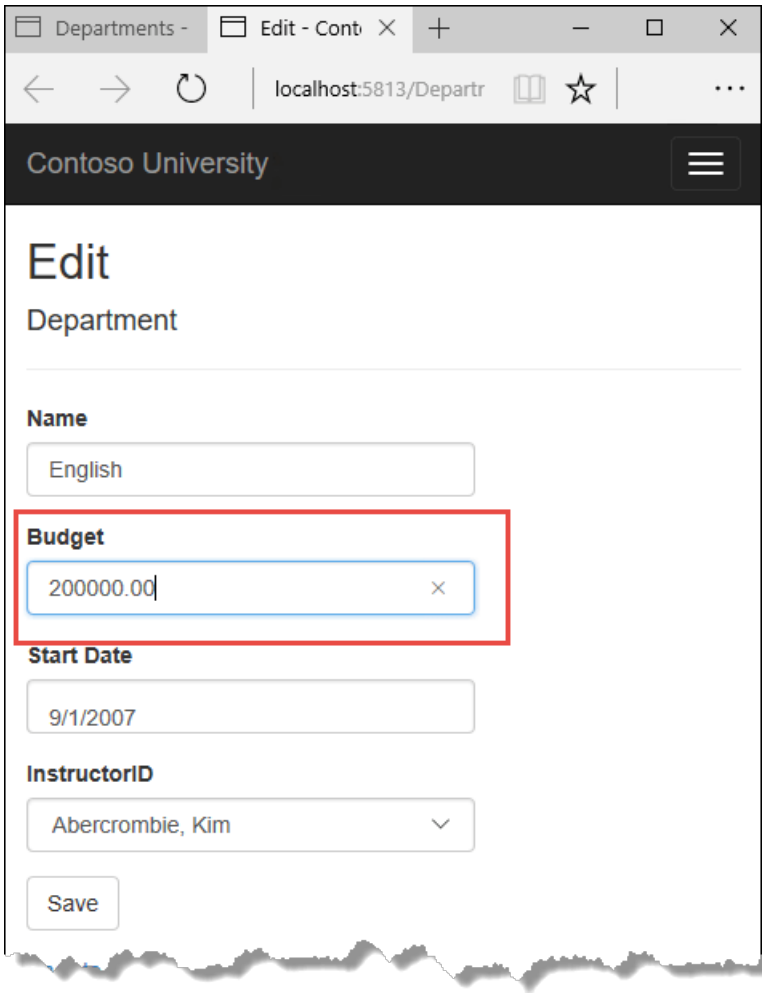
Run the app and go to the Departments Index page. Right-click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.

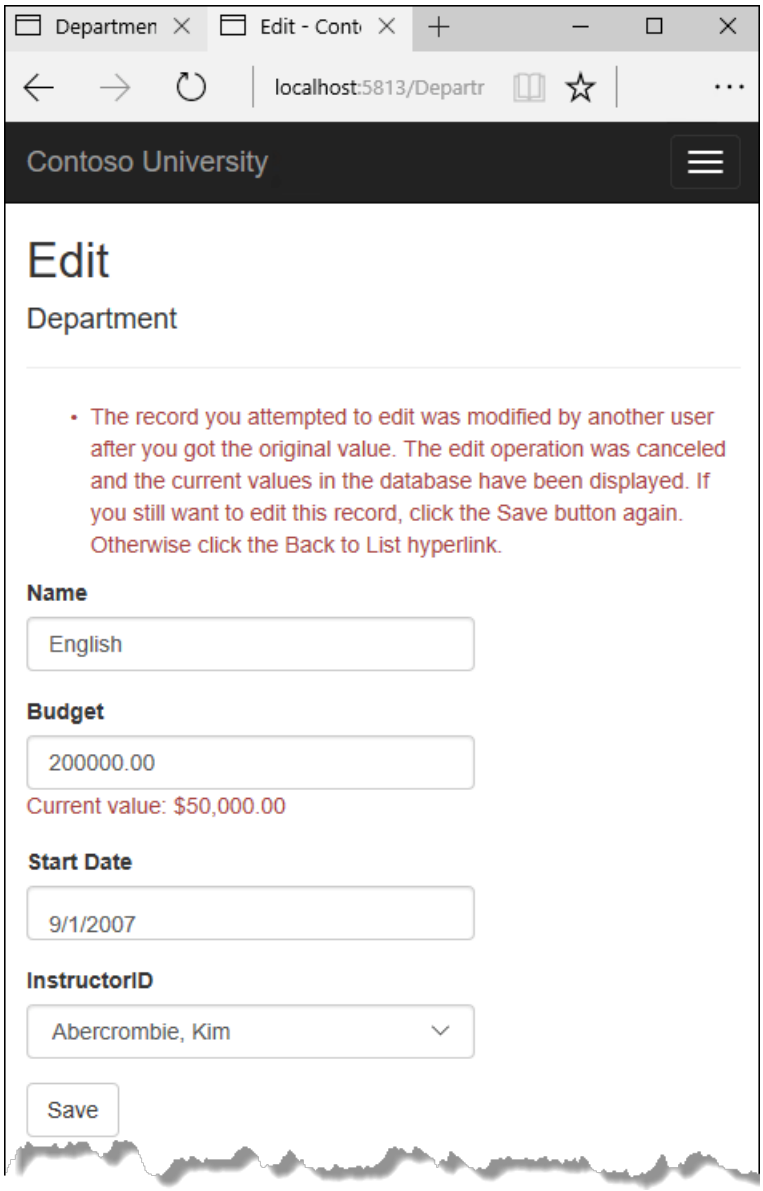


The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the HttpGet `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the HttpPost `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the HttpGet `Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In *DepartmentsController.cs*, replace the HttpGet `Delete` method with the following code:

```
public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

    return View(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the Department does exist, it was changed by another user. In that case, the code sends an error message to the view using `ViewData`.

Replace the code in the HttpPost `Delete` method (named `DeleteConfirmed`) with the following code:

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID == department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError = true, id = department.DepartmentID
});
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a Department entity instance created by the model binder. This gives EF access to the RowVersion property value in addition to the record key.

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In *Views/Departments/Delete.cshtml*, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

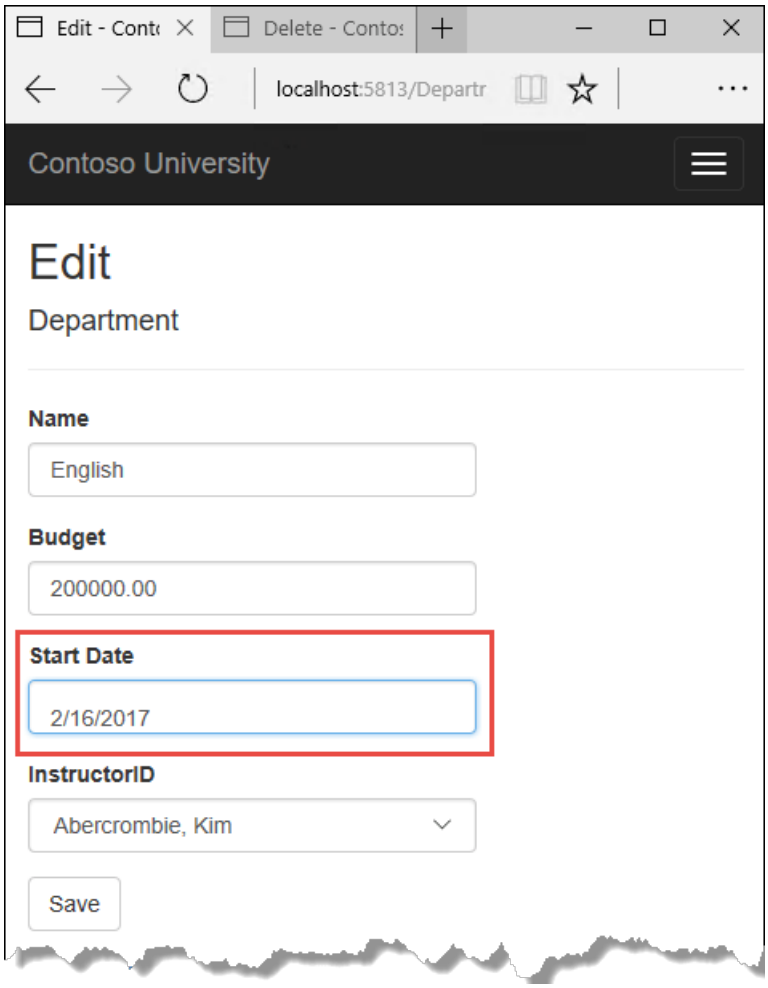
    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

This makes the following changes:

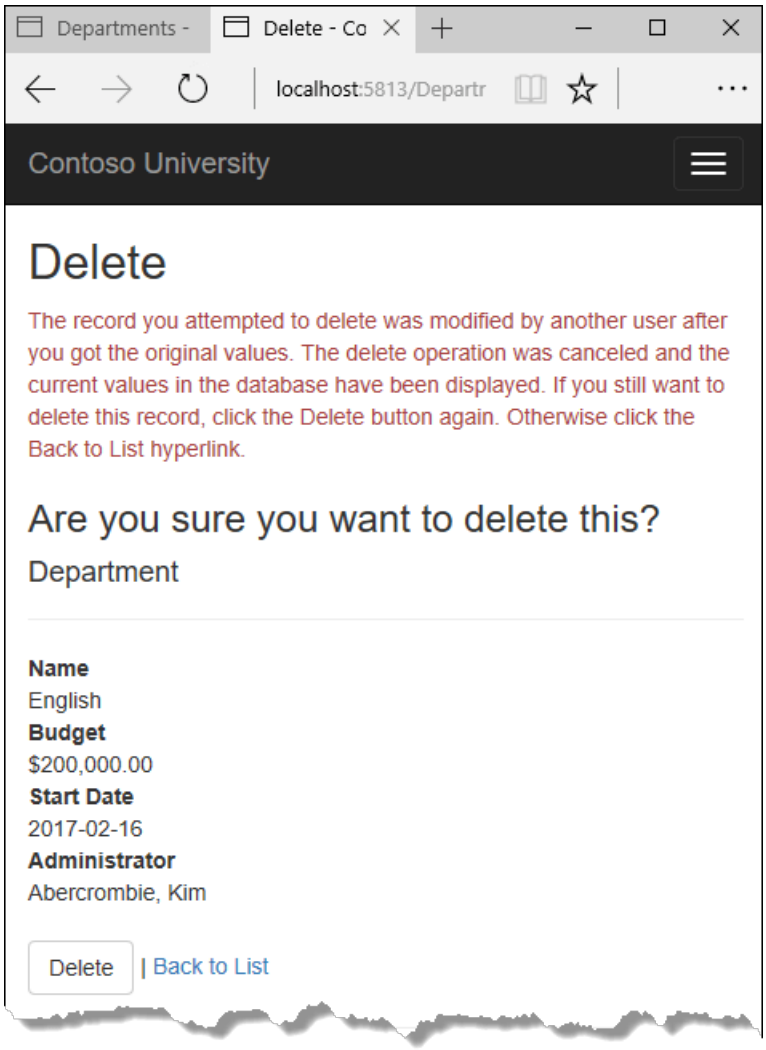
- Adds an error message between the `h2` and `h3` headings.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Removes the RowVersion field.
- Adds a hidden field for the `RowVersion` property.

Run the app and go to the Departments Index page. Right-click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in *Views/Departments/Details.cshtml* to delete the RowVersion column and show the full name of the Administrator.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Replace the code in *Views/Departments/Create.cshtml* to add a Select option to the drop-down list.

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Get the code

Download or view the completed application.

Additional resources

For more information about how to handle concurrency in EF Core, see Concurrency conflicts.

Next steps

In this tutorial, you:

- Learned about concurrency conflicts

- Added a tracking property
- Created Departments controller and views
- Updated Index view
- Updated Edit methods
- Updated Edit view
- Tested concurrency conflicts
- Updated the Delete page
- Updated Details and Create views

Advance to the next tutorial to learn how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

Next: Implement table-per-hierarchy inheritance

Tutorial: Implement inheritance - ASP.NET MVC with EF Core

9/22/2020 • 8 minutes to read • Edit Online

In the previous tutorial, you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

In this tutorial, you:

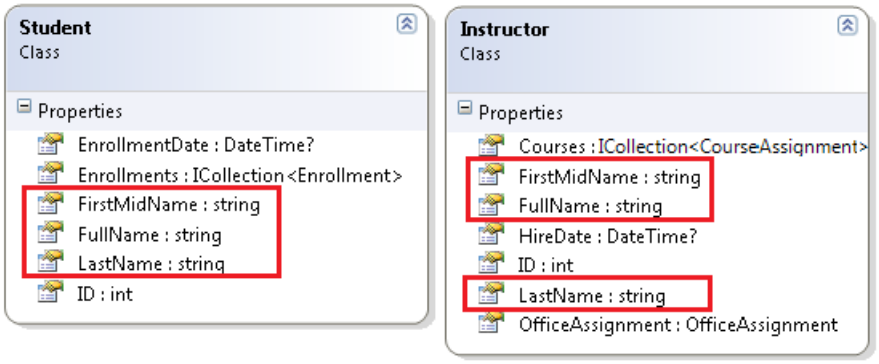
- Map inheritance to database
- Create the Person class
- Update Instructor and Student
- Add Person to the model
- Create and update migrations
- Test the implementation

Prerequisites

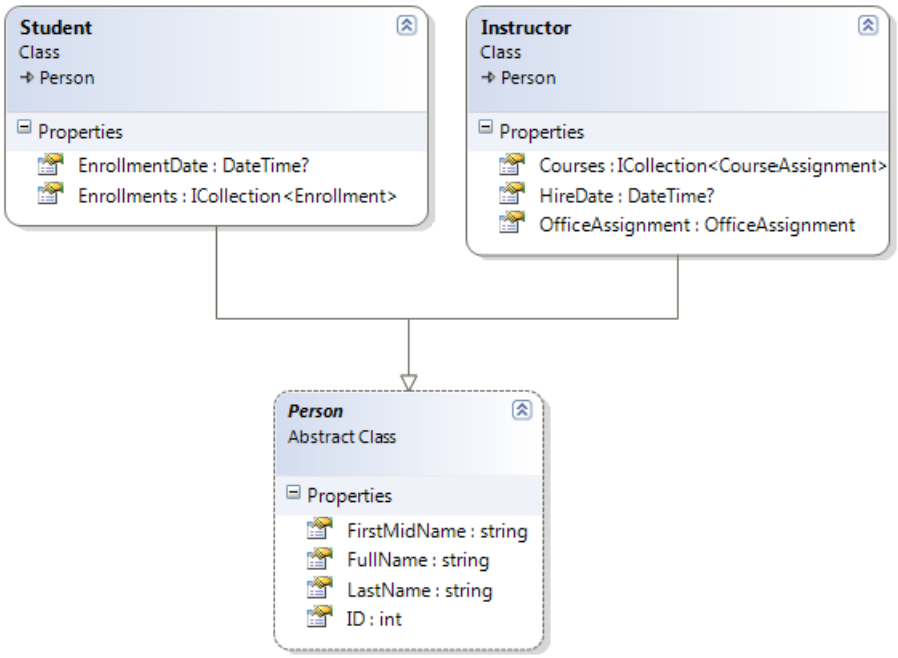
- Handle Concurrency

Map inheritance to database

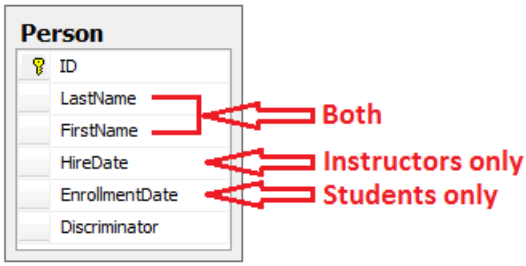
The `Instructor` and `Student` classes in the School data model have several properties that are identical:



Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:



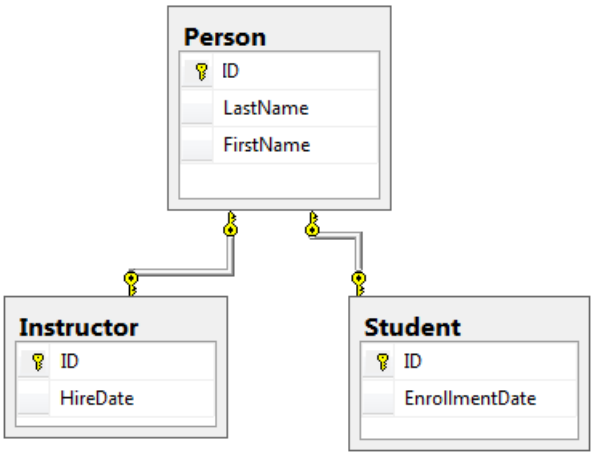
There are several ways this inheritance structure could be represented in the database. You could have a Person table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called table-per-hierarchy (TPH) inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the Person table and have separate Instructor and Student tables with the date fields.

> **WARNING**
>
> Table Per Type (TPT) is not supported by EF Core 3.x, however it is has been implemented in EF Core 5.0.



This pattern of making a database table for each entity class is called table per type (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the Person, Student, and Instructor classes as shown earlier, the Student and Instructor tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

> **TIP**
> Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

Create the Person class

In the Models folder, create Person.cs and replace the template code with the following code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

Update Instructor and Student

In *Instructor.cs*, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Make the same changes in *Student.cs*.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }


        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Add Person to the model

Add the Person entity type to *SchoolContext.cs*. The new lines are highlighted.

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and update migrations

Save your changes and build the project. Then open the command window in the project folder and enter the following command:

```
dotnet ef migrations add Inheritance
```

Don't run the `database update` command yet. That command will result in lost data because it will drop the Instructor table and rename the Student table to Person. You need to provide custom code to preserve existing data.

Open *Migrations/<timestamp>_Inheritance.cs* and replace the `Up` method with the following code:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table: "Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person", nullable: false, maxLength:
128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate, EnrollmentDate,
Discriminator, OldId) SELECT LastName, FirstName, null AS HireDate, EnrollmentDate, 'Student' AS
Discriminator, ID AS OldId FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM dbo.Person WHERE OldId =
Enrollment.StudentId AND Discriminator = 'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
         name: "IX_Enrollment_StudentID",
         table: "Enrollment",
         column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
- Adds nullable EnrollmentDate for students.

- Adds Discriminator column to indicate whether a row is for a student or an instructor.
- Makes HireDate nullable since student rows won't have hire dates.
- Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they will get new primary key values.
- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command:

```
dotnet ef database update
```

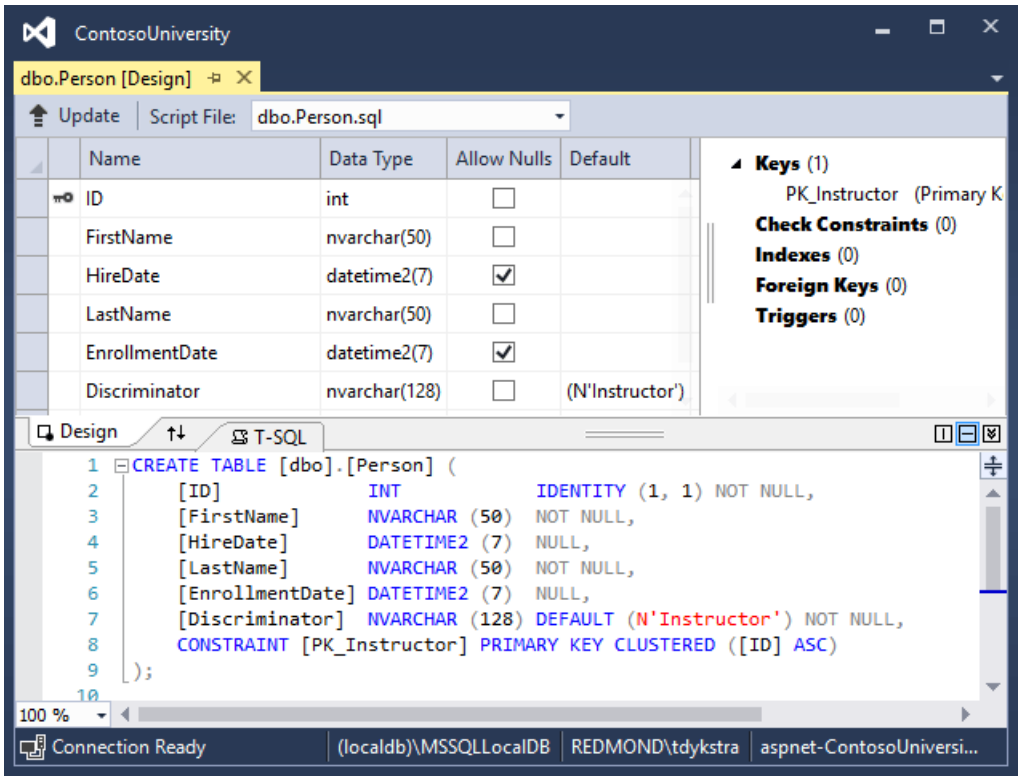
(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

> **NOTE**
> It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `database drop` CLI command.

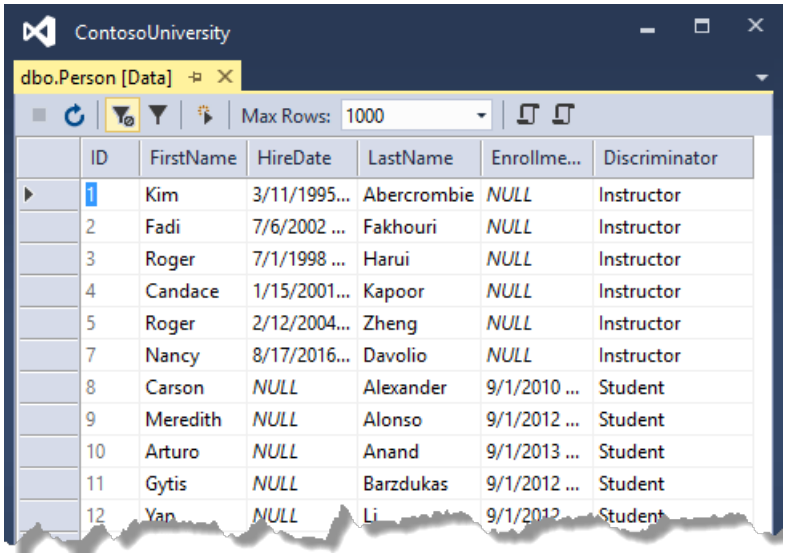
Test the implementation

Run the app and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.



Get the code

Download or view the completed application.

Additional resources

For more information about inheritance in Entity Framework Core, see Inheritance.

Next steps

In this tutorial, you:

- Mapped inheritance to database
- Created the Person class
- Updated Instructor and Student
- Added Person to the model

- Created and update migrations
- Tested the implementation

Advance to the next tutorial to learn how to handle a variety of relatively advanced Entity Framework scenarios.

Next: Advanced topics

Tutorial: Learn about advanced scenarios - ASP.NET MVC with EF Core

9/22/2020 • 13 minutes to read • Edit Online

In the previous tutorial, you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET Core web applications that use Entity Framework Core.

In this tutorial, you:

- Perform raw SQL queries
- Call a query to return entities
- Call a query to return other types
- Call an update query
- Examine SQL queries
- Create an abstraction layer
- Learn about Automatic change detection
- Learn about EF Core source code and development plans
- Learn how to use dynamic LINQ to simplify code

Prerequisites

- Implement Inheritance

Perform raw SQL queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

- Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they're automatically tracked by the database context unless you turn tracking off.
- Use the `Database.ExecuteSqlCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Call a query to return entities

The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type

`TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In *DepartmentsController.cs*, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

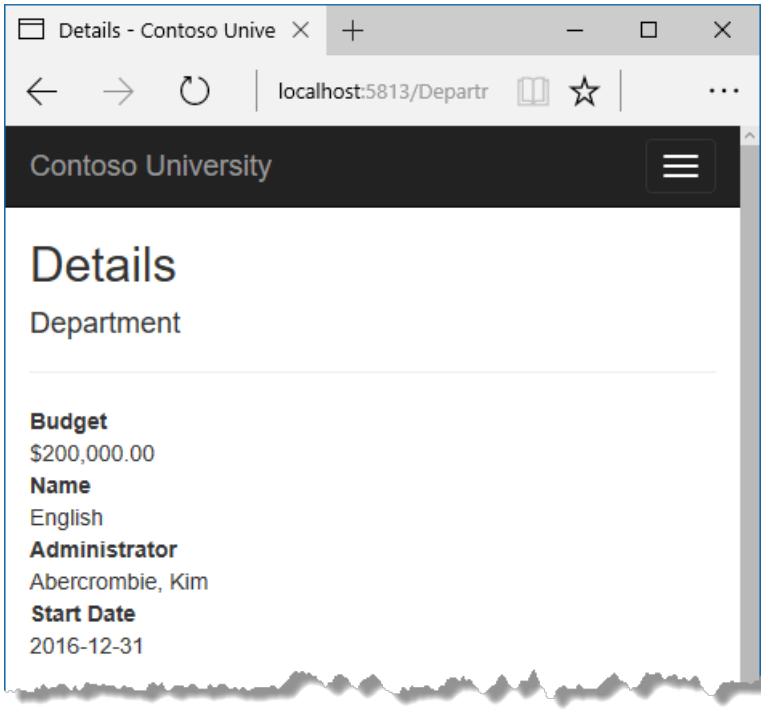
```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Call a query to return other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is write ADO.NET code and get the database connection from EF.

In *HomeController.cs*, replace the `About` method with the following code:

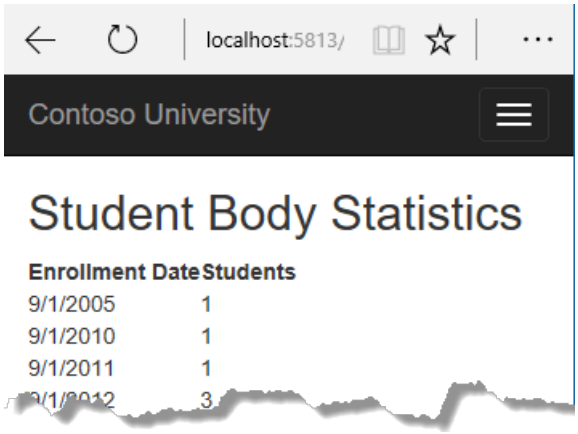
```
public async Task<ActionResult> About()
{
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate = reader.GetDateTime(0), StudentCount
= reader.GetInt32(1) };
                    groups.Add(row);
                }
            }
            reader.Dispose();
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}
```

Add a using statement:

```
using System.Data.Common;
```

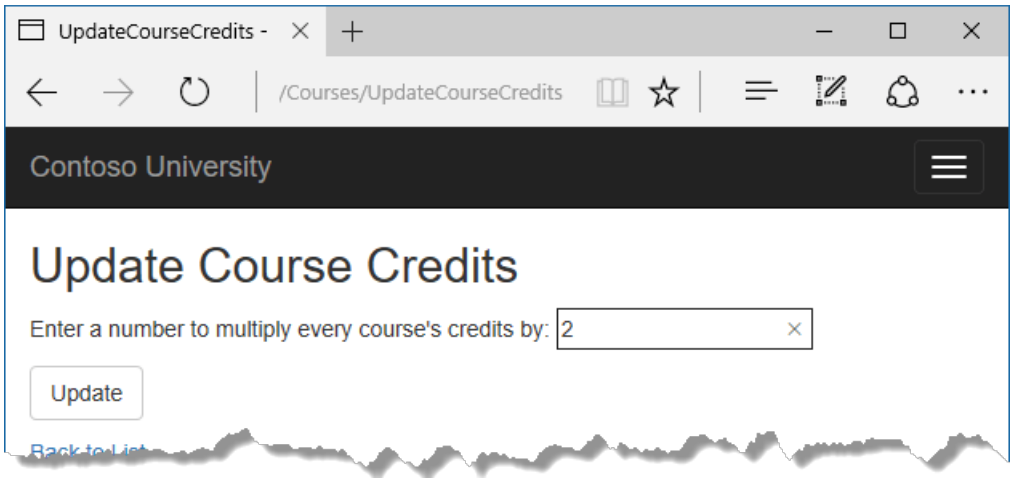
Run the app and go to the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the

change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In *CoursesController.cs*, add UpdateCourseCredits methods for HttpGet and HttpPost:

```
public IActionResult UpdateCourseCredits()
{
    return View();
}
```

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

When the controller processes an HttpGet request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the HttpPost method is called, and multiplier has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

In **Solution Explorer**, right-click the *Views/Courses* folder, and then click **Add > New Item**.

In the **Add New Item** dialog, click **ASP.NET Core** under **Installed** in the left pane, click **Razor View**, and name the new view *UpdateCourseCredits.cshtml*.

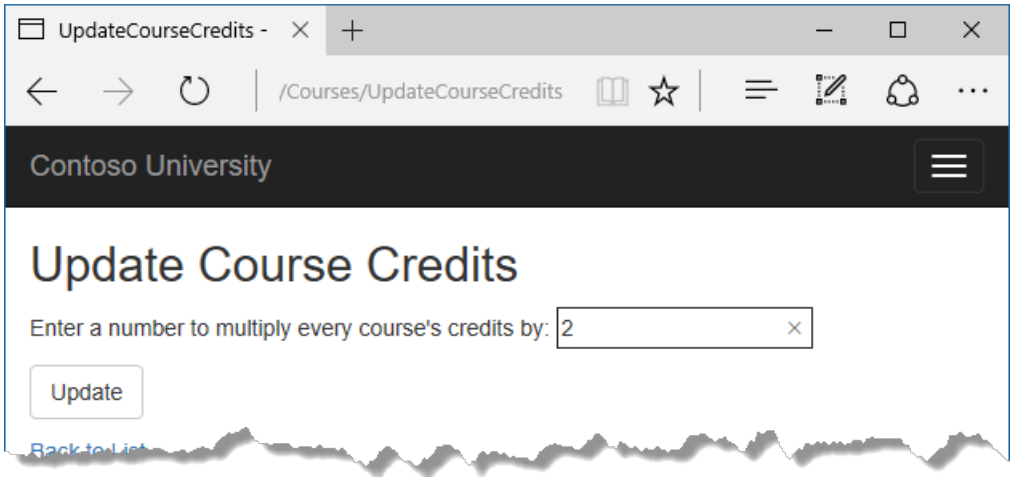
In *Views/Courses/UpdateCourseCredits.cshtml*, replace the template code with the following code:

```
@{
    ViewBag.Title = "UpdateCourseCredits";
}

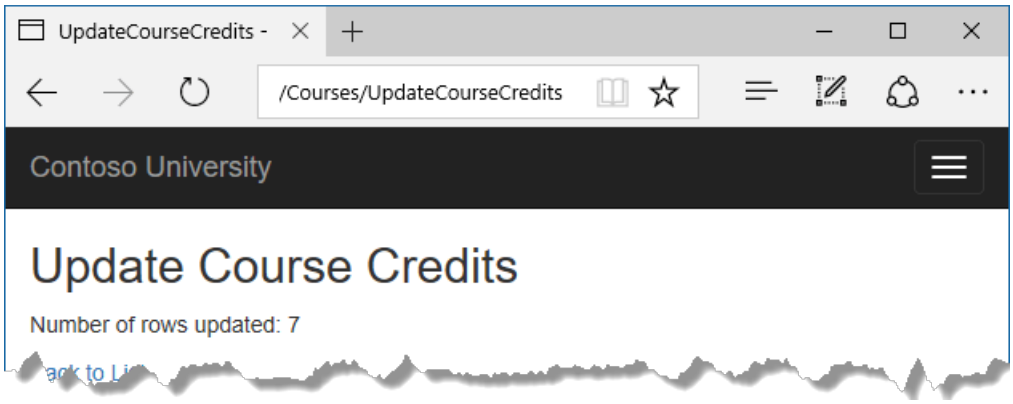
<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
            <p>
                Enter a number to multiply every course's credits by: @Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default" />
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Courses/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see Raw SQL Queries.

Examine SQL queries

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open *StudentsController.cs* and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the app in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (56ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed DbCommand (122ms) [Parameters=
[@__id_0='?'], CommandType='Text', CommandTimeout='30']
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID], [s.Enrollments].[Grade], [s.Enrollments].
[StudentID], [e.Course].[CourseID], [e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] = [e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`) from the Person table. The `SingleOrDefaultAsync` method doesn't resolve to 1 row on the server. Here's why:

- If the query would return multiple rows, the method returns null.
- To determine whether the query would return multiple rows, EF has to check if it returns at least 2.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Create an abstraction layer

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns isn't always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series.

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see Test with InMemory.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- DbContext.SaveChanges
- DbContext.Entry
- ChangeTracker.Entries

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

EF Core source code and development plans

The Entity Framework Core source is at https://github.com/dotnet/efcore. The EF Core repository contains nightly builds, issue tracking, feature specs, design meeting notes, and the roadmap for future development. You can file or find bugs, and contribute.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the scaffold-dbcontext command. See the getting-started tutorial.

Use dynamic LINQ to simplify code

The third tutorial in this series shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] =
        String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
    ViewData["DateSortParm"] =
        sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" : "EnrollmentDate";

    if (searchString != null)
    {
        pageNumber = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }

    if (string.IsNullOrEmpty(sortOrder))
    {
        sortOrder = "LastName";
    }

    bool descending = false;
    if (sortOrder.EndsWith("_desc"))
    {
        sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
        descending = true;
    }

    if (descending)
    {
        students = students.OrderByDescending(e => EF.Property<object>(e, sortOrder));
    }
    else
    {
        students = students.OrderBy(e => EF.Property<object>(e, sortOrder));
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
        pageNumber ?? 1, pageSize));
}
```

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials. John Parente and Paul Goldman worked on updating the tutorial for ASP.NET Core 2.2.

Troubleshoot common errors

ContosoUniversity.dll used by another process

Error message:

> Cannot open '...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in *appsettings.json*. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the `database drop` CLI command:

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

> A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

Get the code

Download or view the completed application.

Additional resources

For more information about EF Core, see the Entity Framework Core documentation. A book is also available: Entity Framework Core in Action.

For information on how to deploy a web app, see Host and deploy ASP.NET Core.

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see Introduction to ASP.NET Core.

Next steps

In this tutorial, you:

- Performed raw SQL queries
- Called a query to return entities
- Called a query to return other types
- Called an update query
- Examined SQL queries
- Created an abstraction layer
- Learned about Automatic change detection
- Learned about EF Core source code and development plans
- Learned how to use dynamic LINQ to simplify code

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET Core MVC application. This series worked with a new database; an alternative is to reverse engineer a model from an existing database.

Tutorial: EF Core with MVC, existing database

ASP.NET Core and Entity Framework 6

9/22/2020 • 3 minutes to read • Edit Online

By Patrick Goode

Using Entity Framework 6 with ASP.NET Core

Entity Framework Core should be used for new development. The download sample uses Entity Framework 6 (EF6), which can be used to migrate exiting apps to ASP.NET Core.

Additional resources

- Entity Framework - Code-Based Configuration

By Paweł Grudzień, Damien Pontifex, and Tom Dykstra

This article shows how to use Entity Framework 6 in an ASP.NET Core application.

Overview

To use Entity Framework 6, your project has to compile against .NET Framework, as Entity Framework 6 doesn't support .NET Core. If you need cross-platform features you will need to upgrade to Entity Framework Core.

The recommended way to use Entity Framework 6 in an ASP.NET Core application is to put the EF6 context and model classes in a class library project that targets .NET Framework. Add a reference to the class library from the ASP.NET Core project. See the sample Visual Studio solution with EF6 and ASP.NET Core projects.

You can't put an EF6 context in an ASP.NET Core project because .NET Core projects don't support all of the functionality that EF6 commands such as *Enable-Migrations* require.

Regardless of project type in which you locate your EF6 context, only EF6 command-line tools work with an EF6 context. For example, `Scaffold-DbContext` is only available in Entity Framework Core. If you need to do reverse engineering of a database into an EF6 model, see /ef/ef6/modeling/code-first/workflows/existing-database.

Reference full framework and EF6 in the ASP.NET Core project

Your ASP.NET Core project needs to target .NET Framework and reference EF6. For example, the *.csproj* file of your ASP.NET Core project will look similar to the following example (only relevant parts of the file are shown).

```
<PropertyGroup>
  <TargetFramework>net452</TargetFramework>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>MVCCore</AssemblyName>
  <OutputType>Exe</OutputType>
  <PackageId>MVCCore</PackageId>
</PropertyGroup>
```

When creating a new project, use the **ASP.NET Core Web Application (.NET Framework)** template.

Handle connection strings

The EF6 command-line tools that you'll use in the EF6 class library project require a default constructor so they can instantiate the context. But you'll probably want to specify the connection string to use in the ASP.NET Core project,

in which case your context constructor must have a parameter that lets you pass in the connection string. Here's an example.

```
public class SchoolContext : DbContext
{
    public SchoolContext(string connString) : base(connString)
    {
    }
```

Since your EF6 context doesn't have a parameterless constructor, your EF6 project has to provide an implementation of /dotnet/api/system.data.entity.infrastructure.idbcontextfactory-1?view=entity-framework-6.2.0. The EF6 command-line tools will find and use that implementation so they can instantiate the context. Here's an example.

```
public class SchoolContextFactory : IDbContextFactory<SchoolContext>
{
    public SchoolContext Create()
    {
        return new EF6.SchoolContext("Server=
(localdb)\\mssqllocaldb;Database=EF6MVCCore;Trusted_Connection=True;MultipleActiveResultSets=true");
    }
}
```

In this sample code, the `IDbContextFactory` implementation passes in a hard-coded connection string. This is the connection string that the command-line tools will use. You'll want to implement a strategy to ensure that the class library uses the same connection string that the calling application uses. For example, you could get the value from an environment variable in both projects.

Set up dependency injection in the ASP.NET Core project

In the Core project's *Startup.cs* file, set up the EF6 context for dependency injection (DI) in `ConfigureServices`. EF context objects should be scoped for a per-request lifetime.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
    services.AddScoped<SchoolContext>(_ => new
SchoolContext(Configuration.GetConnectionString("DefaultConnection")));
}
```

You can then get an instance of the context in your controllers by using DI. The code is similar to what you'd write for an EF Core context:

```
public class StudentsController : Controller
{
    private readonly SchoolContext _context;

    public StudentsController(SchoolContext context)
    {
        _context = context;
    }
```

Sample application

For a working sample application, see the sample Visual Studio solution that accompanies this article.

This sample can be created from scratch by the following steps in Visual Studio:

- Create a solution.
- **Add** > **New Project** > **Web** > **ASP.NET Core Web Application**
  - In project template selection dialog, select API and .NET Framework in dropdown
- **Add** > **New Project** > **Windows Desktop** > **Class Library (.NET Framework)**
- In **Package Manager Console** (PMC) for both projects, run the command `Install-Package Entityframework`.
- In the class library project, create data model classes and a context class, and an implementation of `IDbContextFactory`.
- In PMC for the class library project, run the commands `Enable-Migrations` and `Add-Migration Initial`. If you have set the ASP.NET Core project as the startup project, add `-StartupProjectName EF6` to these commands.
- In the Core project, add a project reference to the class library project.
- In the Core project, in *Startup.cs*, register the context for DI.
- In the Core project, in *appsettings.json*, add the connection string.
- In the Core project, add a controller and view(s) to verify that you can read and write data. (Note that ASP.NET Core MVC scaffolding won't work with the EF6 context referenced from the class library.)

Host and deploy ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

In general, to deploy an ASP.NET Core app to a hosting environment:

- Deploy the published app to a folder on the hosting server.
- Set up a process manager that starts the app when requests arrive and restarts the app after it crashes or the server reboots.
- For configuration of a reverse proxy, set up a reverse proxy to forward requests to the app.

Publish to a folder

The dotnet publish command compiles app code and copies the files required to run the app into a *publish* folder. When deploying from Visual Studio, the `dotnet publish` step occurs automatically before the files are copied to the deployment destination.

Folder contents

The *publish* folder contains one or more app assembly files, dependencies, and optionally the .NET runtime.

A .NET Core app can be published as *self-contained deployment* or *framework-dependent deployment*. If the app is self-contained, the assembly files that contain the .NET runtime are included in the *publish* folder. If the app is framework-dependent, the .NET runtime files aren't included because the app has a reference to a version of .NET that's installed on the server. The default deployment model is framework-dependent. For more information, see .NET Core application deployment.

In addition to *.exe* and *.dll* files, the *publish* folder for an ASP.NET Core app typically contains configuration files, static assets, and MVC views. For more information, see ASP.NET Core directory structure.

Set up a process manager

An ASP.NET Core app is a console app that must be started when a server boots and restarted if it crashes. To automate starts and restarts, a process manager is required. The most common process managers for ASP.NET Core are:

- Linux
  - Nginx
  - Apache
- Windows
  - IIS
  - Windows Service

Set up a reverse proxy

If the app uses the Kestrel server, Nginx, Apache, or IIS can be used as a reverse proxy server. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.

Either configuration—with or without a reverse proxy server—is a supported hosting configuration. For more information, see When to use Kestrel with a reverse proxy.

Proxy server and load balancer scenarios

Additional configuration might be required for apps hosted behind proxy servers and load balancers. Without additional configuration, an app might not have access to the scheme (HTTP/HTTPS) and the remote IP address where a request originated. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Use Visual Studio and MSBuild to automate deployments

Deployment often requires additional tasks besides copying the output from dotnet publish to a server. For example, extra files might be required or excluded from the *publish* folder. Visual Studio uses MSBuild for web deployment, and MSBuild can be customized to do many other tasks during deployment. For more information, see Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment and the Using MSBuild and Team Foundation Build book.

By using the Publish Web feature or built-in Git support, apps can be deployed directly from Visual Studio to the Azure App Service. Azure DevOps Services supports continuous deployment to Azure App Service. For more information, see DevOps with ASP.NET Core and Azure.

Publish to Azure

See Publish an ASP.NET Core app to Azure with Visual Studio for instructions on how to publish an app to Azure using Visual Studio. An additional example is provided by Create an ASP.NET Core web app in Azure.

Publish with MSDeploy on Windows

See Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment for instructions on how to publish an app with a Visual Studio publish profile, including from a Windows command prompt using the dotnet msbuild command.

Internet Information Services (IIS)

For deployments to Internet Information Services (IIS) with configuration provided by the *web.config* file, see the articles under Host ASP.NET Core on Windows with IIS.

Host in a web farm

For information on configuration for hosting ASP.NET Core apps in a web farm environment (for example, deployment of multiple instances of your app for scalability), see Host ASP.NET Core in a web farm.

Host on Docker

For more information, see Host ASP.NET Core in Docker containers.

Perform health checks

Use Health Check Middleware to perform health checks on an app and its dependencies. For more information, see Health checks in ASP.NET Core.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- ASP.NET Hosting

In general, to deploy an ASP.NET Core app to a hosting environment:

- Deploy the published app to a folder on the hosting server.

- Set up a process manager that starts the app when requests arrive and restarts the app after it crashes or the server reboots.
- For configuration of a reverse proxy, set up a reverse proxy to forward requests to the app.

Publish to a folder

The dotnet publish command compiles app code and copies the files required to run the app into a *publish* folder. When deploying from Visual Studio, the `dotnet publish` step occurs automatically before the files are copied to the deployment destination.

Folder contents

The *publish* folder contains one or more app assembly files, dependencies, and optionally the .NET runtime.

A .NET Core app can be published as *self-contained deployment* or *framework-dependent deployment*. If the app is self-contained, the assembly files that contain the .NET runtime are included in the *publish* folder. If the app is framework-dependent, the .NET runtime files aren't included because the app has a reference to a version of .NET that's installed on the server. The default deployment model is framework-dependent. For more information, see .NET Core application deployment.

In addition to *.exe* and *.dll* files, the *publish* folder for an ASP.NET Core app typically contains configuration files, static assets, and MVC views. For more information, see ASP.NET Core directory structure.

Set up a process manager

An ASP.NET Core app is a console app that must be started when a server boots and restarted if it crashes. To automate starts and restarts, a process manager is required. The most common process managers for ASP.NET Core are:

- Linux
  - Nginx
  - Apache
- Windows
  - IIS
  - Windows Service

Set up a reverse proxy

If the app uses the Kestrel server, Nginx, Apache, or IIS can be used as a reverse proxy server. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.

Either configuration—with or without a reverse proxy server—is a supported hosting configuration. For more information, see When to use Kestrel with a reverse proxy.

Proxy server and load balancer scenarios

Additional configuration might be required for apps hosted behind proxy servers and load balancers. Without additional configuration, an app might not have access to the scheme (HTTP/HTTPS) and the remote IP address where a request originated. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Use Visual Studio and MSBuild to automate deployments

Deployment often requires additional tasks besides copying the output from dotnet publish to a server. For example, extra files might be required or excluded from the *publish* folder. Visual Studio uses MSBuild for web

deployment, and MSBuild can be customized to do many other tasks during deployment. For more information, see Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment and the Using MSBuild and Team Foundation Build book.

By using the Publish Web feature or built-in Git support, apps can be deployed directly from Visual Studio to the Azure App Service. Azure DevOps Services supports continuous deployment to Azure App Service. For more information, see DevOps with ASP.NET Core and Azure.

Publish to Azure

See Publish an ASP.NET Core app to Azure with Visual Studio for instructions on how to publish an app to Azure using Visual Studio. An additional example is provided by Create an ASP.NET Core web app in Azure.

Publish with MSDeploy on Windows

See Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment for instructions on how to publish an app with a Visual Studio publish profile, including from a Windows command prompt using the dotnet msbuild command.

Internet Information Services (IIS)

For deployments to Internet Information Services (IIS) with configuration provided by the *web.config* file, see the articles under Host ASP.NET Core on Windows with IIS.

Host in a web farm

For information on configuration for hosting ASP.NET Core apps in a web farm environment (for example, deployment of multiple instances of your app for scalability), see Host ASP.NET Core in a web farm.

Host on Docker

For more information, see Host ASP.NET Core in Docker containers.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- ASP.NET Hosting

Deploy ASP.NET Core apps to Azure App Service

9/22/2020 • 13 minutes to read • Edit Online

Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core.

Useful resources

App Service Documentation is the home for Azure Apps documentation, tutorials, samples, how-to guides, and other resources. Two notable tutorials that pertain to hosting ASP.NET Core apps are:

Create an ASP.NET Core web app in Azure
Use Visual Studio to create and deploy an ASP.NET Core web app to Azure App Service on Windows.

Create an ASP.NET Core app in App Service on Linux
Use the command line to create and deploy an ASP.NET Core web app to Azure App Service on Linux.

See the ASP.NET Core on App Service Dashboard for the version of ASP.NET Core available on Azure App service.

Subscribe to the App Service Announcements repository and monitor the issues. The App Service team regularly posts announcements and scenarios arriving in App Service.

The following articles are available in ASP.NET Core documentation:

Publish an ASP.NET Core app to Azure with Visual Studio
Learn how to publish an ASP.NET Core app to Azure App Service using Visual Studio.

Continuous deployment to Azure with Visual Studio and Git with ASP.NET Core
Learn how to create an ASP.NET Core web app using Visual Studio and deploy it to Azure App Service using Git for continuous deployment.

Create your first pipeline
Set up a CI build for an ASP.NET Core app, then create a continuous deployment release to Azure App Service.

Azure Web App sandbox
Discover Azure App Service runtime execution limitations enforced by the Azure Apps platform.

Troubleshoot and debug ASP.NET Core projects
Understand and troubleshoot warnings and errors with ASP.NET Core projects.

Application configuration

Platform

The platform architecture (x86/x64) of an App Services app is set in the app's settings in the Azure Portal for apps that are hosted on an A-series compute (Basic) or higher hosting tier. Confirm that the app's publish settings (for example, in the Visual Studio publish profile (.pubxml)) match the setting in the app's service configuration in the Azure Portal.

Runtimes for 64-bit (x64) and 32-bit (x86) apps are present on Azure App Service. The .NET Core SDK available on App Service is 32-bit, but you can deploy 64-bit apps built locally using the Kudu console or the publish process in Visual Studio. For more information, see the Publish and deploy the app section.

For apps with native dependencies, runtimes for 32-bit (x86) apps are present on Azure App Service. The .NET Core SDK available on App Service is 32-bit.

For more information on .NET Core framework components and distribution methods, such as information on the .NET Core runtime and the .NET Core SDK, see About .NET Core: Composition.

Packages

Include the following NuGet packages to provide automatic logging features for apps deployed to Azure App Service:

- Microsoft.AspNetCore.AzureAppServices.HostingStartup uses IHostingStartup to provide ASP.NET Core light-up integration with Azure App Service. The added logging features are provided by the `Microsoft.AspNetCore.AzureAppServicesIntegration` package.
- Microsoft.AspNetCore.AzureAppServicesIntegration executes AddAzureWebAppDiagnostics to add Azure App Service diagnostics logging providers in the `Microsoft.Extensions.Logging.AzureAppServices` package.
- Microsoft.Extensions.Logging.AzureAppServices provides logger implementations to support Azure App Service diagnostics logs and log streaming features.

The preceding packages aren't available from the Microsoft.AspNetCore.App metapackage. Apps that target .NET Framework or reference the `Microsoft.AspNetCore.App` metapackage must explicitly reference the individual packages in the app's project file.

Override app configuration using the Azure Portal

App settings in the Azure Portal permit you to set environment variables for the app. Environment variables can be consumed by the Environment Variables Configuration Provider.

When an app setting is created or modified in the Azure Portal and the **Save** button is selected, the Azure App is restarted. The environment variable is available to the app after the service restarts.

When an app uses the Generic Host, environment variables are loaded into the app's configuration when CreateDefaultBuilder is called to build the host. For more information, see .NET Generic Host and the Environment Variables Configuration Provider.

App settings in the Azure Portal permit you to set environment variables for the app. Environment variables can be consumed by the Environment Variables Configuration Provider.

When an app setting is created or modified in the Azure Portal and the **Save** button is selected, the Azure App is restarted. The environment variable is available to the app after the service restarts.

When an app uses the Web Host, environment variables are loaded into the app's configuration when CreateDefaultBuilder is called to build the host. For more information, see ASP.NET Core Web Host and the Environment Variables Configuration Provider.

Proxy server and load balancer scenarios

The IIS Integration Middleware, which configures Forwarded Headers Middleware when hosting out-of-process, and the ASP.NET Core Module are configured to forward the scheme (HTTP/HTTPS) and the remote IP address where the request originated. Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Monitoring and logging

ASP.NET Core apps deployed to App Service automatically receive an App Service extension, **ASP.NET Core Logging Integration**. The extension enables logging integration for ASP.NET Core apps on Azure App Service.

ASP.NET Core apps deployed to App Service automatically receive an App Service extension, **ASP.NET Core Logging Extensions**. The extension enables logging integration for ASP.NET Core apps on Azure App Service.

For monitoring, logging, and troubleshooting information, see the following articles:

Monitor apps in Azure App Service
Learn how to review quotas and metrics for apps and App Service plans.

Enable diagnostics logging for apps in Azure App Service
Discover how to enable and access diagnostic logging for HTTP status codes, failed requests, and web server activity.

Handle errors in ASP.NET Core
Understand common approaches to handling errors in ASP.NET Core apps.

Troubleshoot ASP.NET Core on Azure App Service and IIS
Learn how to diagnose issues with Azure App Service deployments with ASP.NET Core apps.

Common errors reference for Azure App Service and IIS with ASP.NET Core
See the common deployment configuration errors for apps hosted by Azure App Service/IIS with troubleshooting advice.

Data Protection key ring and deployment slots

Data Protection keys are persisted to the *%HOME%\ASP.NET\DataProtection-Keys* folder. This folder is backed by network storage and is synchronized across all machines hosting the app. Keys aren't protected at rest. This folder supplies the key ring to all instances of an app in a single deployment slot. Separate deployment slots, such as Staging and Production, don't share a key ring.

When swapping between deployment slots, any system using data protection won't be able to decrypt stored data using the key ring inside the previous slot. ASP.NET Cookie Middleware uses data protection to protect its cookies. This leads to users being signed out of an app that uses the standard ASP.NET Cookie Middleware. For a slot-independent key ring solution, use an external key ring provider, such as:

- Azure Blob Storage
- Azure Key Vault
- SQL store
- Redis cache

For more information, see Key storage providers in ASP.NET Core.

Deploy an ASP.NET Core app that uses a .NET Core preview

To deploy an app that uses a preview release of .NET Core, see the following resources. These approaches are also used when the runtime is available but the SDK hasn't been installed on Azure App Service.

- Specify the .NET Core SDK Version using Azure Pipelines
- Deploy a self-contained preview app
- Use Docker with Web Apps for containers
- Install the preview site extension

See the ASP.NET Core on App Service Dashboard for the version of ASP.NET Core available on Azure App service.

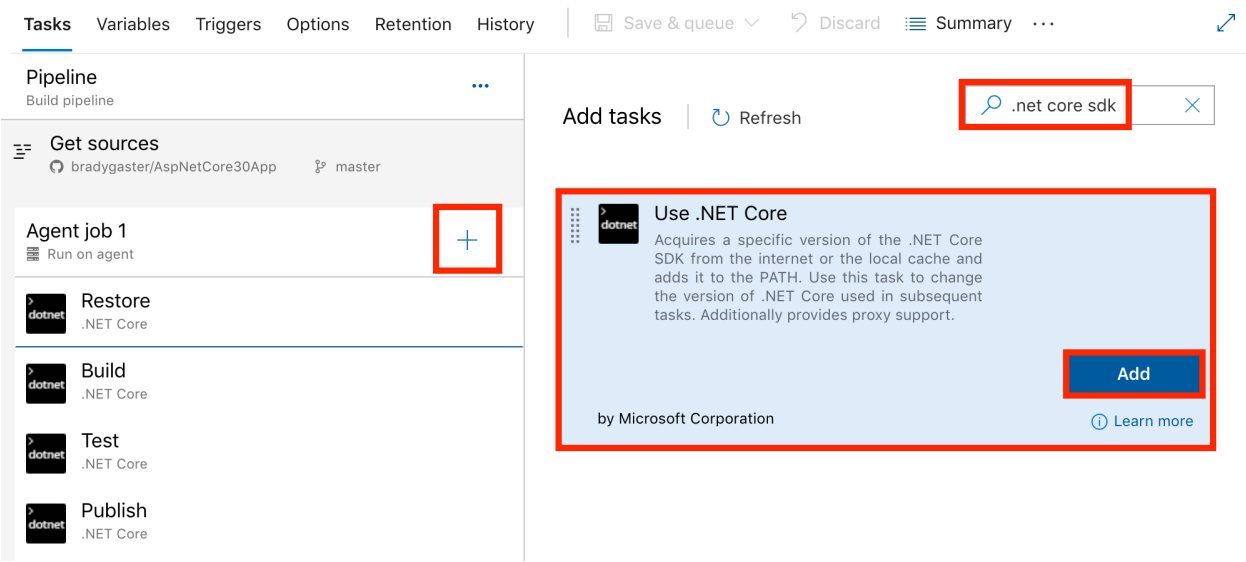
Specify the .NET Core SDK Version using Azure Pipelines

Use Azure App Service CI/CD scenarios to set up a continuous integration build with Azure DevOps. After the Azure DevOps build is created, optionally configure the build to use a specific SDK version.

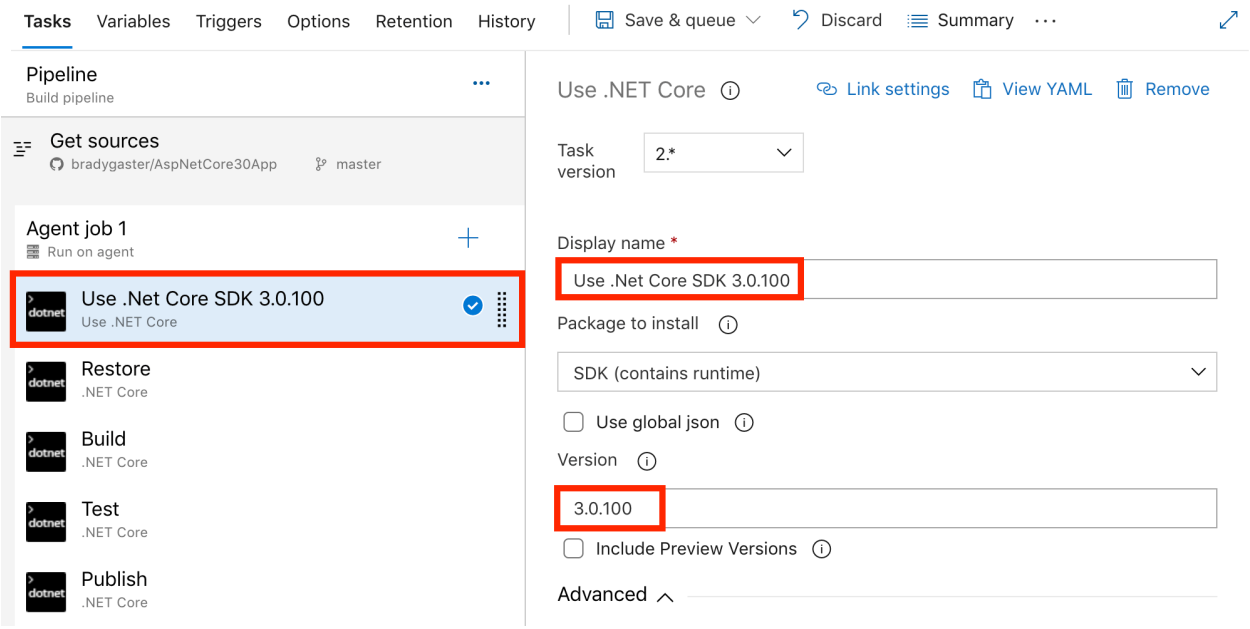
Specify the .NET Core SDK version

When using the App Service deployment center to create an Azure DevOps build, the default build pipeline includes steps for `Restore`, `Build`, `Test`, and `Publish`. To specify the SDK version, select the **Add (+)** button in

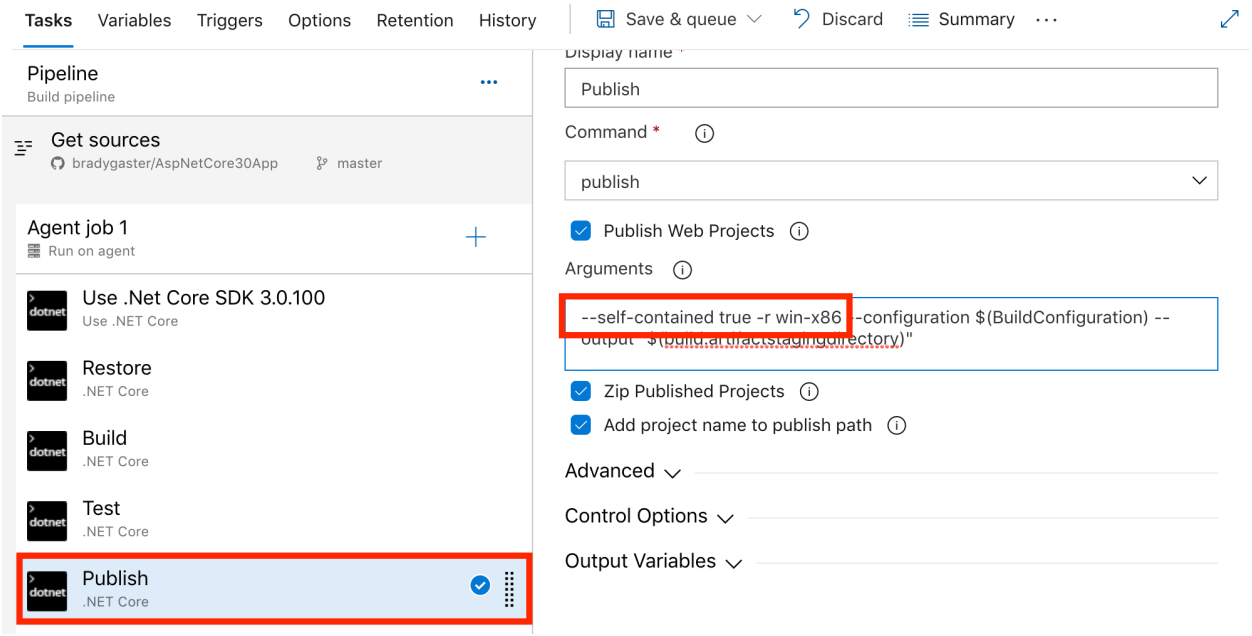
the Agent job list to add a new step. Search for **.NET Core SDK** in the search bar.



Move the step into the first position in the build so that the steps following it use the specified version of the .NET Core SDK. Specify the version of the .NET Core SDK. In this example, the SDK is set to `3.0.100`.



To publish a self-contained deployment (SCD), configure SCD in the `Publish` step and provide the Runtime Identifier (RID).



Deploy a self-contained preview app

A self-contained deployment (SCD) that targets a preview runtime carries the preview runtime in the deployment.

When deploying a self-contained app:

- The site in Azure App Service doesn't require the preview site extension.
- The app must be published following a different approach than when publishing for a framework-dependent deployment (FDD).

Follow the guidance in the Deploy the app self-contained section.

Use Docker with Web Apps for containers

The Docker Hub contains the latest preview Docker images. The images can be used as a base image. Use the image and deploy to Web Apps for Containers normally.

Install the preview site extension

If a problem occurs using the preview site extension, open an dotnet/AspNetCore issue.

1. From the Azure Portal, navigate to the App Service.
2. Select the web app.
3. Type "ex" in the search box to filter for "Extensions" or scroll down the list of management tools.
4. Select **Extensions**.
5. Select **Add**.
6. Select the **ASP.NET Core {X.Y} ({x64|x86}) Runtime** extension from the list, where `{X.Y}` is the ASP.NET Core preview version and `{x64|x86}` specifies the platform.
7. Select **OK** to accept the legal terms.
8. Select **OK** to install the extension.

When the operation completes, the latest .NET Core preview is installed. Verify the installation:

1. Select **Advanced Tools**.
2. Select **Go** in **Advanced Tools**.
3. Select the **Debug console** > **PowerShell** menu item.
4. At the PowerShell prompt, execute the following command. Substitute the ASP.NET Core runtime version for `{X.Y}` and the platform for `{PLATFORM}` in the command:

```
Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.{PLATFORM}\
```

The command returns `True` when the x64 preview runtime is installed.

> **NOTE**
>
> The platform architecture (x86/x64) of an App Services app is set in the app's settings in the Azure Portal for apps that are hosted on an A-series compute (Basic) or higher hosting tier. Confirm that the app's publish settings (for example, in the Visual Studio publish profile (.pubxml)) match the setting in the app's service configuration in the Azure portal.
>
> If the app is run in in-process mode and the platform architecture is configured for 64-bit (x64), the ASP.NET Core Module uses the 64-bit preview runtime, if present. Install the **ASP.NET Core {X.Y} (x64) Runtime** extension using the Azure Portal.
>
> After installing the x64 preview runtime, run the following command in the Azure Kudu PowerShell command window to verify the installation. Substitute the ASP.NET Core runtime version for `{X.Y}` in the following command:
>
> ```
> Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64\
> ```
>
> The command returns `True` when the x64 preview runtime is installed.

> **NOTE**
>
> **ASP.NET Core Extensions** enables additional functionality for ASP.NET Core on Azure App Services, such as enabling Azure logging. The extension is installed automatically when deploying from Visual Studio. If the extension isn't installed, install it for the app.

Use the preview site extension with an ARM template

If an ARM template is used to create and deploy apps, the `siteextensions` resource type can be used to add the site extension to a web app. For example:

```
{
    "type": "siteextensions",
    "name": "AspNetCoreRuntime",
    "apiVersion": "2015-04-01",
    "location": "[resourceGroup().location]",
    "properties": {
        "version": "[parameters('aspnetcoreVersion')]"
    },
    "dependsOn": [
        "[resourceId('Microsoft.Web/Sites', parameters('siteName'))]"
    ]
}
```

Publish and deploy the app

For a 64-bit deployment:

- Use a 64-bit .NET Core SDK to build a 64-bit app.
- Set the **Platform** to **64 Bit** in the App Service's **Configuration** > **General settings**. The app must use a Basic or higher service plan to enable the choice of platform bitness.

Deploy the app framework-dependent

- Visual Studio

- .NET Core CLI

1. Select **Build** > **Publish {Application Name}** from the Visual Studio toolbar or right-click the project in **Solution Explorer** and select **Publish**.
2. In the **Pick a publish target** dialog, confirm that **App Service** is selected.
3. Select **Advanced**. The **Publish** dialog opens.
4. In the **Publish** dialog:
   - Confirm that the **Release** configuration is selected.
   - Open the **Deployment Mode** drop-down list and select **Framework-Dependent**.
   - Select **Portable** as the **Target Runtime**.
   - If you need to remove additional files upon deployment, open **File Publish Options** and select the check box to remove additional files at the destination.
   - Select **Save**.
5. Create a new site or update an existing site by following the remaining prompts of the publish wizard.

Deploy the app self-contained

Use Visual Studio or the .NET Core CLI for a self-contained deployment (SCD).

- Visual Studio
- .NET Core CLI

1. Select **Build** > **Publish {Application Name}** from the Visual Studio toolbar or right-click the project in **Solution Explorer** and select **Publish**.
2. In the **Pick a publish target** dialog, confirm that **App Service** is selected.
3. Select **Advanced**. The **Publish** dialog opens.
4. In the **Publish** dialog:
   - Confirm that the **Release** configuration is selected.
   - Open the **Deployment Mode** drop-down list and select **Self-Contained**.
   - Select the target runtime from the **Target Runtime** drop-down list. The default is `win-x86`.
   - If you need to remove additional files upon deployment, open **File Publish Options** and select the check box to remove additional files at the destination.
   - Select **Save**.
5. Create a new site or update an existing site by following the remaining prompts of the publish wizard.

Protocol settings (HTTPS)

Secure protocol bindings allow you specify a certificate to use when responding to requests over HTTPS. Binding requires a valid private certificate (*.pfx*) issued for the specific hostname. For more information, see Tutorial: Bind an existing custom SSL certificate to Azure App Service.

Transform web.config

If you need to transform *web.config* on publish (for example, set environment variables based on the configuration, profile, or environment), see Transform web.config.

Additional resources

- App Service overview
- Azure App Service: The Best Place to Host your .NET Apps (55-minute overview video)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)
- Azure App Service diagnostics overview

- Host ASP.NET Core in a web farm

Azure App Service on Windows Server uses Internet Information Services (IIS). The following topics pertain to the underlying IIS technology:

- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module
- IIS modules with ASP.NET Core
- Windows Server - IT administrator content for current and previous releases

Publish an ASP.NET Core app to Azure with Visual Studio

9/22/2020 • 4 minutes to read • Edit Online

By Rick Anderson

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

See Publish a Web app to Azure App Service using Visual Studio for Mac if you are working on macOS.

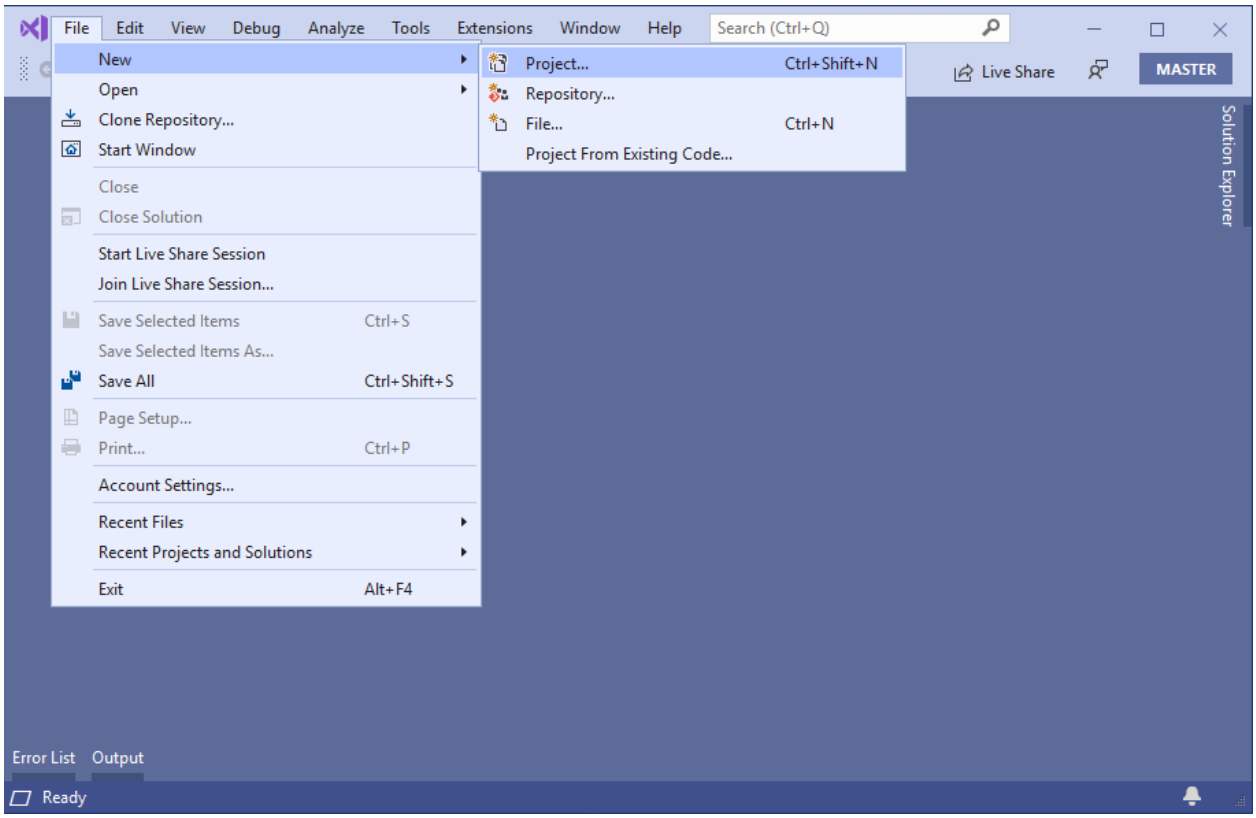
To troubleshoot an App Service deployment issue, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Set up

- Open a free Azure account if you don't have one.

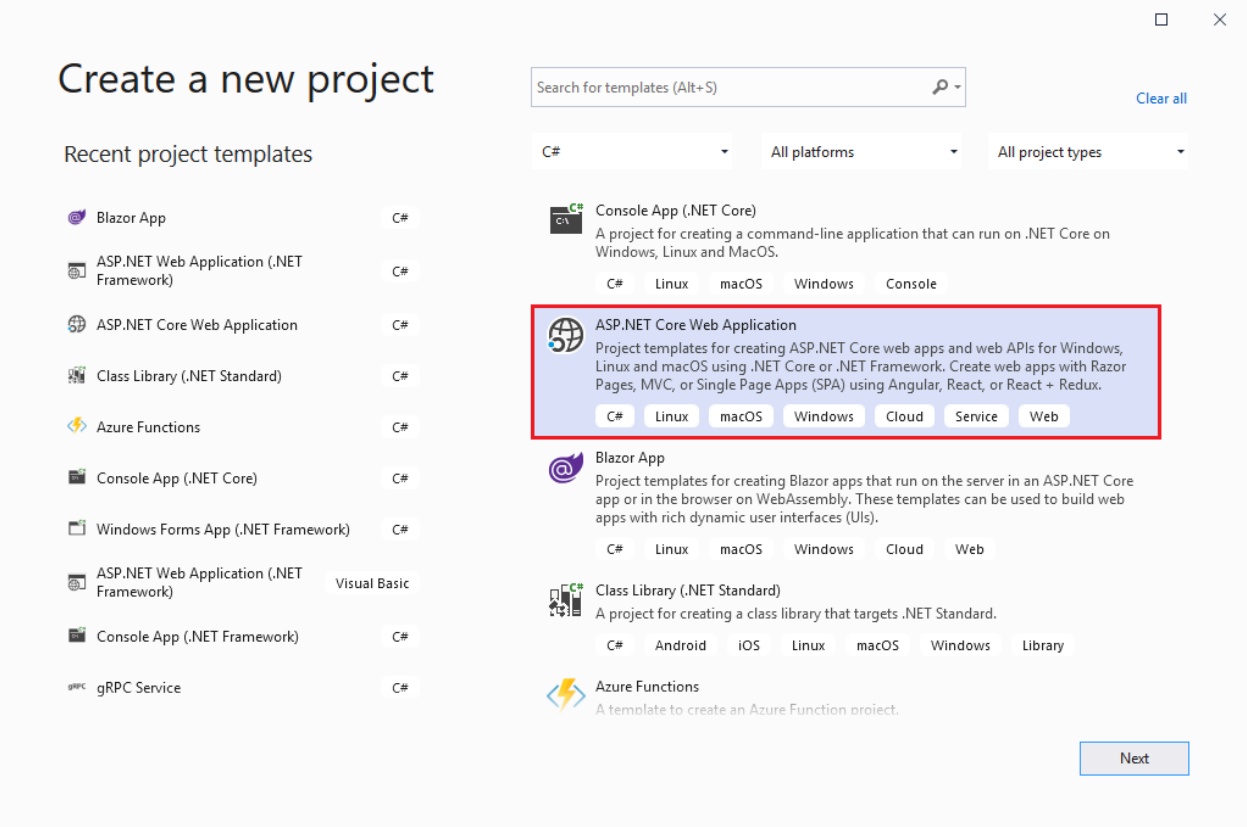
Create a web app

In the Visual Studio Start Page, select **File** > **New** > **Project...**



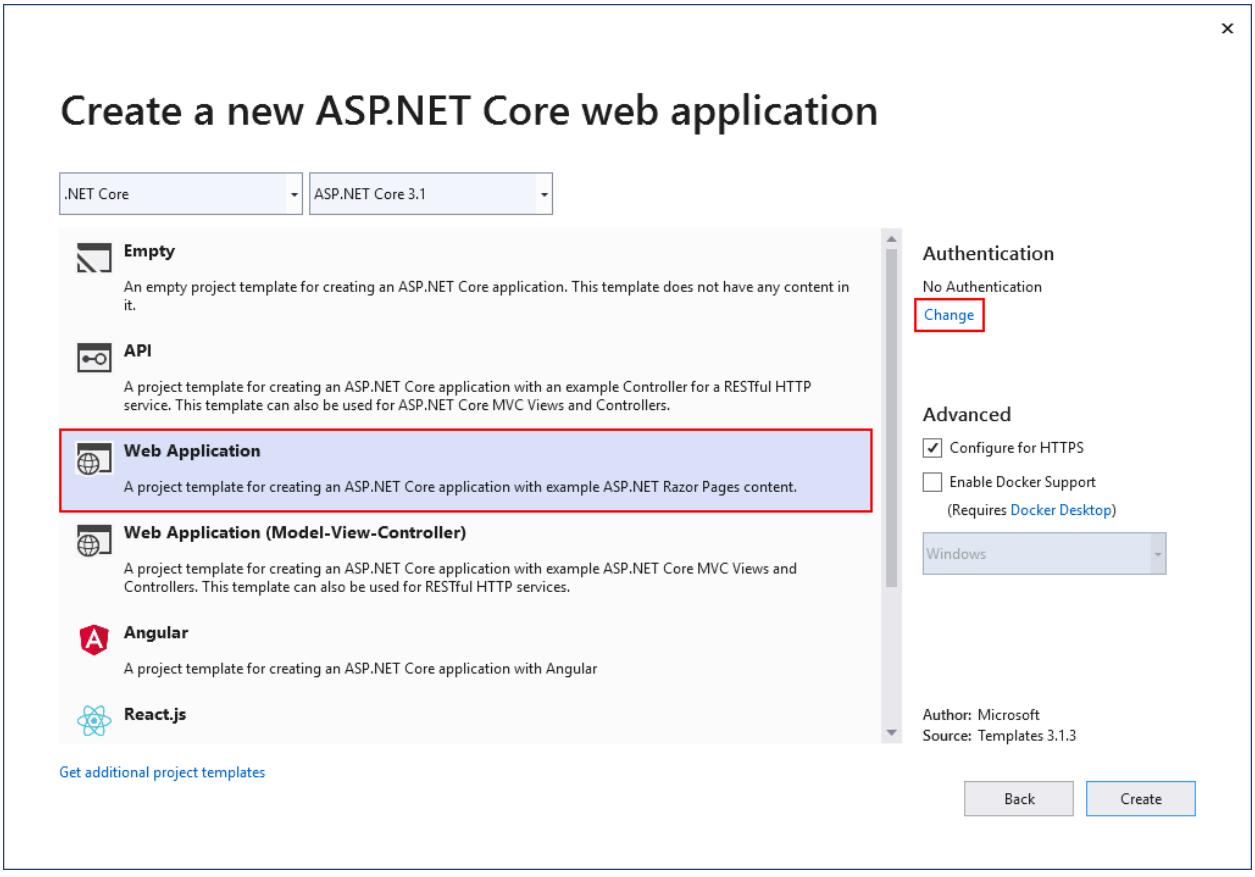
Complete the **New Project** dialog:

- Select **ASP.NET Core Web Application**.
- Select **Next**.



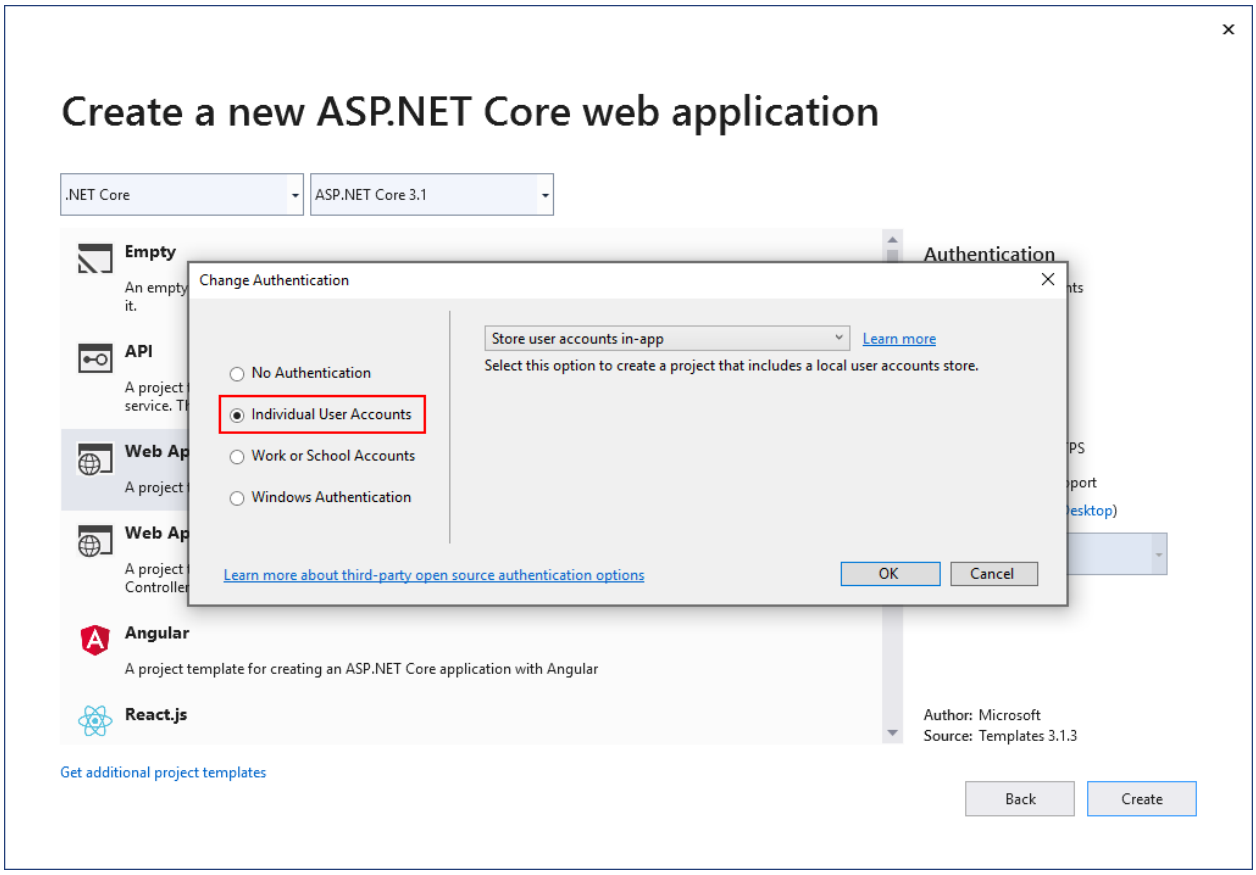
In the **New ASP.NET Core Web Application** dialog:

- Select **Web Application**.
- Select **Change** under Authentication.



The **Change Authentication** dialog appears.

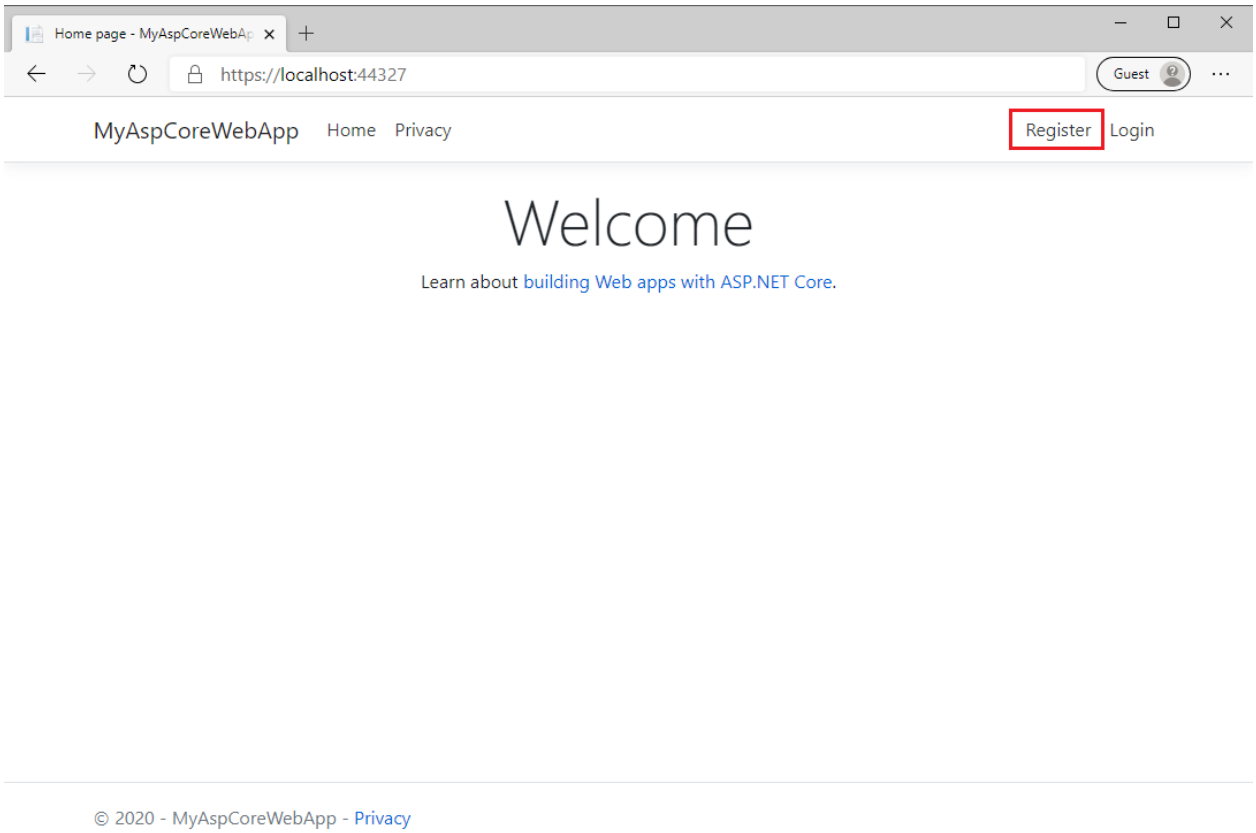
- Select **Individual User Accounts**.
- Select **OK** to return to the **New ASP.NET Core Web Application**, then select **Create**.



Visual Studio creates the solution.

Run the app

- Press CTRL+F5 to run the project.
- Test the **Privacy** link.

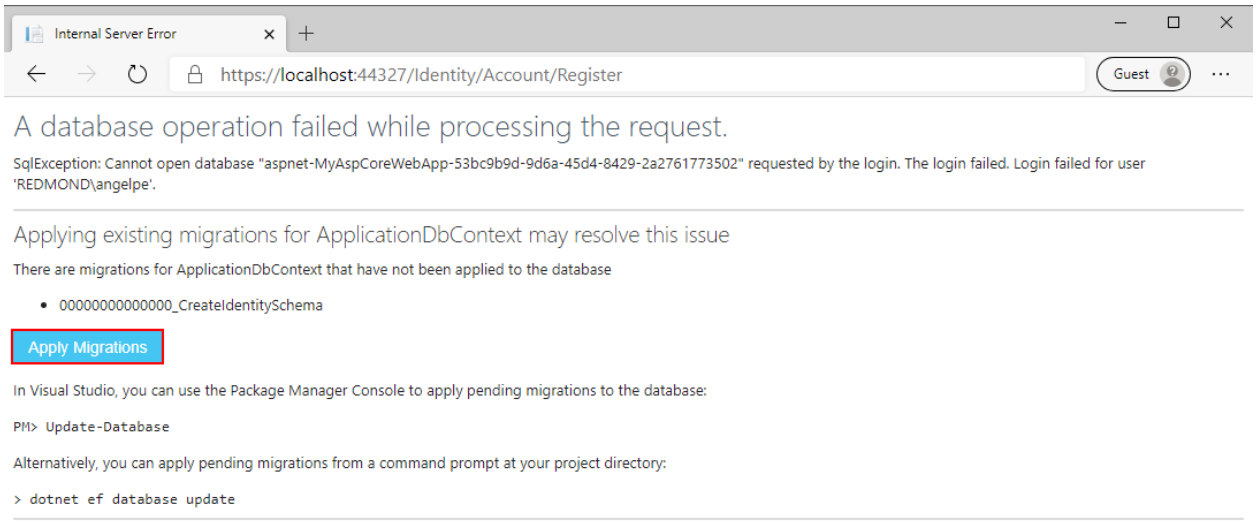


Register a user

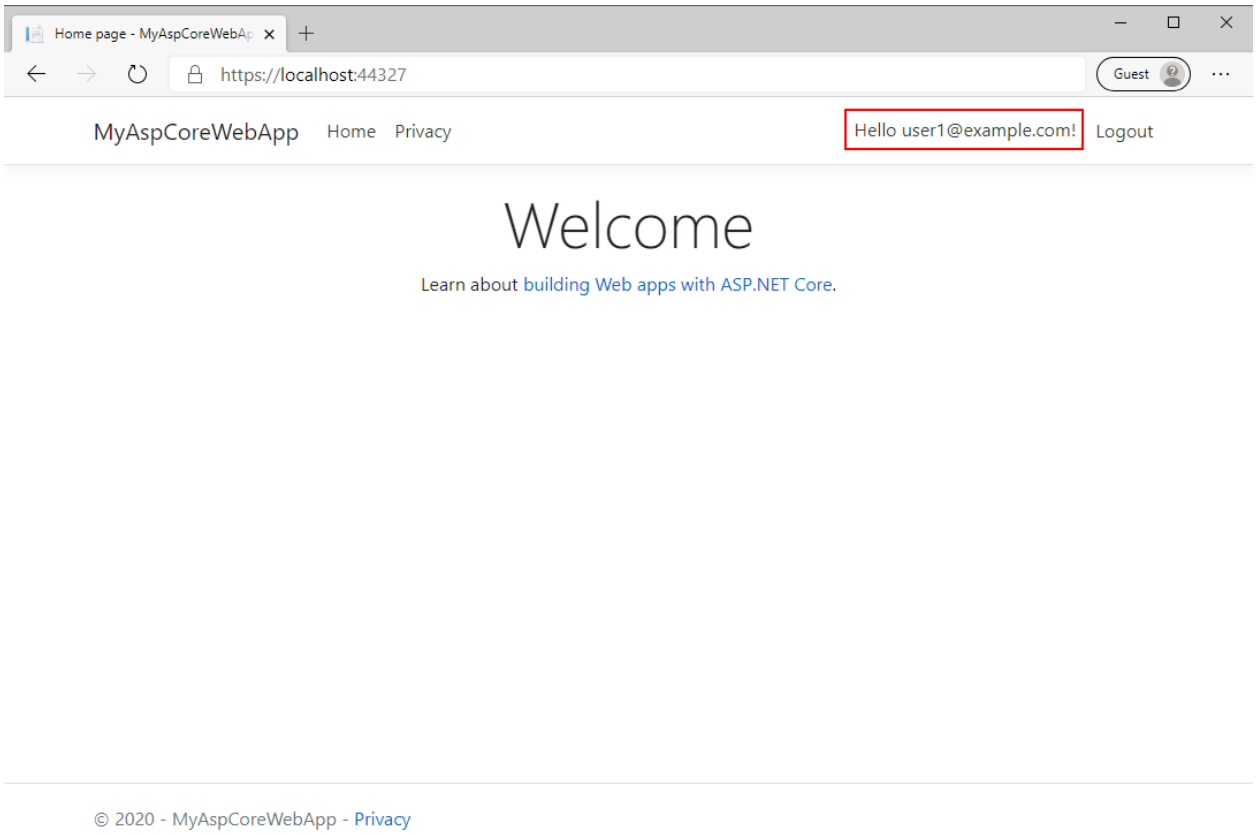
- Select **Register** and register a new user. You can use a fictitious email address. When you submit, the page displays the following error:

  "A database operation failed while processing the request. Applying existing migrations for Application DB context may resolve this issue."

- Select **Apply Migrations** and, once the page updates, refresh the page.

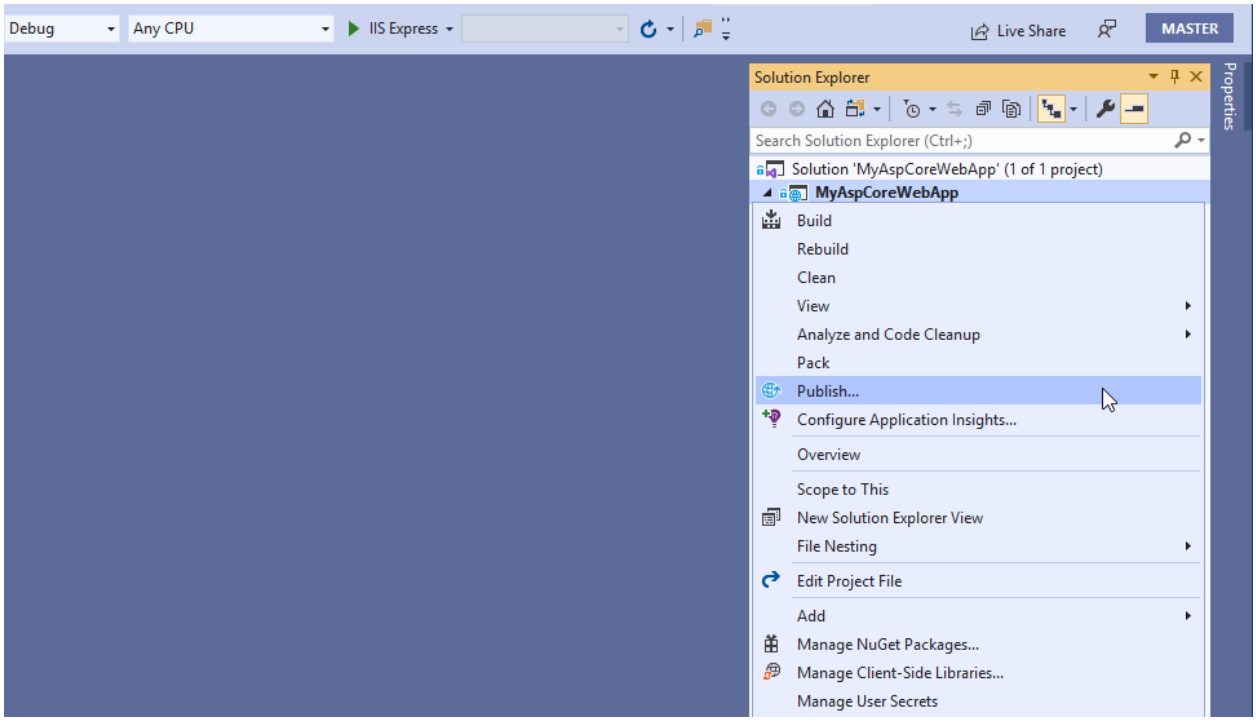


The app displays the email used to register the new user and a **Logout** link.



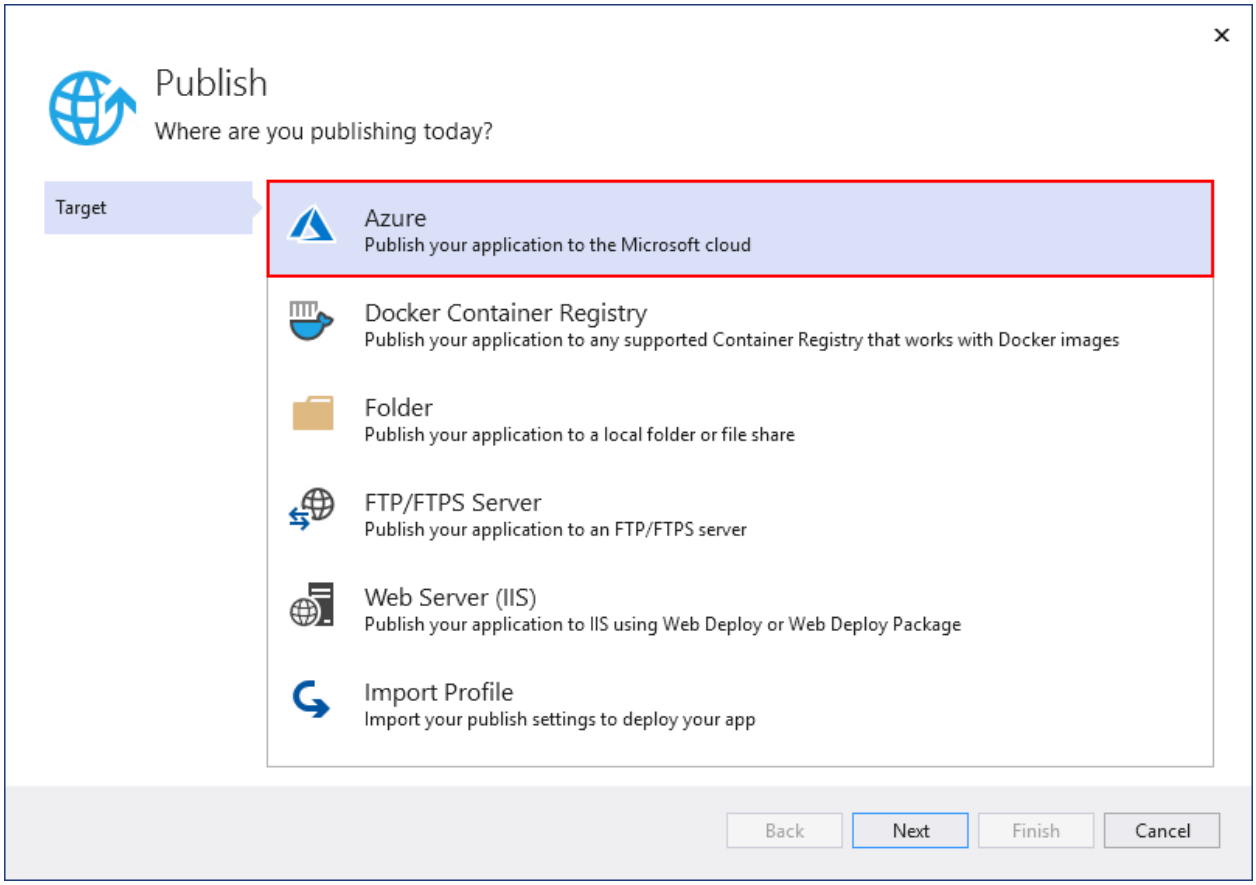
Deploy the app to Azure

Right-click on the project in Solution Explorer and select **Publish...**.



In the **Publish** dialog:

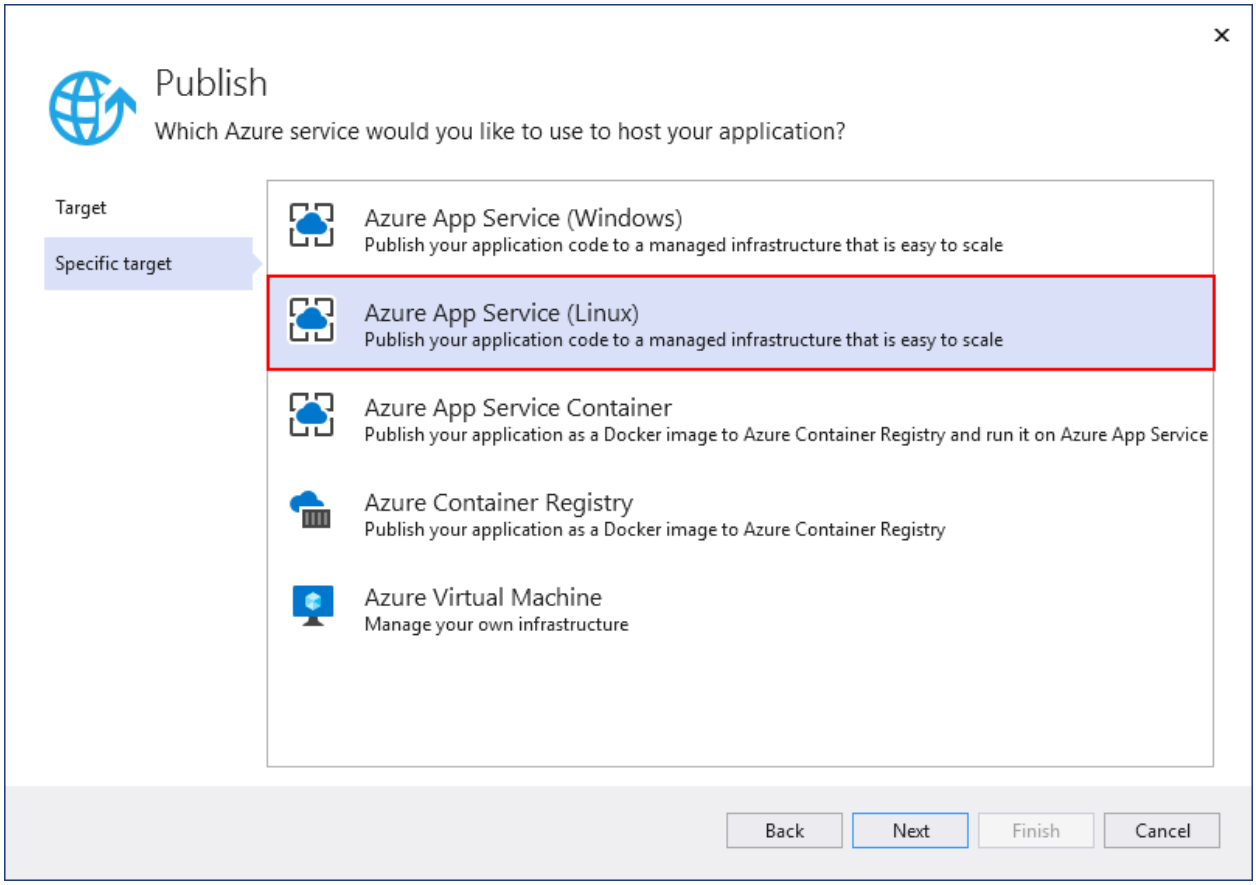
- Select **Azure**.
- Select **Next**.



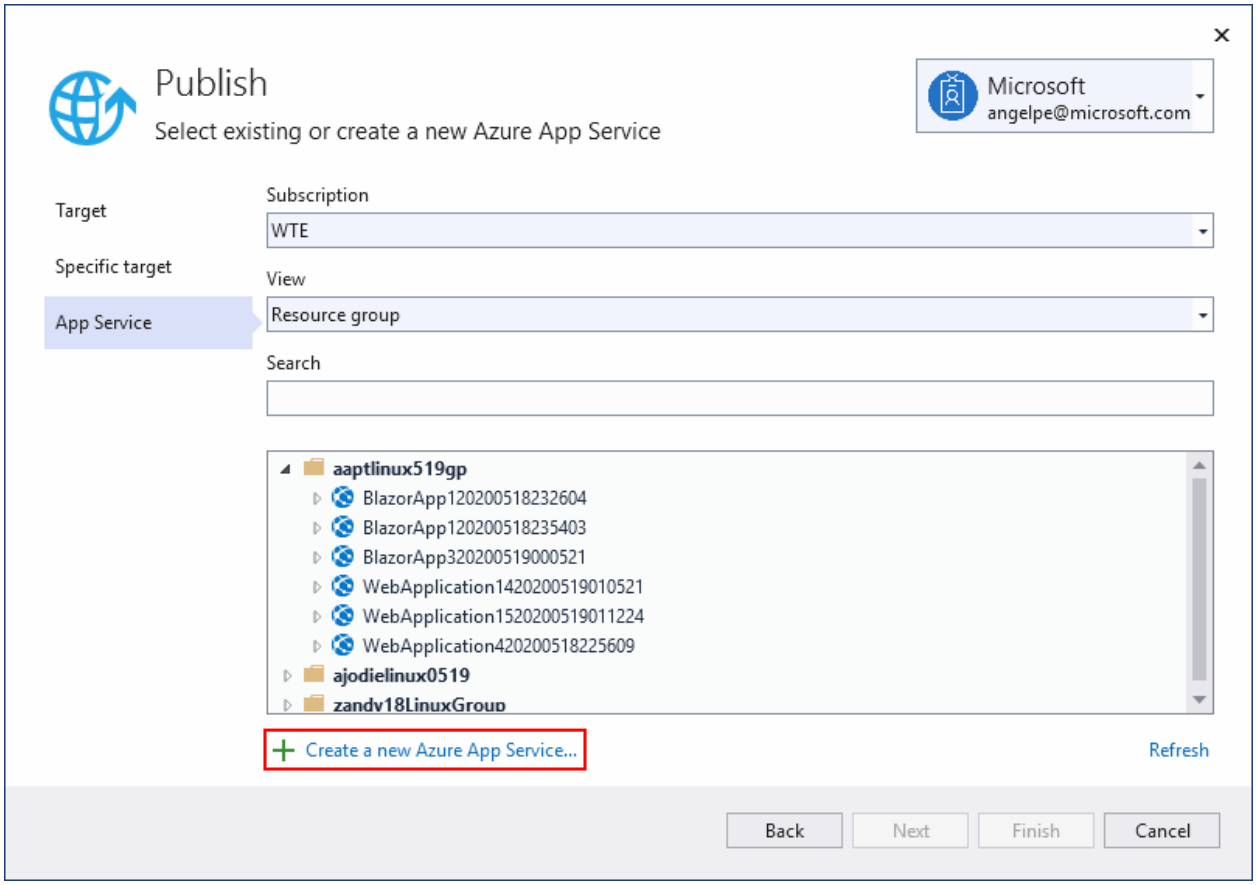
In the **Publish** dialog:

- Select **Azure App Service (Linux)**.

- Select **Next**.



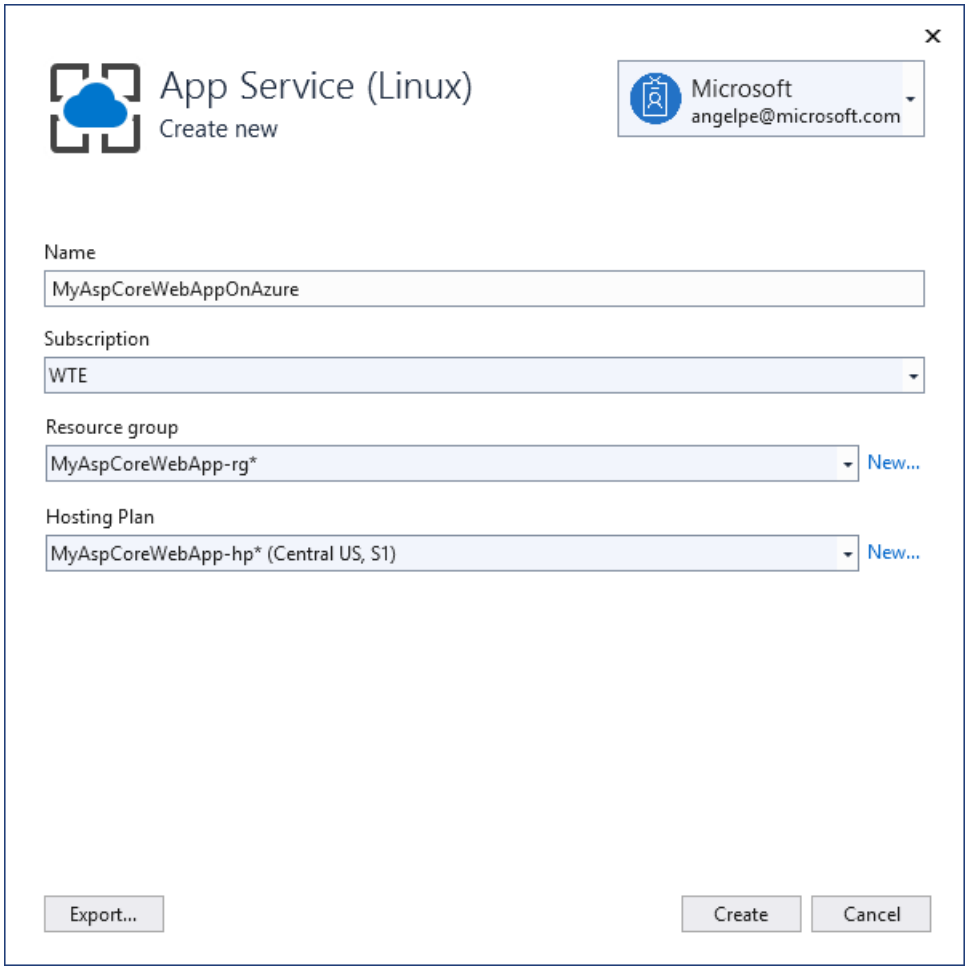
In the **Publish** dialog select **Create a new Azure App Service...**



The **Create App Service** dialog appears:

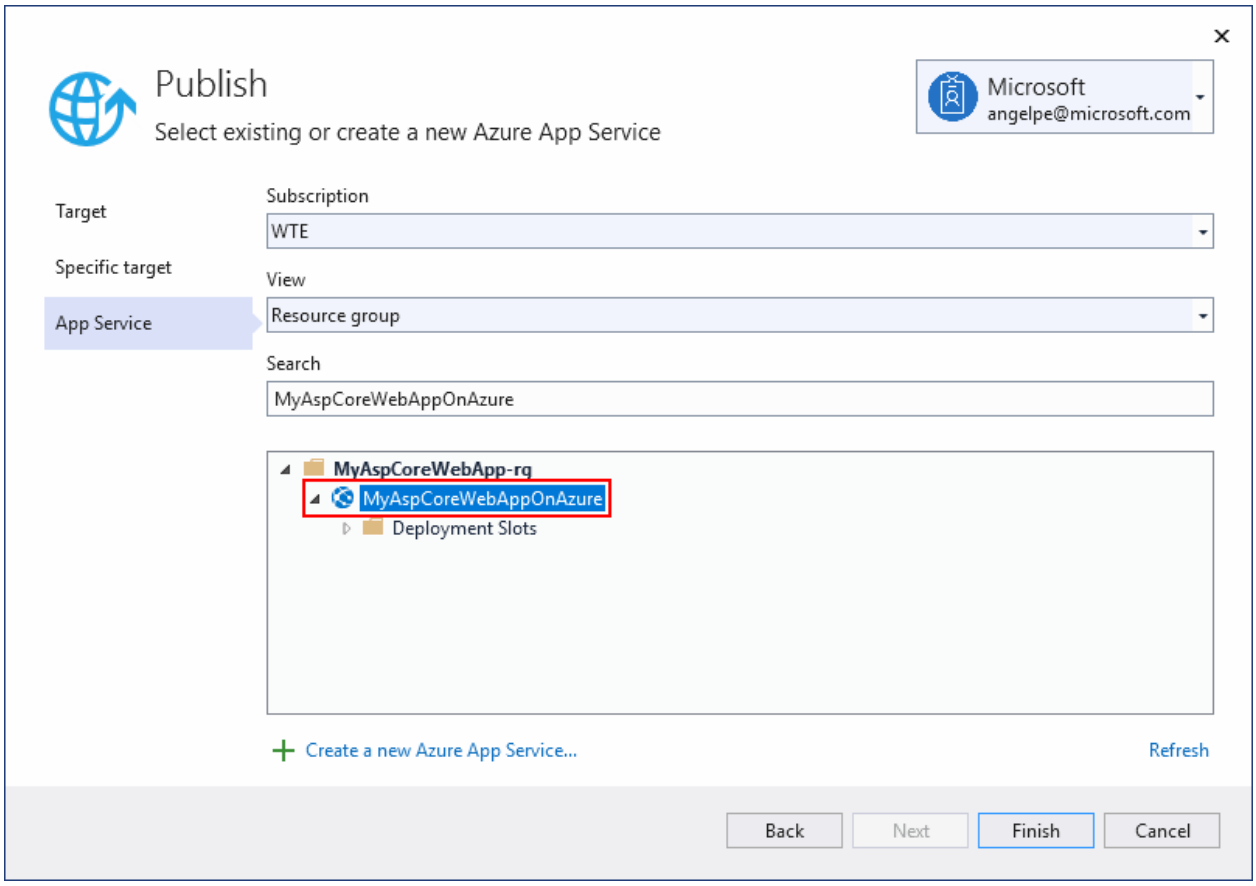
- The **App Name**, **Resource Group**, and **App Service Plan** entry fields are populated. You can keep these names or change them.

- Select **Create**.

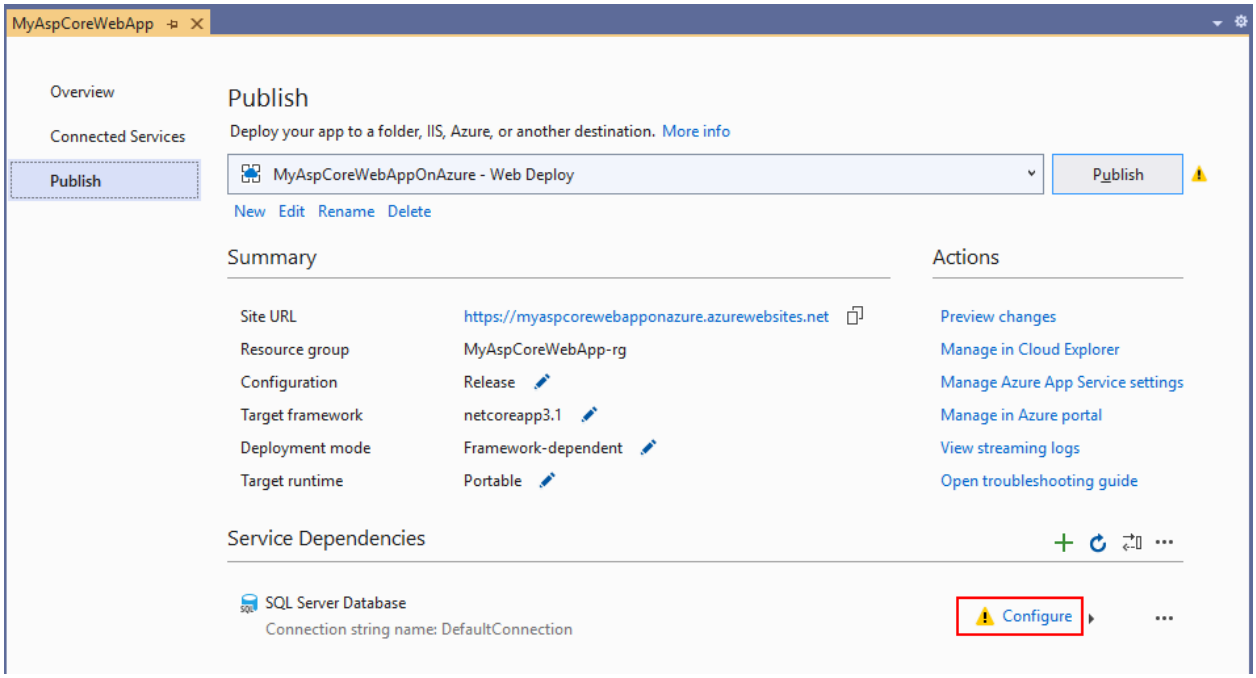


After creation is completed the dialog is automatically closed and the **Publish** dialog gets focus again:

- The new instance that was just created is automatically selected.
- Select **Finish**.

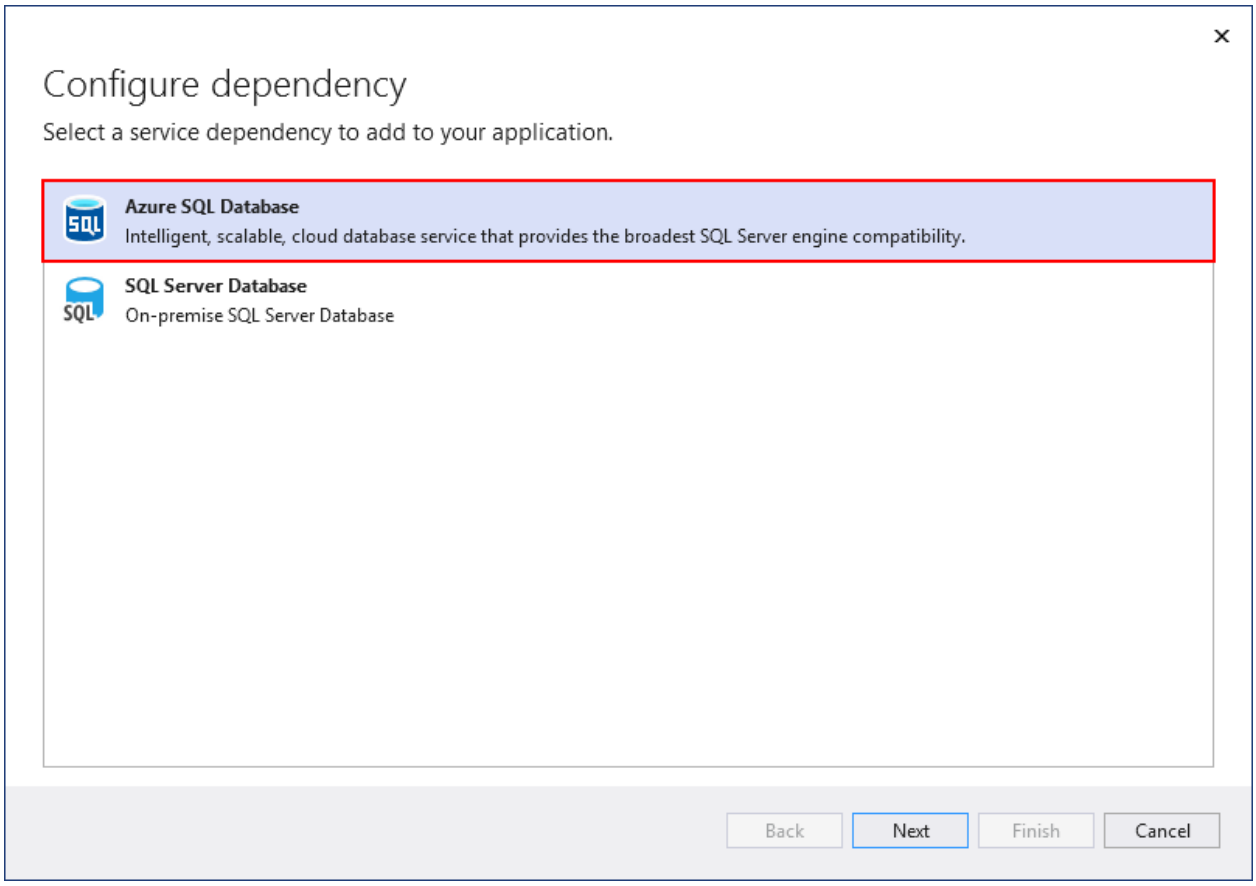


Next you see the **Publish Profile summary** page. Visual Studio has detected that this application requires a SQL Server database and it's asking you to configure it. Select **Configure**.

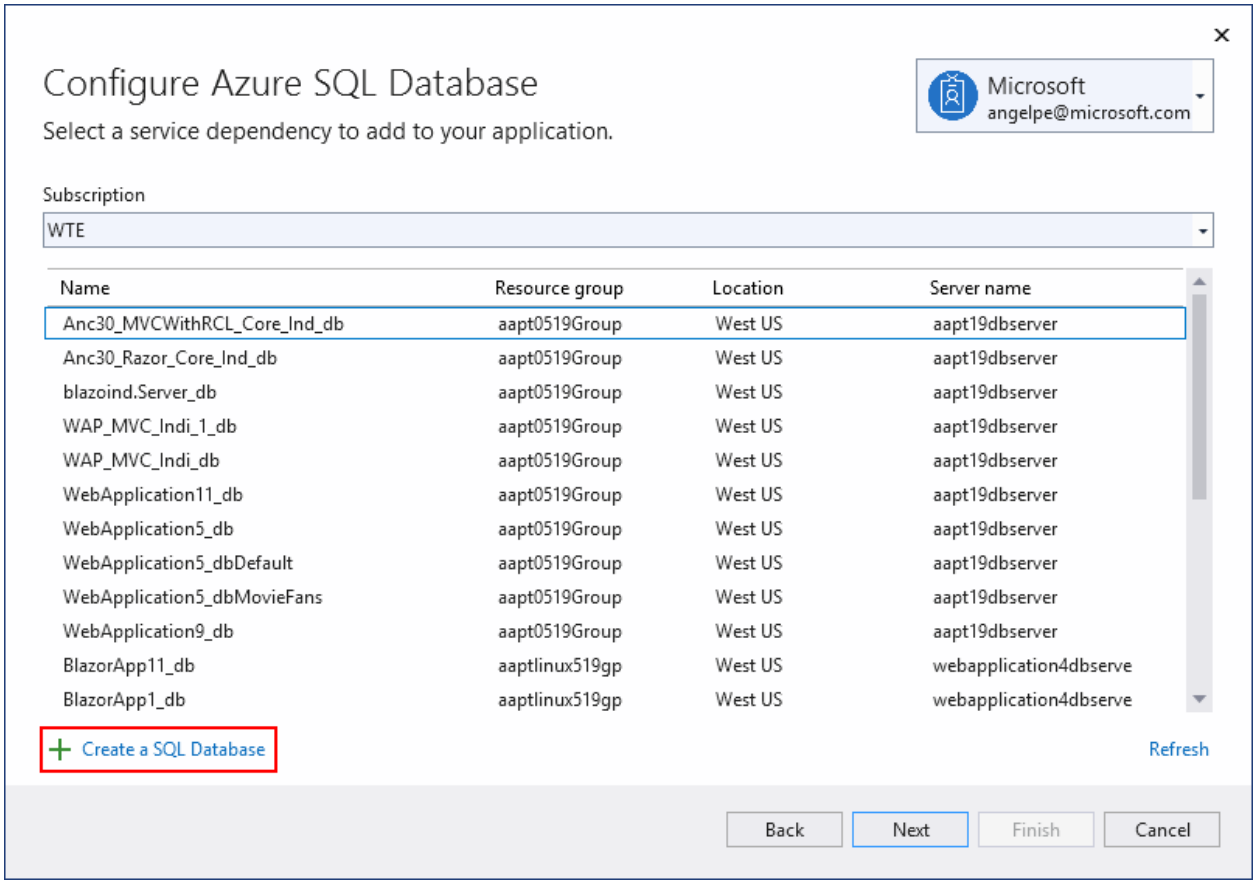


The **Configure dependency** dialog appears:

- Select **Azure SQL Database**.
- Select **Next**.



In the **Configure Azure SQL database** dialog select **Create a SQL Database**

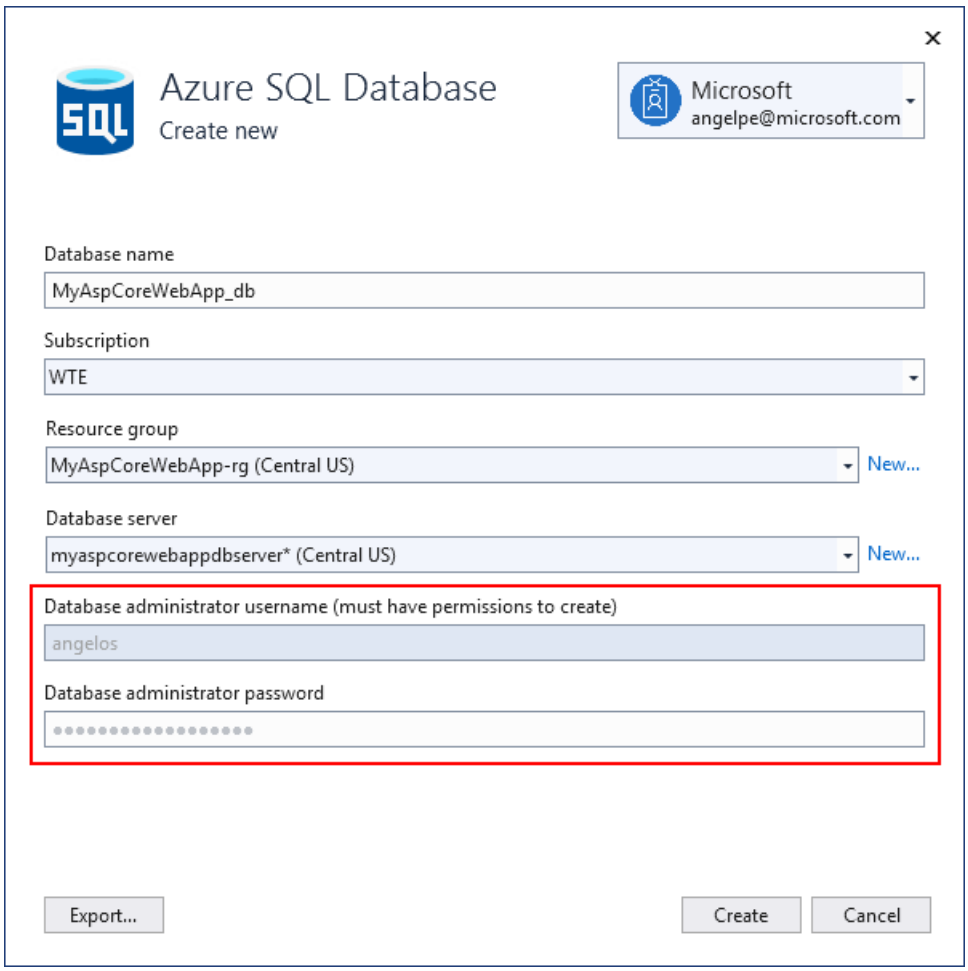


The **Create Azure SQL Database** appears:

- The **Database name**, **Resource Group**, **Database server** and **App Service Plan** entry fields are populated. You can keep these values or change them.
- Enter the **Database administrator username** and **Database administrator password** for the selected

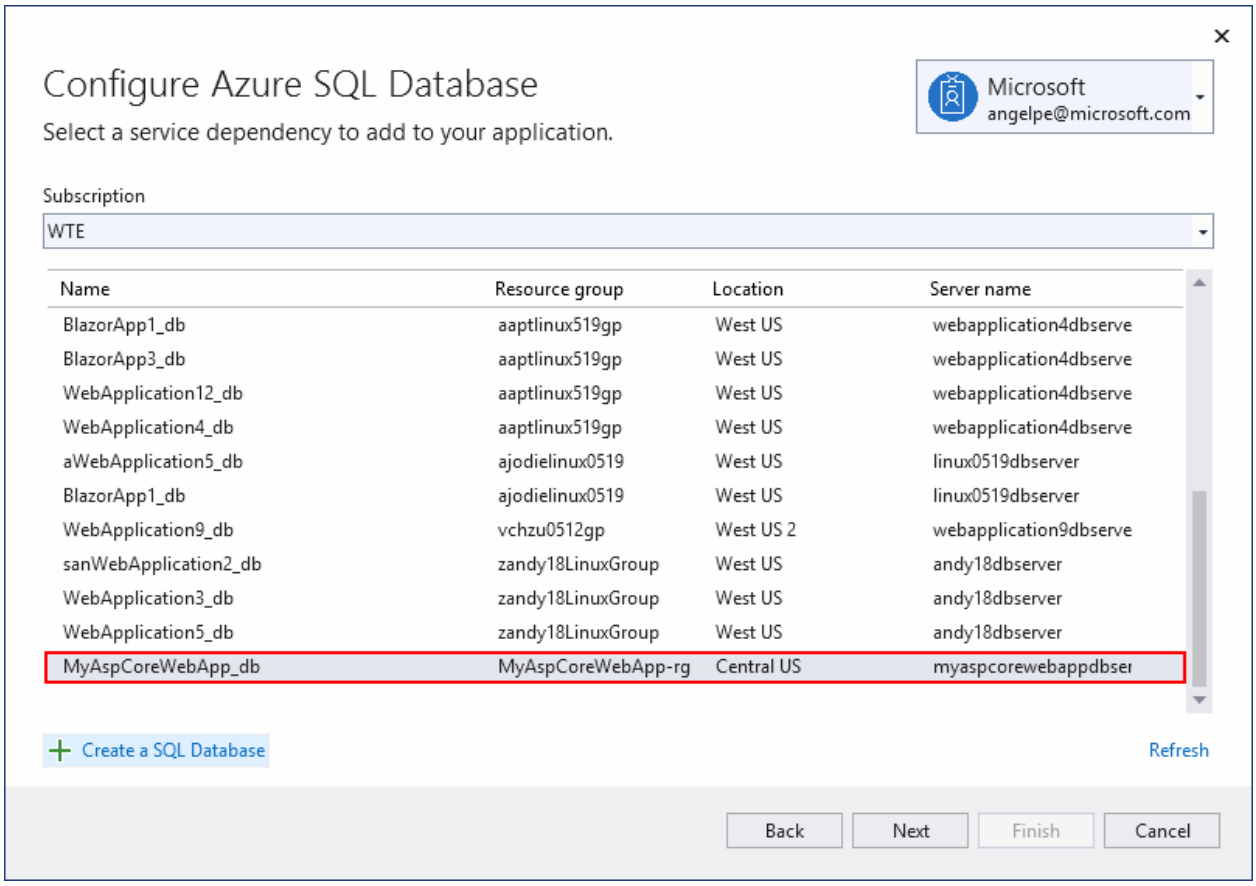
Database server (note the account you use must have the necessary permissions to create the new Azure SQL database)

- Select **Create**.



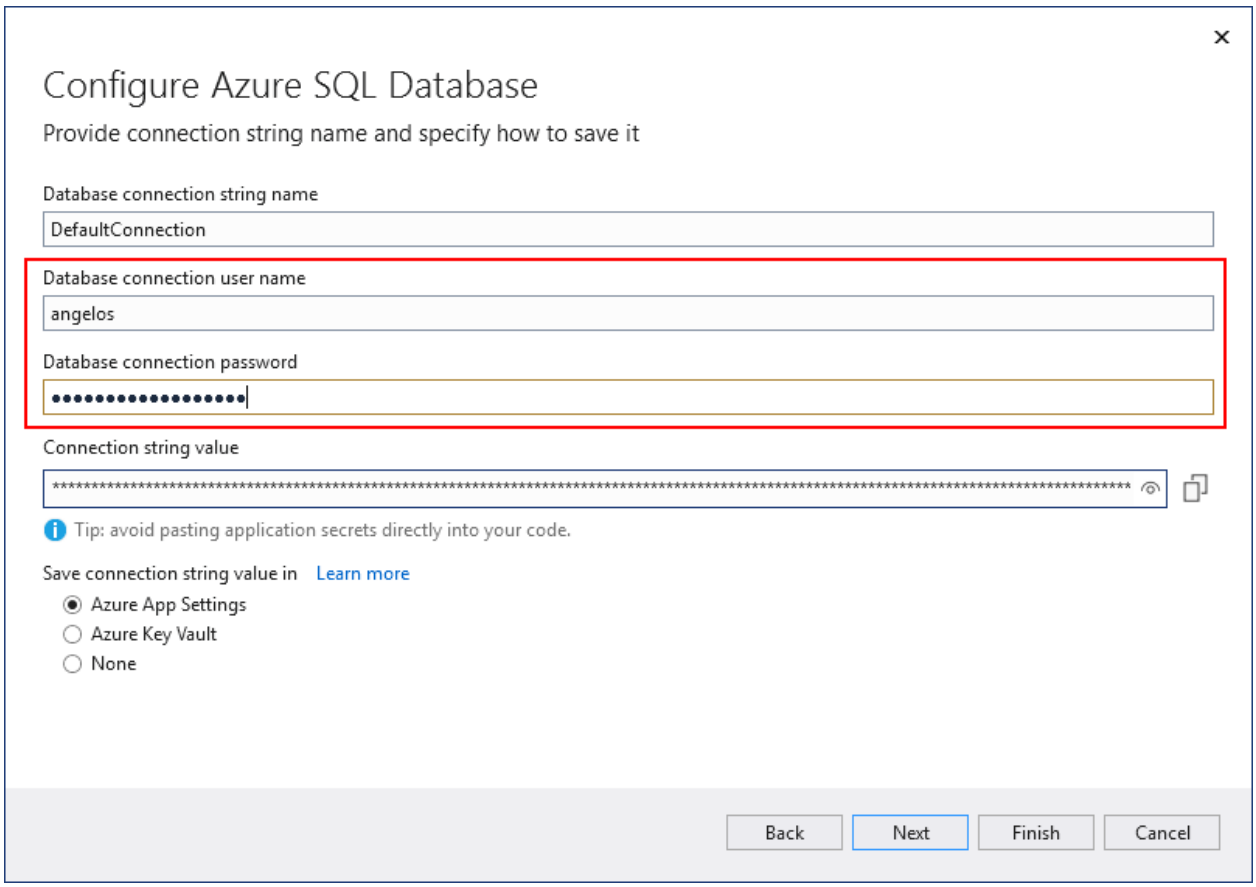
After creation is completed the dialog is automatically closed and the **Configure Azure SQL Database** dialog gets focus again:

- The new instance that was just created is automatically selected.
- Select **Next**.

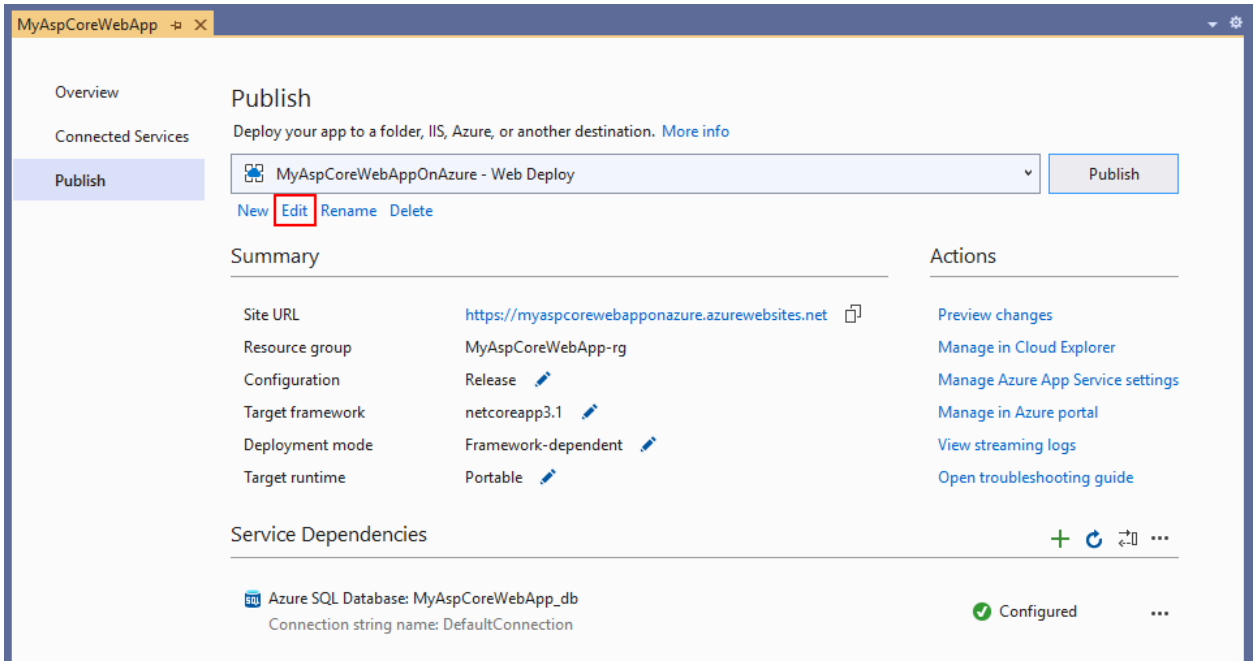


In the next step of the **Configure Azure SQL Database** dialog:

- Enter the **Database connection user name** and **Database connection password** fields. These are the details your application will use to connect to the database at runtime. Best practice is to avoid using the same details as the admin username & password used in the previous step.
- Select **Finish**.

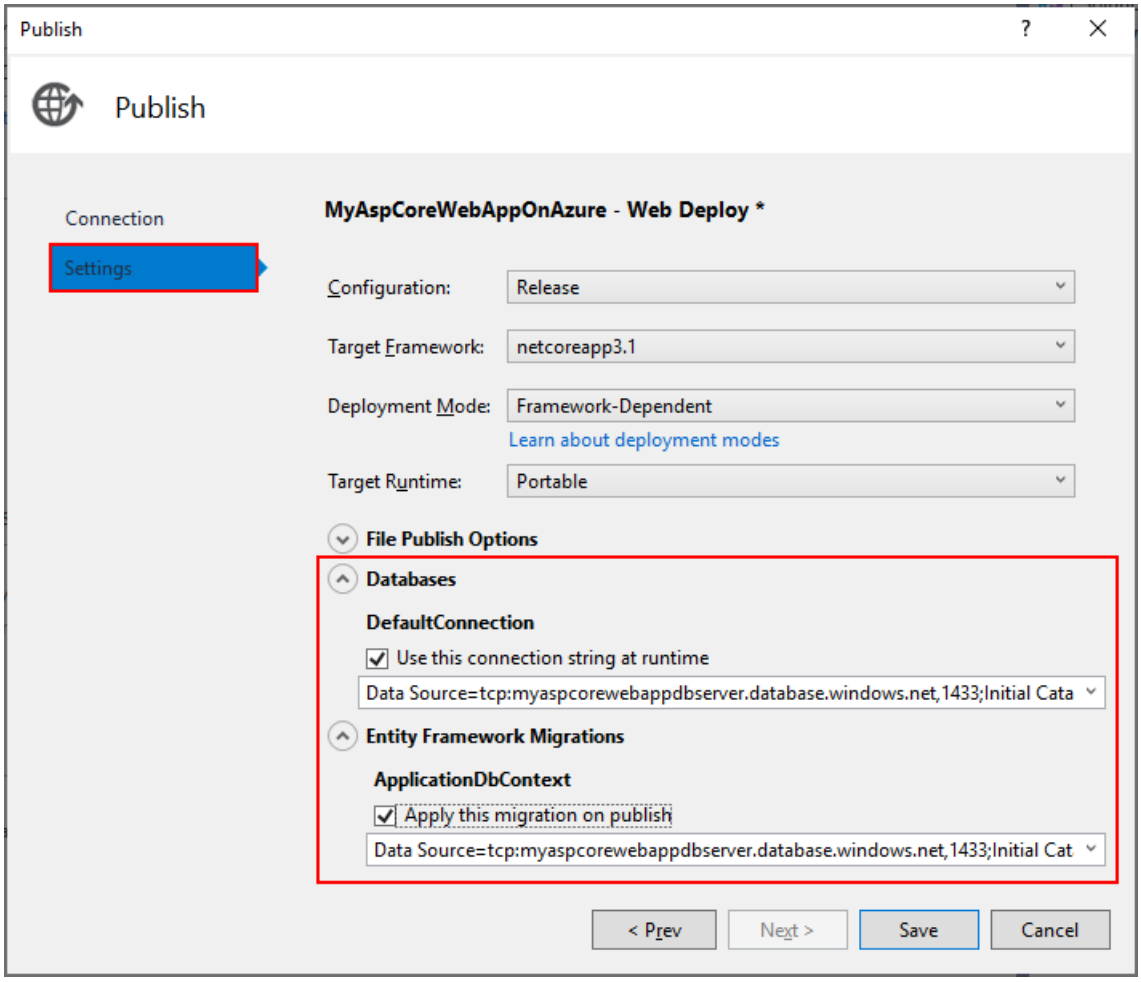


In the **Publish Profile summary** page select **Settings**:

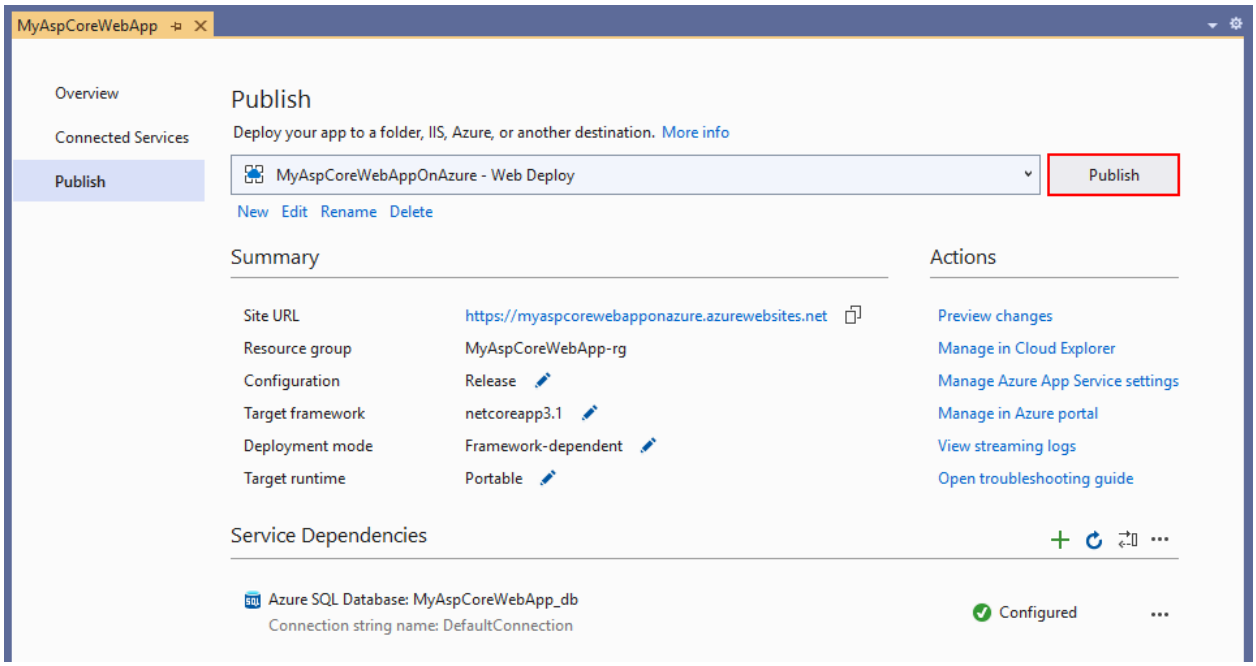


On the **Settings** page of the **Publish** dialog:

- Expand **Databases** and check **Use this connection string at runtime**.
- Expand **Entity Framework Migrations** and check **Apply this migration on publish**.
- Select **Save**. Visual Studio returns to the **Publish** dialog.



Click **Publish**. Visual Studio publishes your app to Azure. When the deployment completes, the app is opened in a browser.



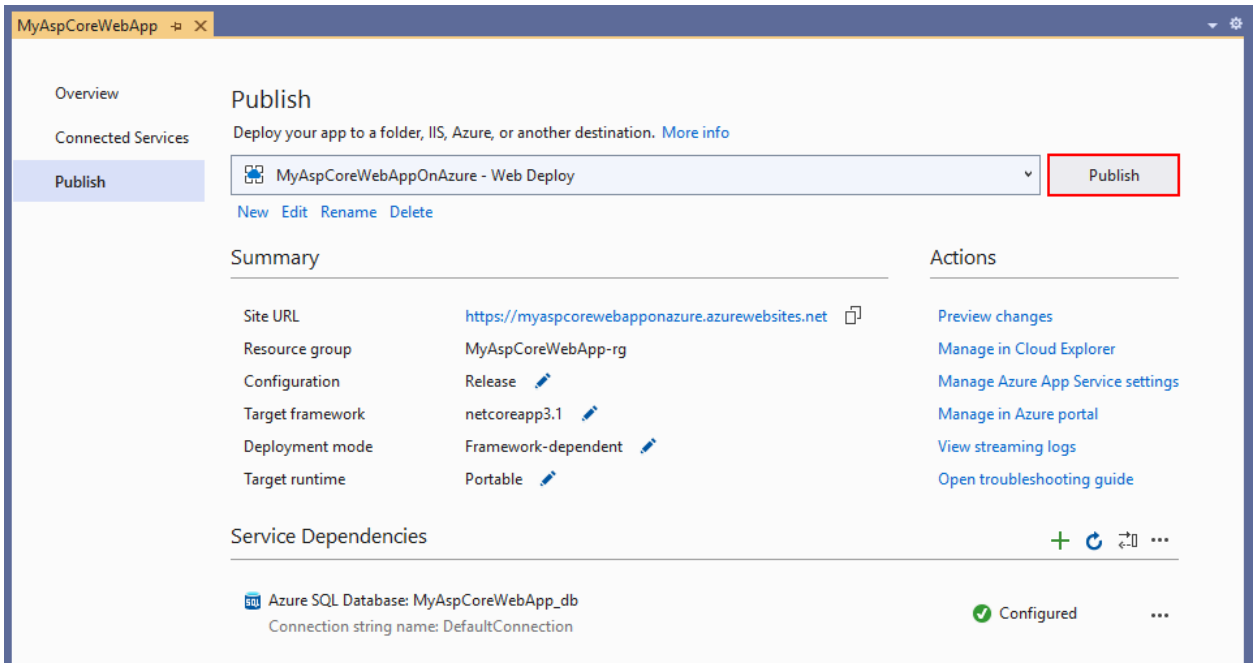
Update the app

- Edit the *Pages/Index.cshtml* Razor page and change its contents. For example, you can modify the paragraph to say "Hello ASP.NET Core!":

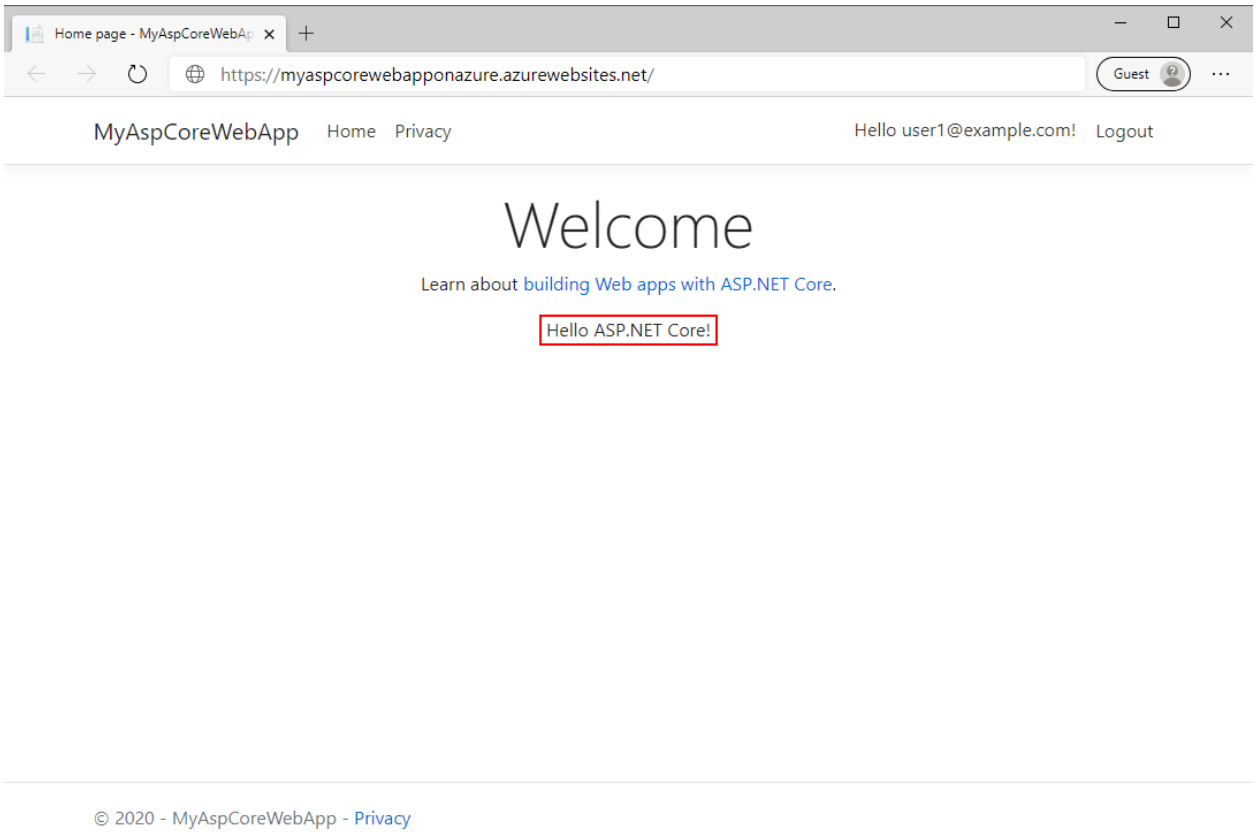
```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building Web apps with ASP.NET
Core</a>.</p>
    <p>Hello ASP.NET Core!</p>
</div>
```

- Select **Publish** from the **Publish Profile summary** page again.



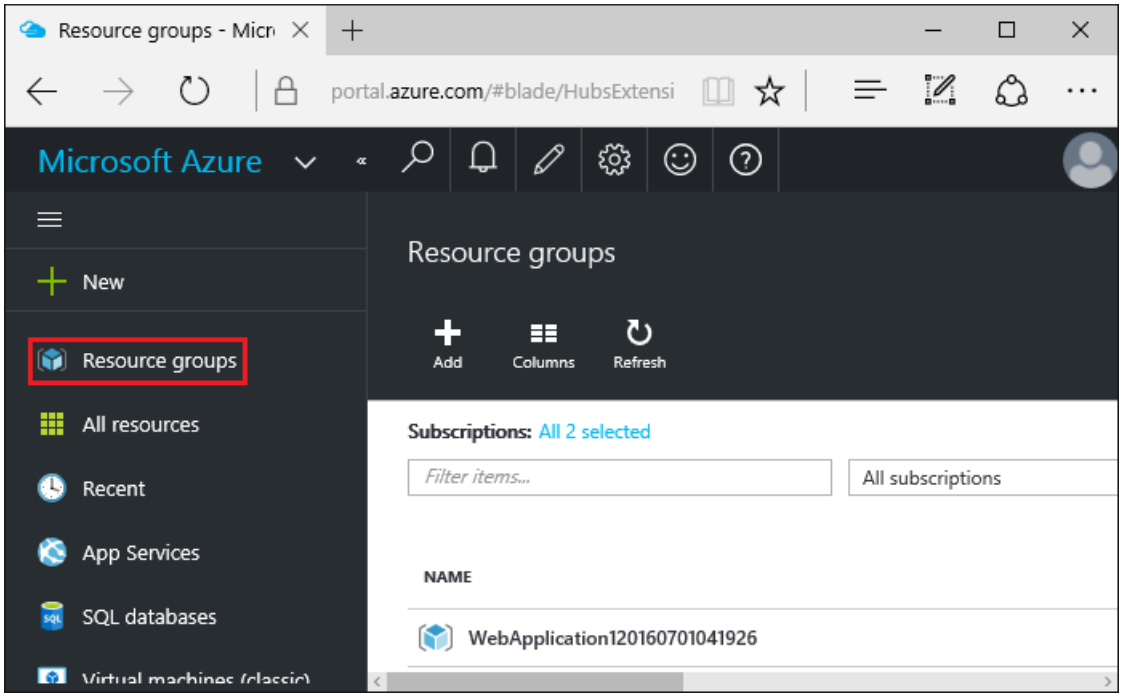
- After the app is published, verify the changes you made are available on Azure.



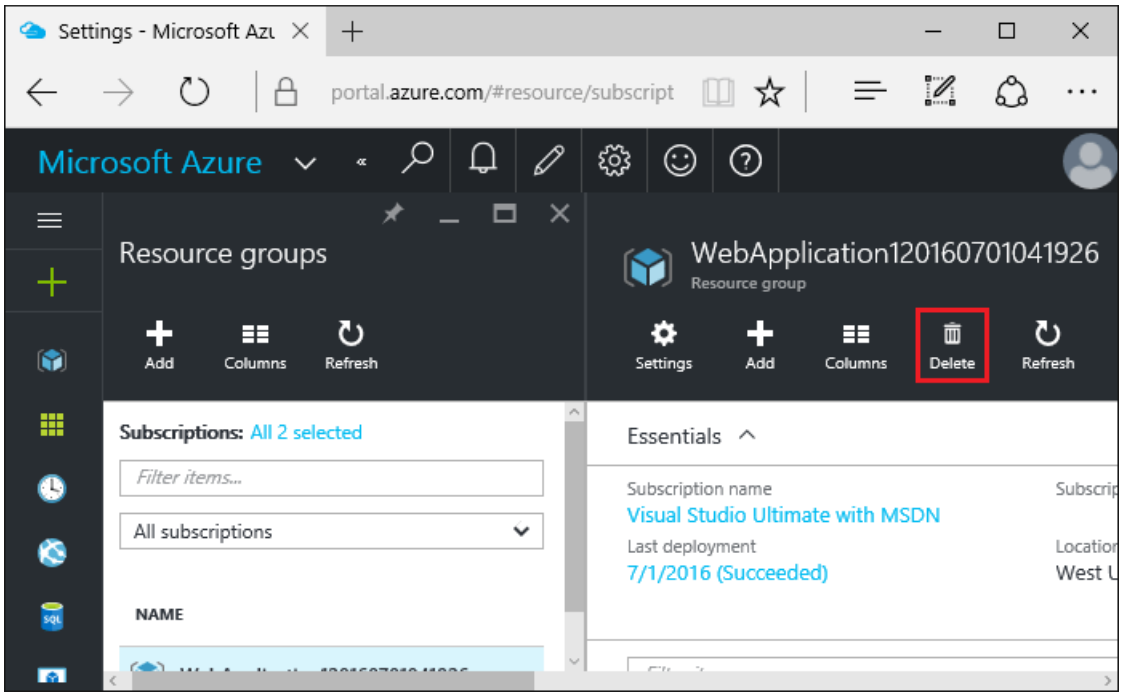
Clean up

When you have finished testing the app, go to the Azure portal and delete the app.

- Select **Resource groups**, then select the resource group you created.



- In the **Resource groups** page, select **Delete**.



- Enter the name of the resource group and select **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure.

Next steps

- Continuous deployment to Azure with Visual Studio and Git with ASP.NET Core

Additional resources

- For Visual Studio Code, see Publish profiles.
- Azure App Service
- Azure resource groups
- Azure SQL Database
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment
- Troubleshoot ASP.NET Core on Azure App Service and IIS

Continuous deployment to Azure with Visual Studio and Git with ASP.NET Core

9/22/2020 • 6 minutes to read • Edit Online

By Erik Reitan

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

This tutorial shows how to create an ASP.NET Core web app using Visual Studio and deploy it from Visual Studio to Azure App Service using continuous deployment.

See also Create your first pipeline with Azure Pipelines, which shows how to configure a continuous delivery (CD) workflow for Azure App Service using Azure DevOps Services. Azure Pipelines (an Azure DevOps Services service) simplifies setting up a robust deployment pipeline to publish updates for apps hosted in Azure App Service. The pipeline can be configured from the Azure portal to build, run tests, deploy to a staging slot, and then deploy to production.

> **NOTE**
>
> To complete this tutorial, a Microsoft Azure account is required. To obtain an account, activate MSDN subscriber benefits or sign up for a free trial.

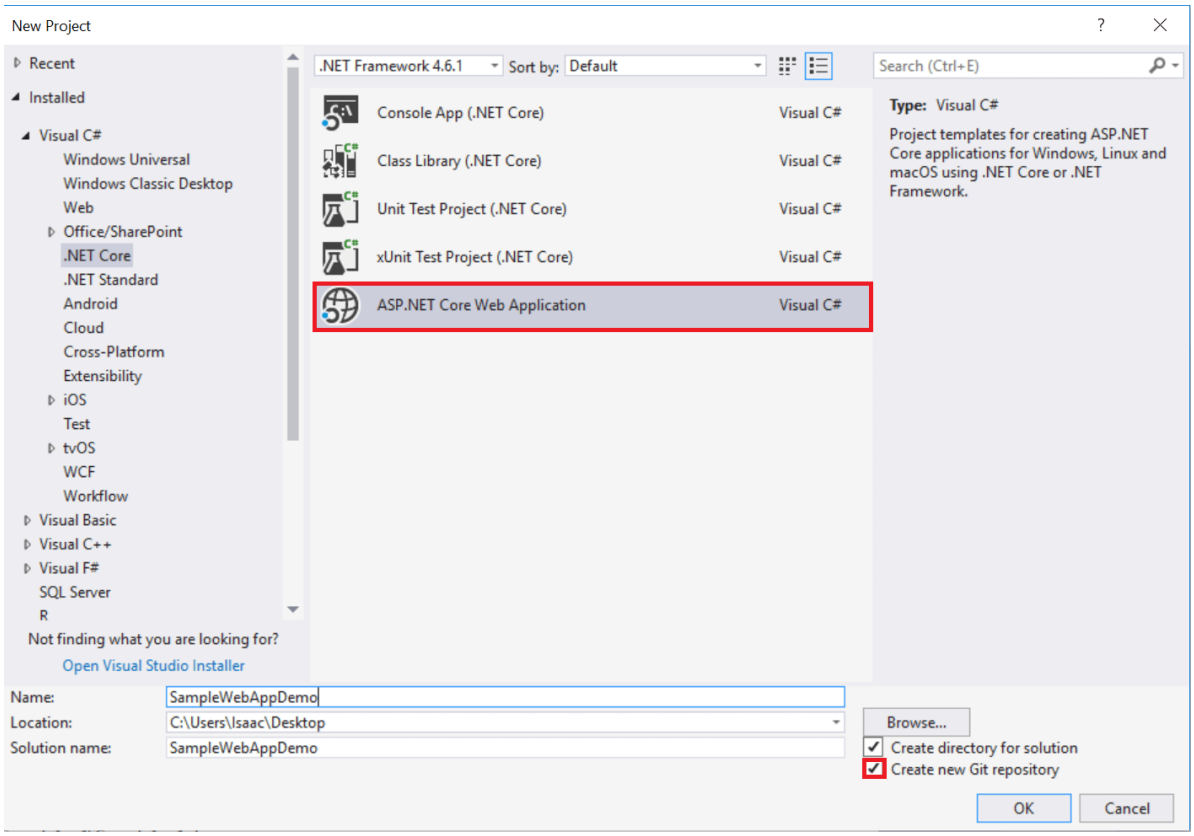
Prerequisites

This tutorial assumes the following software is installed:

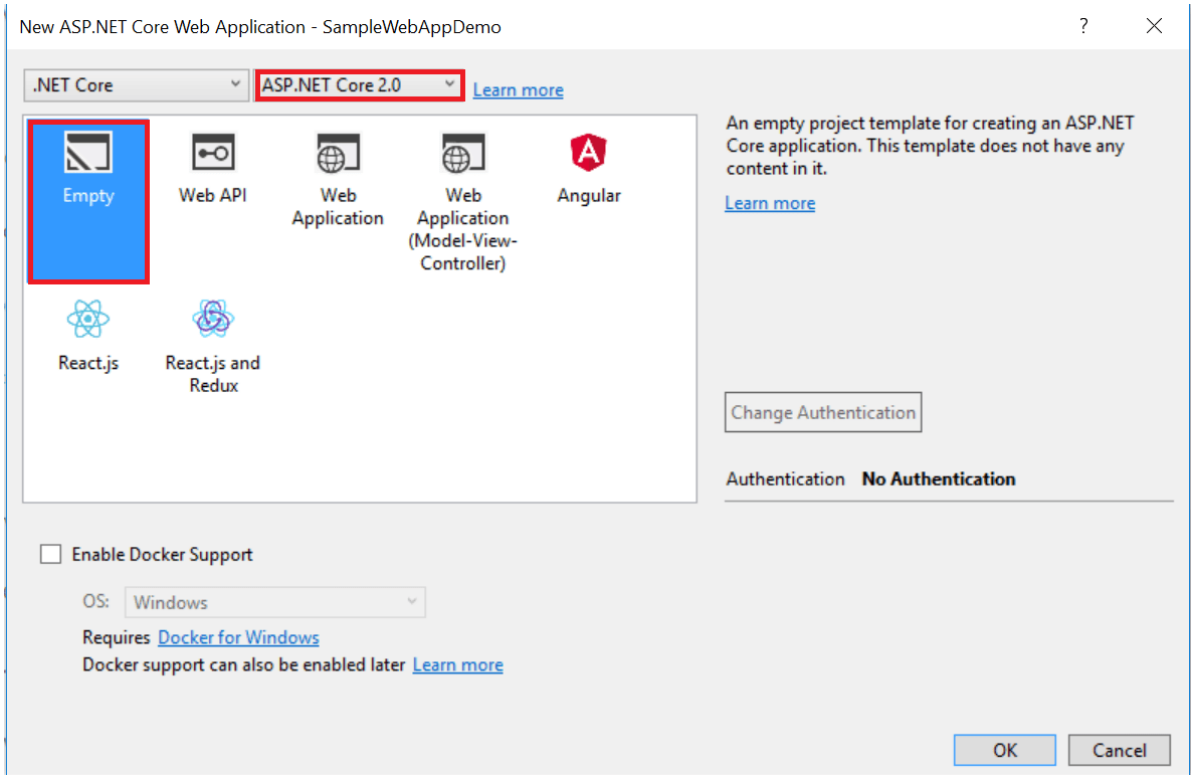
- Visual Studio
- .NET Core SDK 2.0 or later
- Git for Windows

Create an ASP.NET Core web app

1. Start Visual Studio.
2. From the **File** menu, select **New** > **Project**.
3. Select the **ASP.NET Core Web Application** project template. It appears under **Installed** > **Templates** > **Visual C#** > **.NET Core**. Name the project `SampleWebAppDemo`. Select the **Create new Git repository** option and click **OK**.



4. In the **New ASP.NET Core Project** dialog, select the ASP.NET Core **Empty** template, then click **OK**.



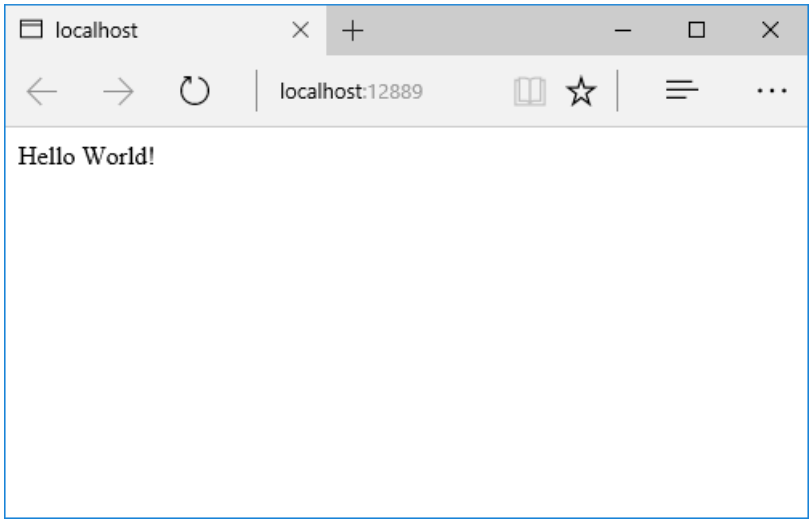
> **NOTE**
>
> The most recent release of .NET Core is 2.0.

Running the web app locally

1. Once Visual Studio finishes creating the app, run the app by selecting **Debug** > **Start Debugging**. As an alternative, press **F5**.

   It may take time to initialize Visual Studio and the new app. Once it's complete, the browser shows the

running app.

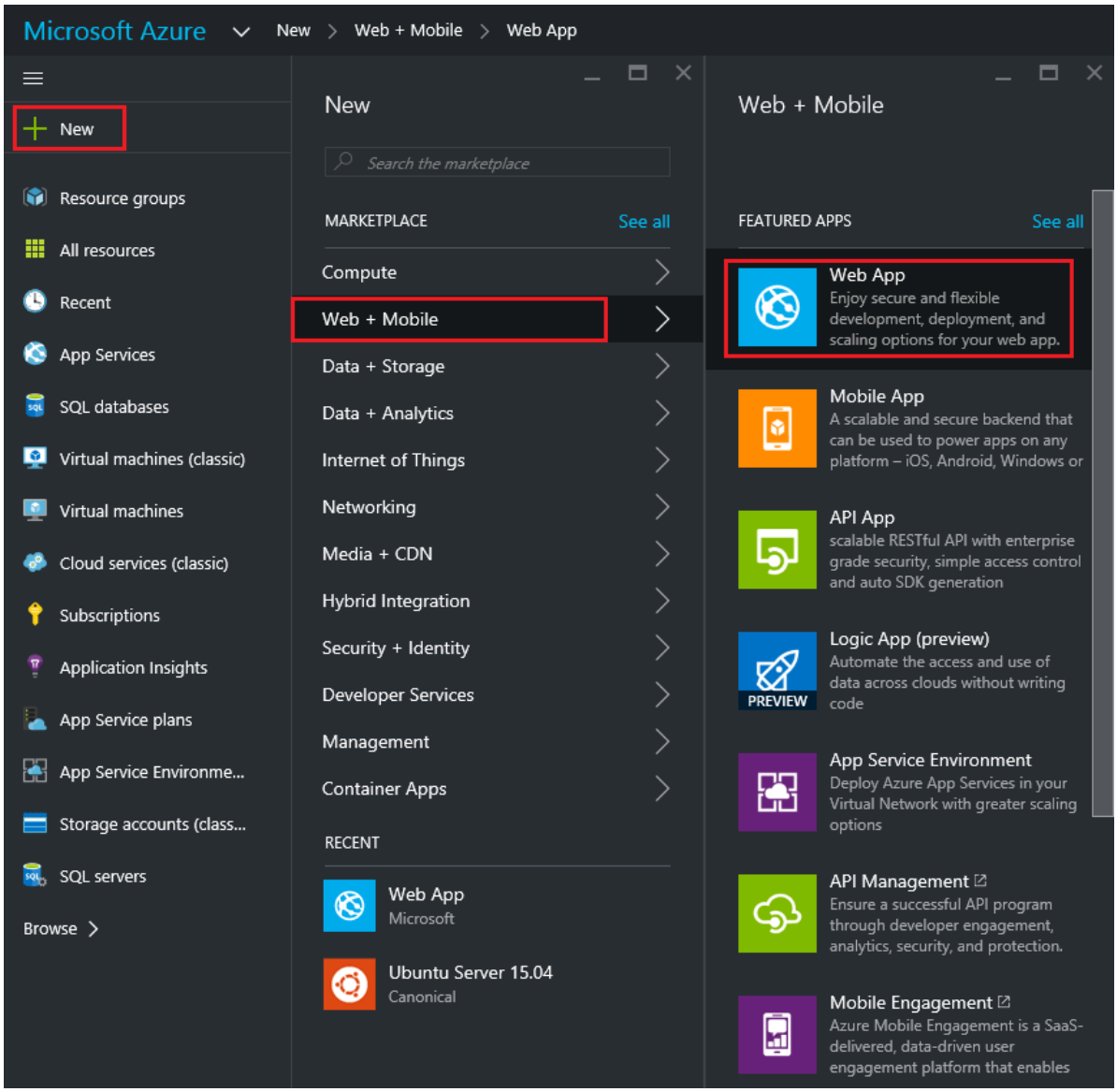


2. After reviewing the running Web app, close the browser and select the "Stop Debugging" icon in the toolbar of Visual Studio to stop the app.

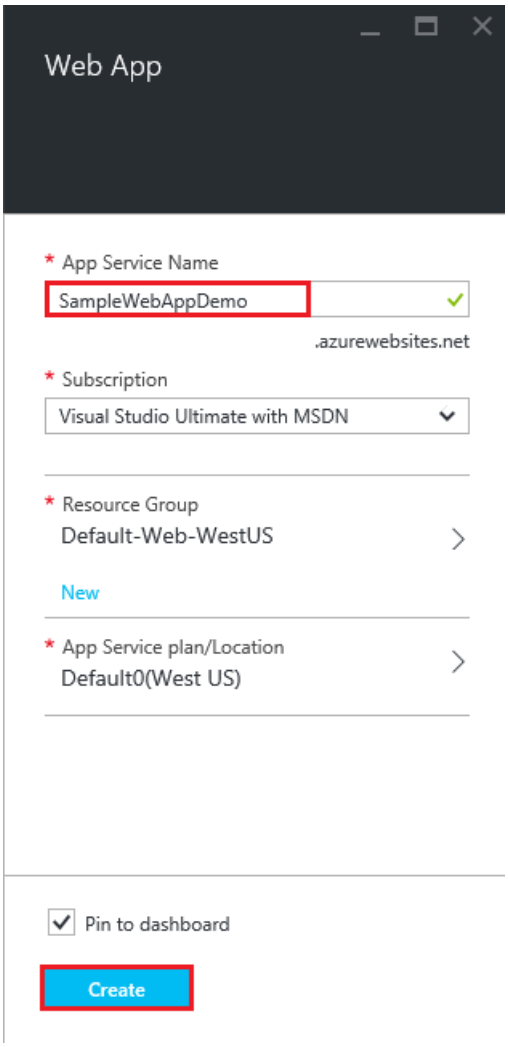
Create a web app in the Azure Portal

The following steps create a web app in the Azure Portal:

1. Log in to the Azure Portal.
2. Select **NEW** at the top left of the portal interface.
3. Select **Web + Mobile** > **Web App**.



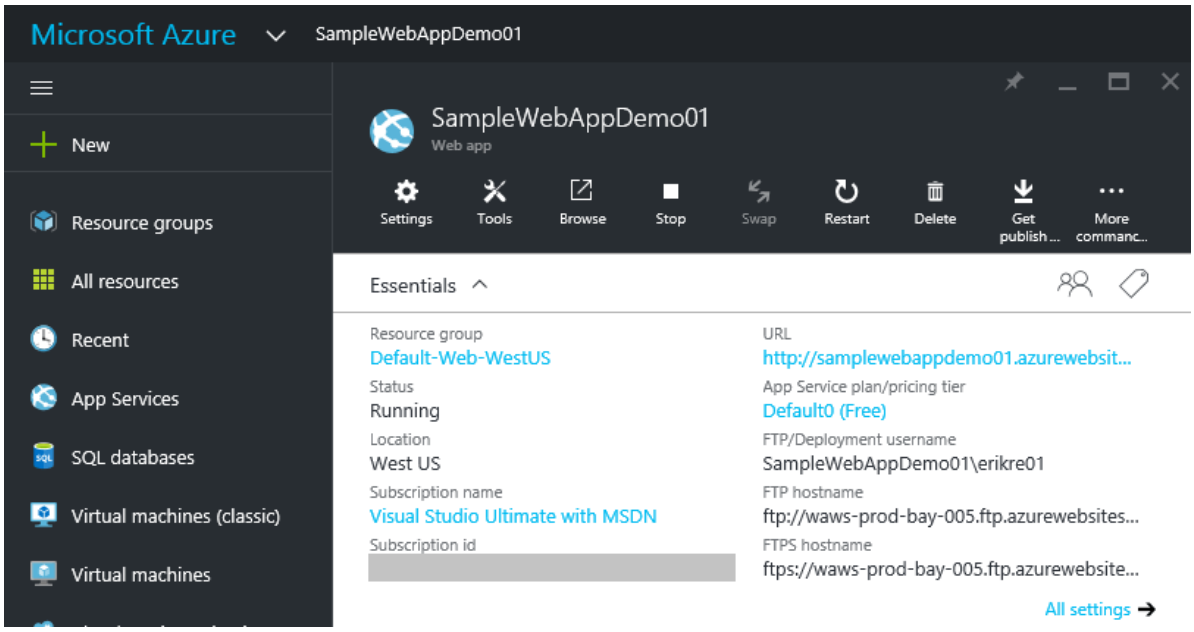
4. In the **Web App** blade, enter a unique value for the **App Service Name**.



> **NOTE**
>
> The **App Service Name** name must be unique. The portal enforces this rule when the name is provided. If providing a different value, substitute that value for each occurrence of **SampleWebAppDemo** in this tutorial.

Also in the **Web App** blade, select an existing **App Service Plan/Location** or create a new one. If creating a new plan, select the pricing tier, location, and other options. For more information on App Service plans, see Azure App Service plans in-depth overview.

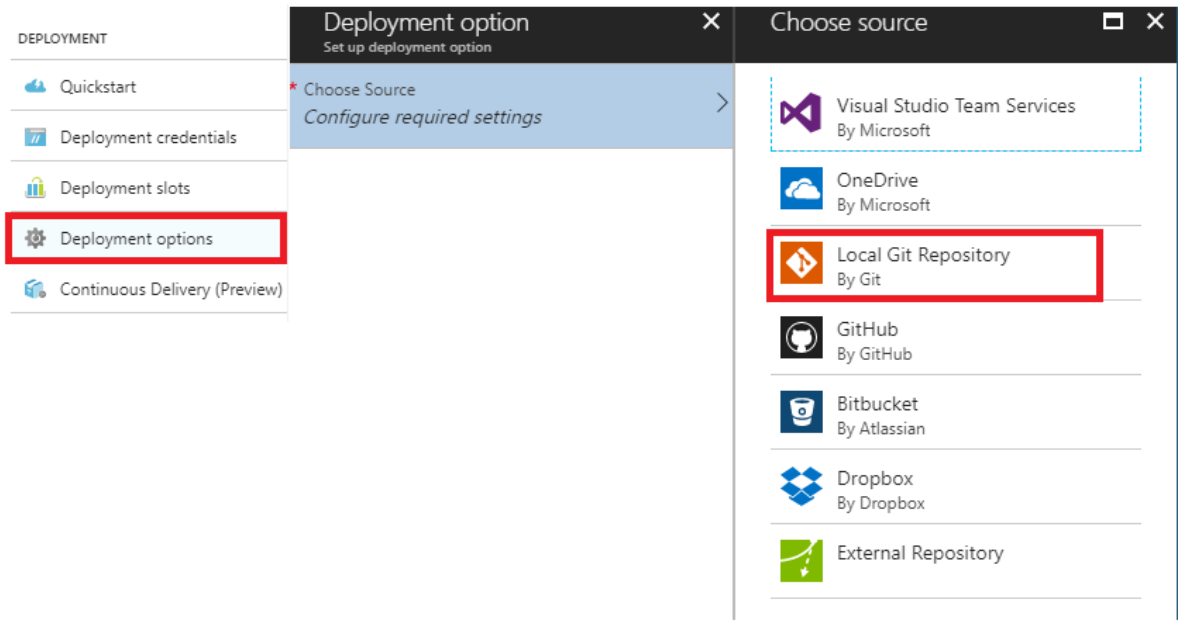
5. Select **Create**. Azure will provision and start the web app.



Enable Git publishing for the new web app

Git is a distributed version control system that can be used to deploy an Azure App Service web app. Web app code is stored in a local Git repository, and the code is deployed to Azure by pushing to a remote repository.

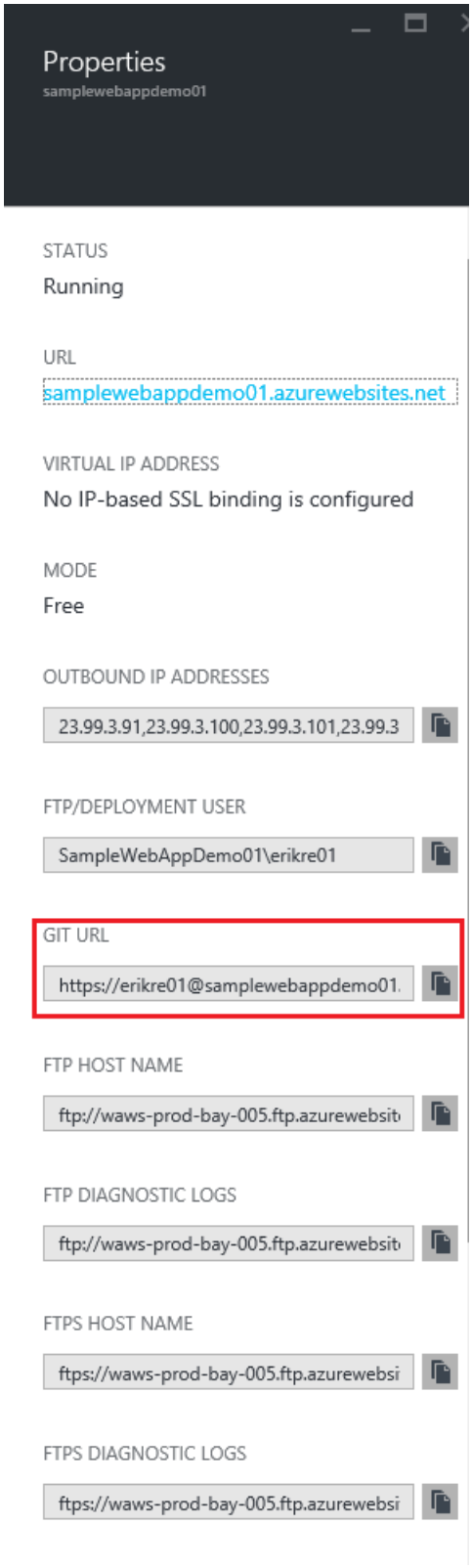
1. Log into the Azure Portal.
2. Select **App Services** to view a list of the app services associated with the Azure subscription.
3. Select the web app created in the previous section of this tutorial.
4. In the **Deployment** blade, select **Deployment options** > **Choose Source** > **Local Git Repository**.



5. Select **OK**.
6. If deployment credentials for publishing a web app or other App Service app haven't previously been set up, set them up now:
   - Select **Settings** > **Deployment credentials**. The **Set deployment credentials** blade is displayed.
   - Create a user name and password. Save the password for later use when setting up Git.
   - Select **Save**.
7. In the **Web App** blade, select **Settings** > **Properties**. The URL of the remote Git repository to deploy to is

shown under **GIT URL**.

8. Copy the **GIT URL** value for later use in the tutorial.

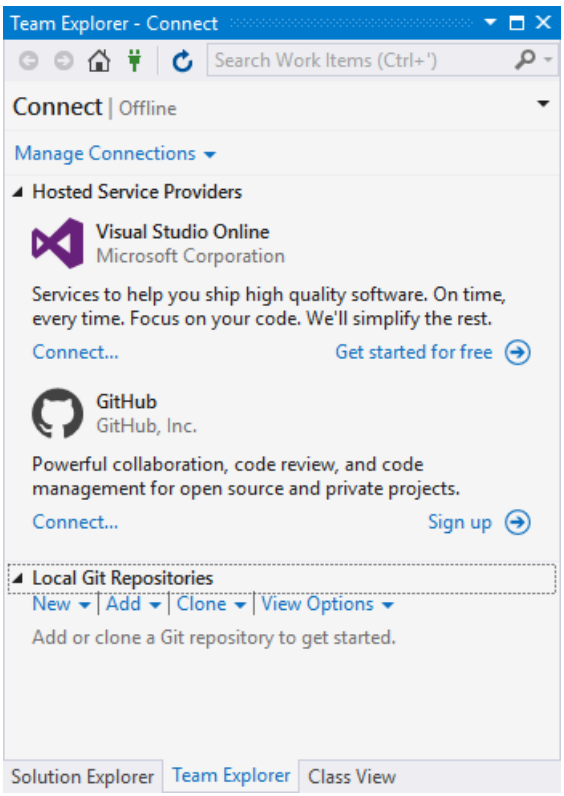


Publish the web app to Azure App Service

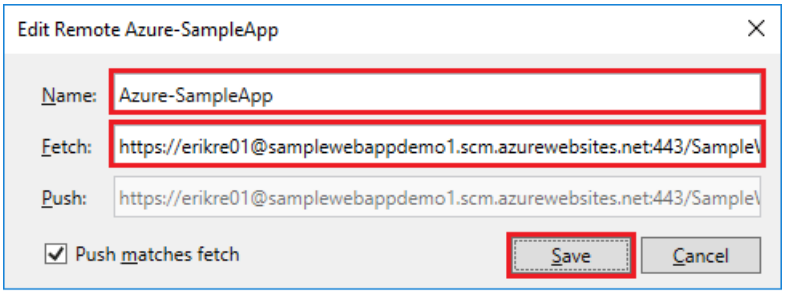
In this section, create a local Git repository using Visual Studio and push from that repository to Azure to deploy the web app. The steps involved include the following:

- Add the remote repository setting using the GIT URL value, so the local repository can be deployed to Azure.
- Commit project changes.
- Push project changes from the local repository to the remote repository on Azure.

1. In **Solution Explorer** right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** is displayed.

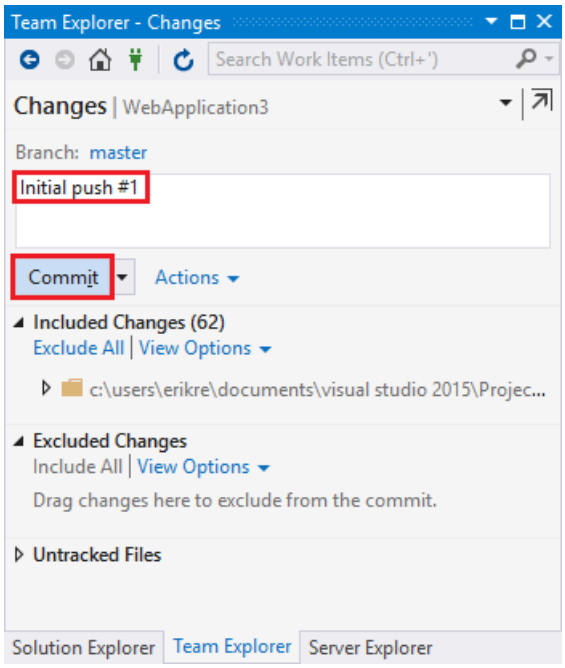


2. In **Team Explorer**, select the **Home** (home icon) > **Settings** > **Repository Settings**.
3. In the **Remotes** section of the **Repository Settings**, select **Add**. The **Add Remote** dialog box is displayed.
4. Set the **Name** of the remote to **Azure-SampleApp**.
5. Set the value for **Fetch** to the **Git URL** that copied from Azure earlier in this tutorial. Note that this is the URL that ends with **.git**.



> **NOTE**
>
> As an alternative, specify the remote repository from the **Command Window** by opening the **Command Window**, changing to the project directory, and entering the command. Example:
>
> ```
> git remote add Azure-SampleApp https://me@sampleapp.scm.azurewebsites.net:443/SampleApp.git
> ```

6. Select the **Home** (home icon) > **Settings** > **Global Settings**. Confirm that the name and email address are set. Select **Update** if required.
7. Select **Home** > **Changes** to return to the **Changes** view.
8. Enter a commit message, such as **Initial Push #1** and select **Commit**. This action creates a *commit* locally.



> **NOTE**
>
> As an alternative, commit changes from the **Command Window** by opening the **Command Window**, changing to the project directory, and entering the git commands. Example:
>
> ```
> git add .
> ```
>
> ```
> git commit -am "Initial Push #1"
> ```

9. Select **Home** > **Sync** > **Actions** > **Open Command Prompt**. The command prompt opens to the project directory.

10. Enter the following command in the command window:

    ```
    git push -u Azure-SampleApp master
    ```

11. Enter the Azure **deployment credentials** password created earlier in Azure.

    This command starts the process of pushing the local project files to Azure. The output from the above command ends with a message that the deployment was successful.

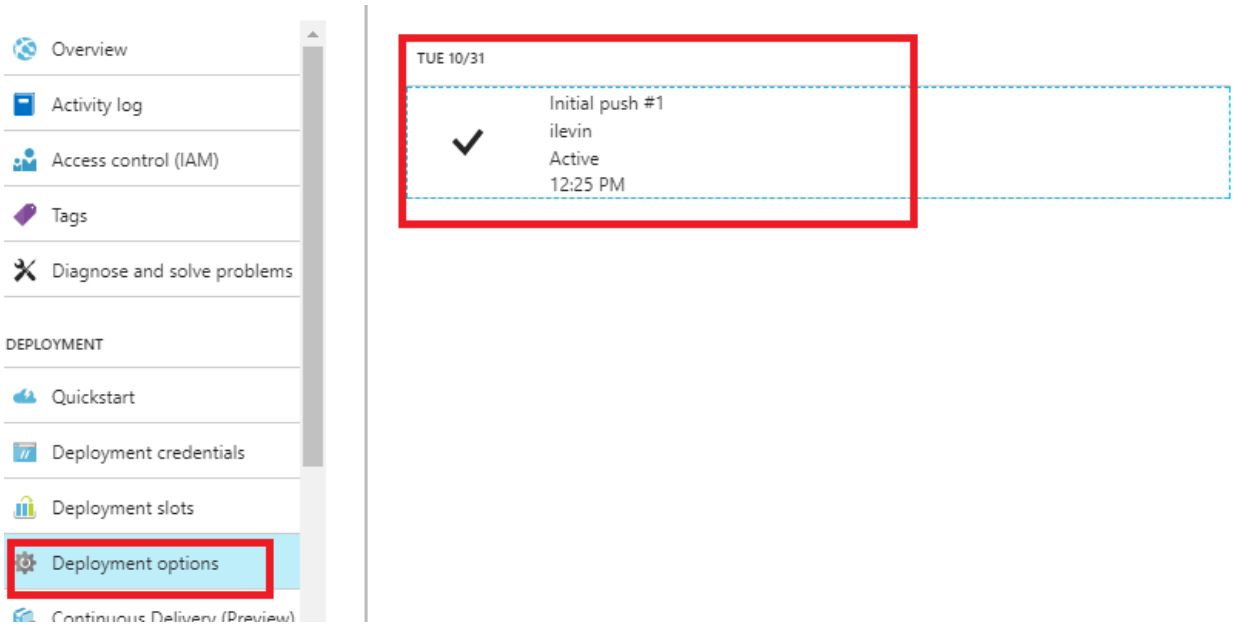
    ```
    remote: Finished successfully.
    remote: Running post deployment command(s)...
    remote: Deployment successful.
    To https://username@samplewebappdemo01.scm.azurewebsites.net:443/SampleWebAppDemo01.git
    * [new branch]      master -> master
    Branch master set up to track remote branch master from Azure-SampleApp.
    ```

> **NOTE**
>
> If collaboration on the project is required, consider pushing to GitHub before pushing to Azure.

Verify the Active Deployment

Verify that the web app transfer from the local environment to Azure is successful.

In the Azure Portal, select the web app. Select **Deployment** > **Deployment options**.



Run the app in Azure

Now that the web app is deployed to Azure, run the app.

This can be accomplished in two ways:

- In the Azure Portal, locate the web app blade for the web app. Select **Browse** to view the app in the default browser.
- Open a browser and enter the URL for the web app. Example: `http://SampleWebAppDemo.azurewebsites.net`

Update the web app and republish

After making changes to the local code, republish:

1. In **Solution Explorer** of Visual Studio, open the *Startup.cs* file.
2. In the `Configure` method, modify the `Response.WriteAsync` method so that it appears as follows:

```
await context.Response.WriteAsync("Hello World! Deploy to Azure.");
```

3. Save the changes to *Startup.cs*.
4. In **Solution Explorer**, right-click **Solution 'SampleWebAppDemo'** and select **Commit**. The **Team Explorer** is displayed.
5. Enter a commit message, such as `Update #2`.
6. Press the **Commit** button to commit the project changes.
7. Select **Home** > **Sync** > **Actions** > **Push**.

> **NOTE**
>
> As an alternative, push the changes from the **Command Window** by opening the **Command Window**, changing to the project directory, and entering a git command. Example:
>
> `git push -u Azure-SampleApp master`

View the updated web app in Azure

View the updated web app by selecting **Browse** from the web app blade in the Azure Portal or by opening a browser and entering the URL for the web app. Example: `http://SampleWebAppDemo.azurewebsites.net`

Additional resources

- Create your first pipeline with Azure Pipelines
- Project Kudu
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

ASP.NET Core Module

9/22/2020 • 40 minutes to read • Edit Online

By Tom Dykstra, Rick Strahl, Chris Ross, Rick Anderson, Sourabh Shirhatti, and Justin Kotalik

The ASP.NET Core Module is a native IIS module that plugs into the IIS pipeline to either:

- Host an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forward web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

Supported Windows versions:

- Windows 7 or later
- Windows Server 2012 R2 or later

When hosting in-process, the module uses an in-process server implementation for IIS, called IIS HTTP Server (`IISHttpServer`).

When hosting out-of-process, the module only works with Kestrel. The module doesn't function with HTTP.sys.

Hosting models

In-process hosting model

ASP.NET Core apps default to the in-process hosting model.

The following characteristics apply when hosting in-process:

- IIS HTTP Server (`IISHttpServer`) is used instead of Kestrel server. For in-process, CreateDefaultBuilder calls UseIIS to:
  - Register the `IISHttpServer`.
  - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
  - Configure the host to capture startup errors.
- The requestTimeout attribute doesn't apply to in-process hosting.
- Sharing an app pool among apps isn't supported. Use one app pool per app.
- When using Web Deploy or manually placing an app_offline.htm file in the deployment, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.
- The architecture (bitness) of the app and installed runtime (x64 or x86) must match the architecture of the app pool.
- Client disconnects are detected. The HttpContext.RequestAborted cancellation token is cancelled when the client disconnects.
- In ASP.NET Core 2.2.1 or earlier, GetCurrentDirectory returns the worker directory of the process started by IIS rather than the app's directory (for example, *C:\Windows\System32\inetsrv* for *w3wp.exe*).

  For sample code that sets the app's current directory, see the CurrentDirectoryHelpers class. Call the `SetCurrentDirectory` method. Subsequent calls to GetCurrentDirectory provide the app's directory.

- When hosting in-process, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. When transforming claims with an IClaimsTransformation implementation, call AddAuthentication to add authentication services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IClaimsTransformation, ClaimsTransformer>();
    services.AddAuthentication(IISServerDefaults.AuthenticationScheme);
}

public void Configure(IApplicationBuilder app)
{
    app.UseAuthentication();
}
```

  - Web Package (single-file) deployments aren't supported.

Out-of-process hosting model

To configure an app for out-of-process hosting, set the value of the `<AspNetCoreHostingModel>` property to `OutOfProcess` in the project file (*.csproj*):

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

In-process hosting is set with `InProcess`, which is the default value.

The value of `<AspNetCoreHostingModel>` is case insensitive, so `inprocess` and `outofprocess` are valid values.

Kestrel server is used instead of IIS HTTP Server (`IISHttpServer`).

For out-of-process, CreateDefaultBuilder calls UseIISIntegration to:

- Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
- Configure the host to capture startup errors.

Hosting model changes

If the `hostingModel` setting is changed in the *web.config* file (explained in the Configuration with web.config section), the module recycles the worker process for IIS.

For IIS Express, the module doesn't recycle the worker process but instead triggers a graceful shutdown of the current IIS Express process. The next request to the app spawns a new IIS Express process.

Process name

`Process.GetCurrentProcess().ProcessName` reports `w3wp` / `iisexpress` (in-process) or `dotnet` (out-of-process).

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

How to install and use the ASP.NET Core Module

For instructions on how to install the ASP.NET Core Module, see Install the .NET Core Hosting Bundle.

Configuration with web.config

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's *web.config* file.

The following *web.config* file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\MyApp.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The following *web.config* is published for a self-contained deployment:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the app.

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the *LogFiles* folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Host ASP.NET Core on Windows with IIS.

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION	DEFAULT
`arguments`	Optional string attribute. Arguments to the executable specified in **processPath**.	

ATTRIBUTE	DESCRIPTION	DEFAULT
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code page configured in the *web.config* takes precedence.	`false`
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`hostingModel`	Optional string attribute. Specifies the hosting model as in-process (`InProcess` / `inprocess`) or out-of-process (`OutOfProcess` / `outofprocess`).	`InProcess` `inprocess`
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the **processPath** setting that can be spun up per app. †For in-process hosting, the value is limited to `1` . Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100` †
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.` , the path is considered to be relative to the site root.	

ATTRIBUTE	DESCRIPTION	DEFAULT
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the module stops launching the process for the remainder of the minute. Not supported with in-process hosting.	Default: `10` Min: `0` Max: `100`
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds. Doesn't apply to in-process hosting. For in-process hosting, the module waits for the app to process the request. Valid values for minutes and seconds segments of the string are in the range 0-59. Use of **60** in the value for minutes or seconds results in a *500 - Internal Server Error*.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the *app_offline.htm* file is detected.	Default: `10` Min: `0` Max: `600`

ATTRIBUTE	DESCRIPTION	DEFAULT
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this time limit is exceeded, the module kills the process. When hosting *in-process*: The process is **not** restarted and does **not** use the **rapidFailsPerMinute** setting. When hosting *out-of-process*: The module attempts to relaunch the process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start **rapidFailsPerMinute** number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	Default: `120` Min: `0` Max: `3600`
`stdoutLogEnabled`	Optional Boolean attribute. If true, **stdout** and **stderr** for the process specified in **processPath** are redirected to the file specified in **stdoutLogFile**.	`false`
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which **stdout** and **stderr** from the process specified in **processPath** are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path are created by the module when the log file is created. Using underscore delimiters, a timestamp, process ID, and file extension (*.log*) are added to the last segment of the **stdoutLogFile** path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as *stdout_20180205194132_1934.log* in the *logs* folder when saved on 2/5/2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Set environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment

variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element. Environment variables set in this section take precedence over system environment variables.

The following example sets two environment variables in *web.config*. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

NOTE

An alternative to setting the environment directly in *web.config* is to include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

WARNING

Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

app_offline.htm

If a file with the name *app_offline.htm* is detected in the root directory of an app, the ASP.NET Core Module attempts to gracefully shutdown the app and stop processing incoming requests. If the app is still running after the number of seconds defined in `shutdownTimeLimit`, the ASP.NET Core Module kills the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module responds to requests by sending back the contents of the *app_offline.htm* file. When the *app_offline.htm* file is removed, the next request starts the app.

When using the out-of-process hosting model, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.

Start-up error page

Both in-process and out-of-process hosting produce custom error pages when they fail to start the app.

If the ASP.NET Core Module fails to find either the in-process or out-of-process request handler, a *500.0 - In-Process/Out-Of-Process Handler Load Failure* status code page appears.

For in-process hosting if the ASP.NET Core Module fails to start the app, a *500.30 - Start Failure* status code page appears.

For out-of-process hosting if the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, a *502.5 - Process Failure* status code page appears.

To suppress this page and revert to the default IIS 5xx status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see HTTP Errors <httpErrors>.

Log creation and redirection

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (*.log*) to the last segment of the `stdoutLogFile` path (typically *stdout*) delimited by underscores. If the `stdoutLogFile` path ends with *stdout*, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name *stdout_20180205194132_1934.log*.

If `stdoutLogEnabled` is false, errors that occur on app startup are captured and emitted to the event log up to 30 KB. After startup, all additional logs are discarded.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout"
    hostingModel="inprocess">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is predefined for apps hosted by Azure App Service.

To create logging filter rules, see the Configuration and Log filtering sections of the ASP.NET Core logging documentation.

For more information on path formats, see File path formats on Windows systems.

Enhanced diagnostic logs

The ASP.NET Core Module is configurable to provide enhanced diagnostics logs. Add the `<handlerSettings>` element to the `<aspNetCore>` element in *web.config*. Setting the `debugLevel` to `TRACE` exposes a higher fidelity of diagnostic information:

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="debugFile" value=".\logs\aspnetcore-debug.log" />
    <handlerSetting name="debugLevel" value="FILE,TRACE" />
  </handlerSettings>
</aspNetCore>
```

Any folders in the path (*logs* in the preceding example) are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Debug level (`debugLevel`) values can include both the level and the location.

Levels (in order from least to most verbose):

- ERROR
- WARNING
- INFO
- TRACE

Locations (multiple locations are permitted):

- CONSOLE
- EVENTLOG
- FILE

The handler settings can also be provided via environment variables:

- `ASPNETCORE_MODULE_DEBUG_FILE`: Path to the debug log file. (Default: *aspnetcore-debug.log*)
- `ASPNETCORE_MODULE_DEBUG`: Debug level setting.

> **WARNING**
> Do **not** leave debug logging enabled in the deployment for longer than required to troubleshoot an issue. The size of the log isn't limited. Leaving the debug log enabled can exhaust the available disk space and crash the server or app service.

See Configuration with web.config for an example of the `aspNetCore` element in the *web.config* file.

Modify the stack size

Only applies when using the in-process hosting model.

Configure the managed stack size using the `stackSize` setting in bytes in *web.config*. The default size is `1048576` bytes (1 MB).

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="stackSize" value="2097152" />
  </handlerSettings>
</aspNetCore>
```

Proxy configuration uses HTTP protocol and a pairing token

Only applies to out-of-process hosting.

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, an attacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **TrustedInstaller** account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the *applicationHost.config* file on the share.

When using an IIS Shared Configuration on the same machine as the IIS installation, run the ASP.NET Core Hosting Bundle installer with the `OPT_NO_SHARED_CONFIG_CHECK` parameter set to `1`:

```
dotnet-hosting-{VERSION}.exe OPT_NO_SHARED_CONFIG_CHECK=1
```

When the path to the shared configuration isn't on the same machine as the IIS installation, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated *applicationHost.config* file to the share.
4. Re-enable the IIS Shared Configuration.

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to *%windir%\System32\inetsrv*.
2. Locate the *aspnetcore.dll* file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at *C:\Users\%UserName%\AppData\Local\Temp*. The file

is named *dd_DotNetCoreWinSvrHosting__ <timestamp>_000_AspNetCoreModule_x64.log*.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll
- %ProgramFiles%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll

IIS Express (x86/amd64):

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll
- %ProgramFiles%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll

Schema

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml
- %windir%\System32\inetsrv\config\schema\aspnetcore_schema_v2.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml
- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema_v2.xml

Configuration

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- Visual Studio: {APPLICATION ROOT}\.vs\config\applicationHost.config
- *iisexpress.exe* CLI: %USERPROFILE%\Documents\IISExpress\config\applicationhost.config

The files can be found by searching for *aspnetcore* in the *applicationHost.config* file.

The ASP.NET Core Module is a native IIS module that plugs into the IIS pipeline to either:

- Host an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forward web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

Supported Windows versions:

- Windows 7 or later
- Windows Server 2008 R2 or later

When hosting in-process, the module uses an in-process server implementation for IIS, called IIS HTTP Server (`IISHttpServer`).

When hosting out-of-process, the module only works with Kestrel. The module doesn't function with HTTP.sys.

Hosting models

In-process hosting model

To configure an app for in-process hosting, add the `<AspNetCoreHostingModel>` property to the app's project file with a value of `InProcess` (out-of-process hosting is set with `OutOfProcess`):

```
<PropertyGroup>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

The in-process hosting model isn't supported for ASP.NET Core apps that target the .NET Framework.

The value of `<AspNetCoreHostingModel>` is case insensitive, so `inprocess` and `outofprocess` are valid values.

If the `<AspNetCoreHostingModel>` property isn't present in the file, the default value is `OutOfProcess` .

The following characteristics apply when hosting in-process:

- IIS HTTP Server (`IISHttpServer`) is used instead of Kestrel server. For in-process, CreateDefaultBuilder calls UseIIS to:
  - Register the `IISHttpServer` .
  - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
  - Configure the host to capture startup errors.
- The requestTimeout attribute doesn't apply to in-process hosting.
- Sharing an app pool among apps isn't supported. Use one app pool per app.
- When using Web Deploy or manually placing an app_offline.htm file in the deployment, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.
- The architecture (bitness) of the app and installed runtime (x64 or x86) must match the architecture of the app pool.
- Client disconnects are detected. The HttpContext.RequestAborted cancellation token is cancelled when the client disconnects.
- In ASP.NET Core 2.2.1 or earlier, GetCurrentDirectory returns the worker directory of the process started by IIS rather than the app's directory (for example, *C:\Windows\System32\inetsrv* for *w3wp.exe*).

  For sample code that sets the app's current directory, see the CurrentDirectoryHelpers class. Call the `SetCurrentDirectory` method. Subsequent calls to GetCurrentDirectory provide the app's directory.
- When hosting in-process, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. When transforming claims with an IClaimsTransformation implementation, call AddAuthentication to add authentication services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IClaimsTransformation, ClaimsTransformer>();
    services.AddAuthentication(IISServerDefaults.AuthenticationScheme);
}

public void Configure(IApplicationBuilder app)
{
    app.UseAuthentication();
}
```

Out-of-process hosting model

To configure an app for out-of-process hosting, use either of the following approaches in the project file:

- Don't specify the `<AspNetCoreHostingModel>` property. If the `<AspNetCoreHostingModel>` property isn't present in the file, the default value is `OutOfProcess`.
- Set the value of the `<AspNetCoreHostingModel>` property to `OutOfProcess` (in-process hosting is set with `InProcess`):

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

The value is case insensitive, so `inprocess` and `outofprocess` are valid values.

Kestrel server is used instead of IIS HTTP Server (`IISHttpServer`).

For out-of-process, CreateDefaultBuilder calls UseIISIntegration to:

- Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
- Configure the host to capture startup errors.

Hosting model changes

If the `hostingModel` setting is changed in the *web.config* file (explained in the Configuration with web.config section), the module recycles the worker process for IIS.

For IIS Express, the module doesn't recycle the worker process but instead triggers a graceful shutdown of the current IIS Express process. The next request to the app spawns a new IIS Express process.

Process name

`Process.GetCurrentProcess().ProcessName` reports `w3wp` / `iisexpress` (in-process) or `dotnet` (out-of-process).

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

How to install and use the ASP.NET Core Module

For instructions on how to install the ASP.NET Core Module, see Install the .NET Core Hosting Bundle.

Configuration with web.config

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's *web.config* file.

The following *web.config* file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\MyApp.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The following *web.config* is published for a self-contained deployment:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the app.

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the *LogFiles* folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Host ASP.NET Core on Windows with IIS.

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION	DEFAULT
`arguments`	Optional string attribute. Arguments to the executable specified in **processPath**.	

ATTRIBUTE	DESCRIPTION	DEFAULT
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code page configured in the *web.config* takes precedence.	`false`
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`hostingModel`	Optional string attribute. Specifies the hosting model as in-process (`InProcess` / `inprocess`) or out-of-process (`OutOfProcess` / `outofprocess`).	`OutOfProcess` `outofprocess`
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the **processPath** setting that can be spun up per app. †For in-process hosting, the value is limited to `1`. Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100` †
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.`, the path is considered to be relative to the site root.	

ATTRIBUTE	DESCRIPTION	DEFAULT
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the module stops launching the process for the remainder of the minute. Not supported with in-process hosting.	Default: `10` Min: `0` Max: `100`
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds. Doesn't apply to in-process hosting. For in-process hosting, the module waits for the app to process the request. Valid values for minutes and seconds segments of the string are in the range 0-59. Use of **60** in the value for minutes or seconds results in a *500 - Internal Server Error*.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the *app_offline.htm* file is detected.	Default: `10` Min: `0` Max: `600`

ATTRIBUTE	DESCRIPTION	DEFAULT
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this time limit is exceeded, the module kills the process. When hosting *in-process*: The process is **not** restarted and does **not** use the **rapidFailsPerMinute** setting. When hosting *out-of-process*: The module attempts to relaunch the process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start **rapidFailsPerMinute** number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	Default: `120` Min: `0` Max: `3600`
`stdoutLogEnabled`	Optional Boolean attribute. If true, **stdout** and **stderr** for the process specified in **processPath** are redirected to the file specified in **stdoutLogFile**.	`false`
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which **stdout** and **stderr** from the process specified in **processPath** are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path are created by the module when the log file is created. Using underscore delimiters, a timestamp, process ID, and file extension (*.log*) are added to the last segment of the **stdoutLogFile** path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as *stdout_20180205194132_1934.log* in the *logs* folder when saved on 2/5/2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Setting environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element. Environment variables set in this section take precedence over system environment variables.

The following example sets two environment variables. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

NOTE

An alternative to setting the environment directly in *web.config* is to include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

WARNING

Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

app_offline.htm

If a file with the name *app_offline.htm* is detected in the root directory of an app, the ASP.NET Core Module attempts to gracefully shutdown the app and stop processing incoming requests. If the app is still running after the number of seconds defined in `shutdownTimeLimit`, the ASP.NET Core Module kills the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module responds to requests by sending back the contents of the *app_offline.htm* file. When the *app_offline.htm* file is removed, the next request starts the app.

When using the out-of-process hosting model, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.

Start-up error page

Both in-process and out-of-process hosting produce custom error pages when they fail to start the app.

If the ASP.NET Core Module fails to find either the in-process or out-of-process request handler, a *500.0 - In-Process/Out-Of-Process Handler Load Failure* status code page appears.

For in-process hosting if the ASP.NET Core Module fails to start the app, a *500.30 - Start Failure* status code page

appears.

For out-of-process hosting if the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, a *502.5 - Process Failure* status code page appears.

To suppress this page and revert to the default IIS 5xx status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see HTTP Errors <httpErrors>.

Log creation and redirection

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (*.log*) to the last segment of the `stdoutLogFile` path (typically *stdout*) delimited by underscores. If the `stdoutLogFile` path ends with *stdout*, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name *stdout_20180205194132_1934.log*.

If `stdoutLogEnabled` is false, errors that occur on app startup are captured and emitted to the event log up to 30 KB. After startup, all additional logs are discarded.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout"
    hostingModel="inprocess">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is predefined for apps hosted by Azure App Service.

For more information on path formats, see File path formats on Windows systems.

Enhanced diagnostic logs

The ASP.NET Core Module is configurable to provide enhanced diagnostics logs. Add the `<handlerSettings>` element to the `<aspNetCore>` element in *web.config*. Setting the `debugLevel` to `TRACE` exposes a higher fidelity of diagnostic information:

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="debugFile" value=".\logs\aspnetcore-debug.log" />
    <handlerSetting name="debugLevel" value="FILE,TRACE" />
  </handlerSettings>
</aspNetCore>
```

Folders in the path provided to the `<handlerSetting>` value (*logs* in the preceding example) aren't created by the module automatically and should pre-exist in the deployment. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Debug level (`debugLevel`) values can include both the level and the location.

Levels (in order from least to most verbose):

- ERROR
- WARNING
- INFO
- TRACE

Locations (multiple locations are permitted):

- CONSOLE
- EVENTLOG
- FILE

The handler settings can also be provided via environment variables:

- `ASPNETCORE_MODULE_DEBUG_FILE`: Path to the debug log file. (Default: *aspnetcore-debug.log*)
- `ASPNETCORE_MODULE_DEBUG`: Debug level setting.

> **WARNING**
> Do **not** leave debug logging enabled in the deployment for longer than required to troubleshoot an issue. The size of the log isn't limited. Leaving the debug log enabled can exhaust the available disk space and crash the server or app service.

See Configuration with web.config for an example of the `aspNetCore` element in the *web.config* file.

Proxy configuration uses HTTP protocol and a pairing token

Only applies to out-of-process hosting.

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, an attacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **TrustedInstaller** account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the *applicationHost.config* file on the share.

When using an IIS Shared Configuration on the same machine as the IIS installation, run the ASP.NET Core Hosting Bundle installer with the `OPT_NO_SHARED_CONFIG_CHECK` parameter set to `1`:

```
dotnet-hosting-{VERSION}.exe OPT_NO_SHARED_CONFIG_CHECK=1
```

When the path to the shared configuration isn't on the same machine as the IIS installation, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated *applicationHost.config* file to the share.
4. Re-enable the IIS Shared Configuration.

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to *%windir%\System32\inetsrv*.
2. Locate the *aspnetcore.dll* file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at *C:\Users\%UserName%\AppData\Local\Temp*. The file is named *dd_DotNetCoreWinSvrHosting__<timestamp>_000_AspNetCoreModule_x64.log*.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll
- %ProgramFiles%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll

IIS Express (x86/amd64):

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll
- %ProgramFiles%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll

Schema

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml
- %windir%\System32\inetsrv\config\schema\aspnetcore_schema_v2.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml
- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema_v2.xml

Configuration

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- Visual Studio: {APPLICATION ROOT}\.vs\config\applicationHost.config
- *iisexpress.exe* CLI: %USERPROFILE%\Documents\IISExpress\config\applicationhost.config

The files can be found by searching for *aspnetcore* in the *applicationHost.config* file.

The ASP.NET Core Module is a native IIS module that plugs into the IIS pipeline to forward web requests to backend ASP.NET Core apps.

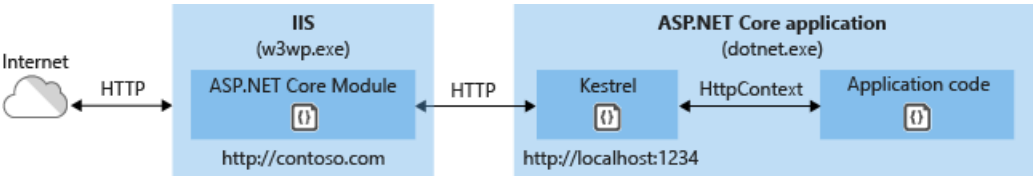
Supported Windows versions:

- Windows 7 or later
- Windows Server 2008 R2 or later

The module only works with Kestrel. The module is incompatible with HTTP.sys.

Because ASP.NET Core apps run in a process separate from the IIS worker process, the module also handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it crashes. This is essentially the same behavior as seen with ASP.NET 4.x apps that run in-process in IIS that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the IIS Integration Middleware configures the server to listen on `http://localhost:{port}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

How to install and use the ASP.NET Core Module

For instructions on how to install the ASP.NET Core Module, see Install the .NET Core Hosting Bundle.

Configuration with web.config

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's *web.config* file.

The following *web.config* file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet"
                arguments=".\MyApp.dll"
                stdoutLogEnabled="false"
                stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

The following *web.config* is published for a self-contained deployment:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath=".\MyApp.exe"
                stdoutLogEnabled="false"
                stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the *LogFiles* folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Host ASP.NET Core on Windows with IIS.

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION	DEFAULT

ATTRIBUTE	DESCRIPTION	DEFAULT
`arguments`	Optional string attribute. Arguments to the executable specified in **processPath**.	
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code page configured in the *web.config* takes precedence.	`false`
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the **processPath** setting that can be spun up per app. Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100`
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.`, the path is considered to be relative to the site root.	
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the module stops launching the process for the remainder of the minute.	Default: `10` Min: `0` Max: `100`

ATTRIBUTE	DESCRIPTION	DEFAULT
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the *app_offline.htm* file is detected.	Default: `10` Min: `0` Max: `600`
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this time limit is exceeded, the module kills the process. The module attempts to relaunch the process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start **rapidFailsPerMinute** number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	Default: `120` Min: `0` Max: `3600`
`stdoutLogEnabled`	Optional Boolean attribute. If true, **stdout** and **stderr** for the process specified in **processPath** are redirected to the file specified in **stdoutLogFile**.	`false`

ATTRIBUTE	DESCRIPTION	DEFAULT
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which **stdout** and **stderr** from the process specified in **processPath** are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path must exist in order for the module to create the log file. Using underscore delimiters, a timestamp, process ID, and file extension (*.log*) are added to the last segment of the **stdoutLogFile** path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as *stdout_20180205194132_1934.log* in the *logs* folder when saved on 2/5/2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Setting environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element.

> **WARNING**
>
> Environment variables set in this section conflict with system environment variables set with the same name. If an environment variable is set in both the *web.config* file and at the system level in Windows, the value from the *web.config* file becomes appended to the system environment variable value (for example, `ASPNETCORE_ENVIRONMENT: Development;Development`), which prevents the app from starting.

The following example sets two environment variables. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile="\\?\%home%\LogFiles\stdout">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

WARNING

Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

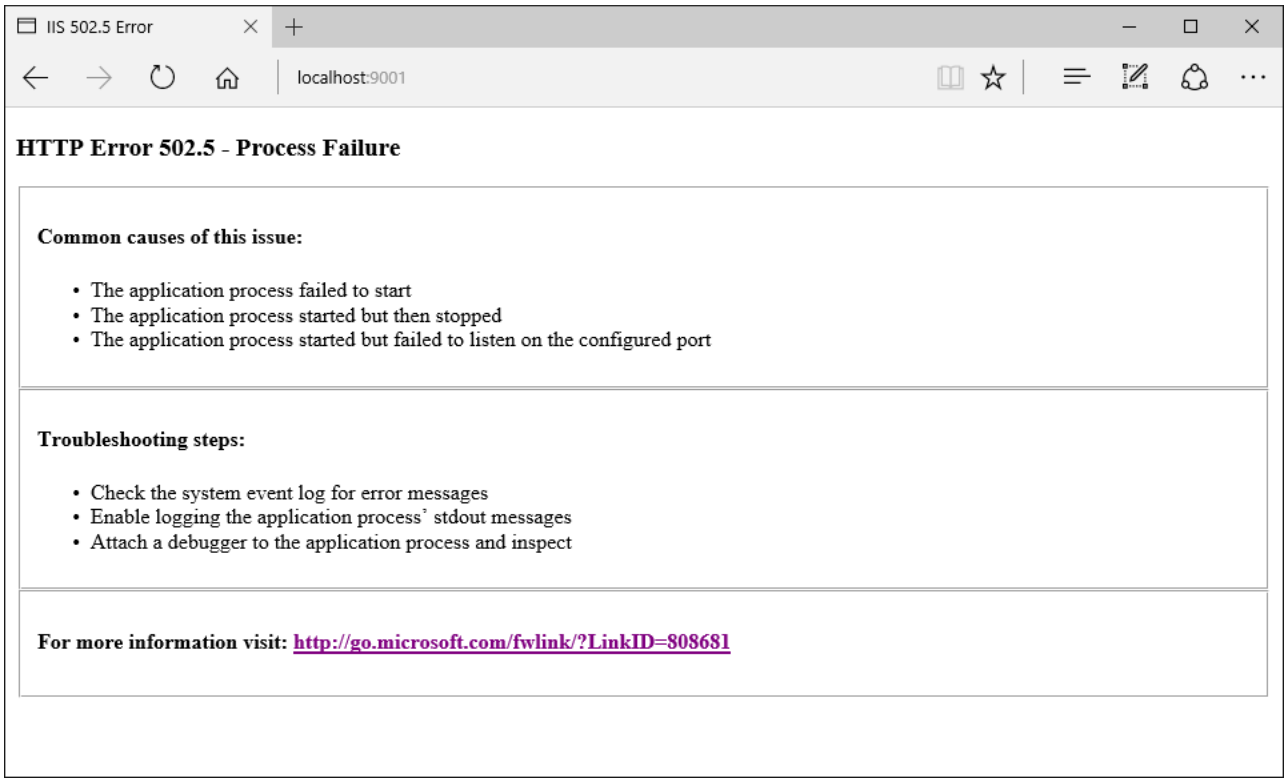
app_offline.htm

If a file with the name *app_offline.htm* is detected in the root directory of an app, the ASP.NET Core Module attempts to gracefully shutdown the app and stop processing incoming requests. If the app is still running after the number of seconds defined in `shutdownTimeLimit`, the ASP.NET Core Module kills the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module responds to requests by sending back the contents of the *app_offline.htm* file. When the *app_offline.htm* file is removed, the next request starts the app.

Start-up error page

If the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, a *502.5 - Process Failure* status code page appears. To suppress this page and revert to the default IIS 502 status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see HTTP Errors <httpErrors>.



Log creation and redirection

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS

Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (*.log*) to the last segment of the `stdoutLogFile` path (typically *stdout*) delimited by underscores. If the `stdoutLogFile` path ends with *stdout*, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name *stdout_20180205194132_1934.log*.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is predefined for apps hosted by Azure App Service.

To create logging filter rules, see the Configuration and Log filtering sections of the ASP.NET Core logging documentation.

For more information on path formats, see File path formats on Windows systems.

Proxy configuration uses HTTP protocol and a pairing token

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, an attacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **TrustedInstaller** account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the *applicationHost.config* file on the share.

When using an IIS Shared Configuration, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated *applicationHost.config* file to the share.
4. Re-enable the IIS Shared Configuration.

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to *%windir%\System32\inetsrv*.
2. Locate the *aspnetcore.dll* file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at *C:\Users\%UserName%\AppData\Local\Temp*. The file is named *dd_DotNetCoreWinSvrHosting__ <timestamp>_000_AspNetCoreModule_x64.log*.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll

IIS Express (x86/amd64):

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll

Schema

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml

Configuration

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- Visual Studio: {APPLICATION ROOT}\.vs\config\applicationHost.config
- *iisexpress.exe* CLI: %USERPROFILE%\Documents\IISExpress\config\applicationhost.config

The files can be found by searching for *aspnetcore* in the *applicationHost.config* file.

Additional resources

- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service
- ASP.NET Core Module reference source (master branch): Use the **Branch** drop down list to select a specific release (for example, `release/3.1`).
- IIS modules with ASP.NET Core

Troubleshoot ASP.NET Core on Azure App Service and IIS

9/22/2020 • 59 minutes to read • Edit Online

By Justin Kotalik

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.

- When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error,

either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file deployment model. Use **one** of the following approaches in the app's project file to fix

this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`
- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.
4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:
   - Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic

logging. Redeploy the app.

- Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
  a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
  b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
  c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.

2. Run the EnableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.

2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.

3. Restore and rebuild the project.

4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
   - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.
3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module fails to find the .NET Core CLR and find the in-process request handler (*aspnetcorev2_inprocess.dll*). Check that:

- The app targets either the Microsoft.AspNetCore.Server.IIS NuGet package or the Microsoft.AspNetCore.App metapackage.
- The version of the ASP.NET Core shared framework that the app targets is installed on the target machine.

500.0 Out-Of-Process Handler Load Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module fails to find the out-of-process hosting request handler. Make sure the *aspnetcorev2_outofprocess.dll* is present in a subfolder next to *aspnetcorev2.dll*.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout

log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.

2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Under **SELECT PROBLEM CATEGORY**, select the **Web App Down** button.
3. Under **Suggested Solutions** > **Enable Stdout Log Redirection**, select the button to **Open Kudu Console to edit Web.Config**.
4. In the Kudu **Diagnostic Console**, open the folders to the path **site** > **wwwroot**. Scroll down to reveal the *web.config* file at the bottom of the list.
5. Click the pencil icon next to the *web.config* file.
6. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
7. Select **Save** to save the updated *web.config* file.
8. Make a request to the app.
9. Return to the Azure portal. Select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
10. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
11. Select the **LogFiles** folder.
12. Inspect the **Modified** column and select the pencil icon to edit the stdout log with the latest modification date.
13. When the log file opens, the error is displayed.

Disable stdout logging when troubleshooting is complete:

1. In the Kudu **Diagnostic Console**, return to the path **site** > **wwwroot** to reveal the *web.config* file. Open the **web.config** file again by selecting the pencil icon.
2. Set **stdoutLogEnabled** to `false`.
3. Select **Save** to save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:
   - Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
   - Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
     a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
     b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
     c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.
2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.

4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.

2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.
2. Run the EnableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 Process Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
   - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.
3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.

2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`
- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Under **SELECT PROBLEM CATEGORY**, select the **Web App Down** button.
3. Under **Suggested Solutions** > **Enable Stdout Log Redirection**, select the button to **Open Kudu Console to edit Web.Config**.
4. In the Kudu **Diagnostic Console**, open the folders to the path **site** > **wwwroot**. Scroll down to reveal the *web.config* file at the bottom of the list.
5. Click the pencil icon next to the *web.config* file.
6. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
7. Select **Save** to save the updated *web.config* file.
8. Make a request to the app.
9. Return to the Azure portal. Select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
10. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
11. Select the **LogFiles** folder.
12. Inspect the **Modified** column and select the pencil icon to edit the stdout log with the latest modification date.
13. When the log file opens, the error is displayed.

Disable stdout logging when troubleshooting is complete:

1. In the Kudu **Diagnostic Console**, return to the path **site** > **wwwroot** to reveal the *web.config* file. Open the **web.config** file again by selecting the pencil icon.
2. Set **stdoutLogEnabled** to `false`.
3. Select **Save** to save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.

2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.

3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.
2. Run the EnableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview

- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common errors reference for Azure App Service and IIS with ASP.NET Core

9/22/2020 • 22 minutes to read • Edit Online

This topic describes common errors and provides troubleshooting advice for specific errors when hosting ASP.NET Core apps on Azure Apps Service and IIS.

For general troubleshooting guidance, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Collect the following information:

- Browser behavior (status code and error message)
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS
    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.
- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely
- **ASP.NET Core Module stdout Log:** The log file is created but empty.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED --**OR**-- Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied --**OR**-- 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS

website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. **--OR--** This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Download Archives and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.
- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure --**OR**-- HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

This topic describes common errors and provides troubleshooting advice for specific errors when hosting ASP.NET Core apps on Azure Apps Service and IIS.

For general troubleshooting guidance, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Collect the following information:

- Browser behavior (status code and error message)
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS
    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.
- ASP.NET Core Module stdout and debug log entries

- Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
- IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site

extension, restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED --**OR**-- Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.

- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied --**OR**-- 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. --**OR**-- This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-

compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0.
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Download Archives and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"dotnet" .{ASSEMBLY}.dll', ErrorCode = '0x80004005 : 80008081.
- **ASP.NET Core Module stdout Log:** The application to execute does not exist: 'PATH{ASSEMBLY}.dll'

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`)

for a self-contained deployment (SCD).

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Warning: Could not create stdoutLogFile \?{PATH}\path_doesnt_exist\stdout_{PROCESS ID}_{TIMESTAMP}.log, ErrorCode = -2147024893.
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' created process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ' but either crashed or did not respond or did not listen on the given port '{PORT}', ErrorCode = '{ERROR CODE}'
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

Troubleshooting:

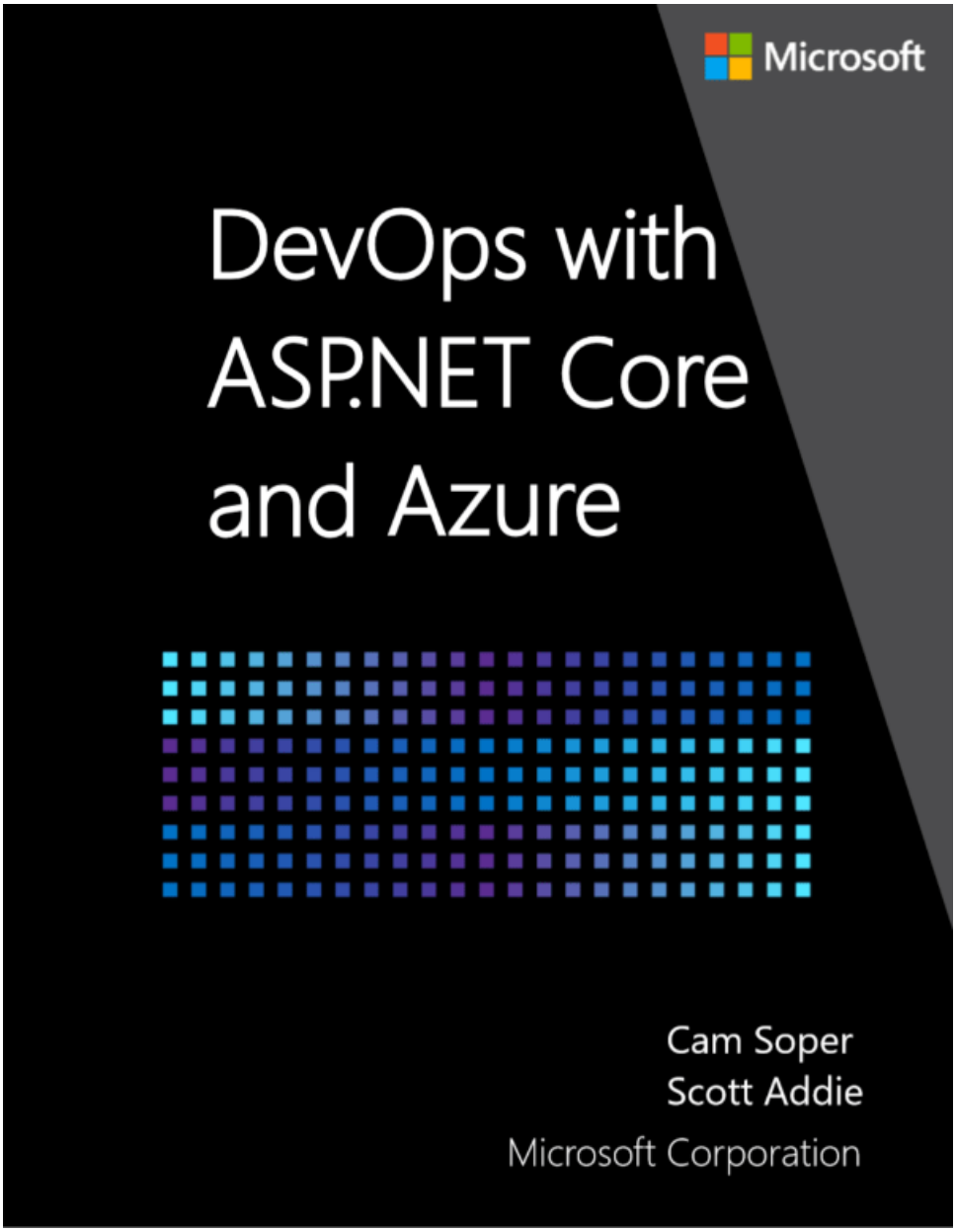
The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

DevOps with ASP.NET Core and Azure

9/22/2020 • 2 minutes to read • Edit Online



By Cam Soper and Scott Addie

This guide is available as a downloadable PDF e-book.

Welcome

Welcome to the Azure Development Lifecycle guide for .NET! This guide introduces the basic concepts of building a development lifecycle around Azure using .NET tools and processes. After finishing this guide, you'll reap the benefits of a mature DevOps toolchain.

Who this guide is for

You should be an experienced ASP.NET Core developer (200-300 level). You don't need to know anything about Azure, as we'll cover that in this introduction. This guide may also be useful for DevOps engineers who are more focused on operations than development.

This guide targets Windows developers. However, Linux and macOS are fully supported by .NET Core. To adapt this guide for Linux/macOS, watch for callouts for Linux/macOS differences.

What this guide doesn't cover

This guide is focused on an end-to-end continuous deployment experience for .NET developers. It's not an exhaustive guide to all things Azure, and it doesn't focus extensively on .NET APIs for Azure services. The emphasis is all around continuous integration, deployment, monitoring, and debugging. Near the end of the guide, recommendations for next steps are offered. Included in the suggestions are Azure platform services that are useful to ASP.NET Core developers.

What's in this guide

Tools and downloads

Learn where to acquire the tools used in this guide.

Deploy to App Service

Learn the various methods for deploying an ASP.NET Core app to Azure App Service.

Continuous integration and deployment

Build an end-to-end continuous integration and deployment solution for your ASP.NET Core app with GitHub, Azure DevOps Services, and Azure.

Monitor and debug

Use Azure's tools to monitor, troubleshoot, and tune your application.

Next steps

Other learning paths for the ASP.NET Core developer learning Azure.

Additional introductory reading

If this is your first exposure to cloud computing, these articles explain the basics.

- What is Cloud Computing?
- Examples of Cloud Computing
- What is IaaS?
- What is PaaS?

Tools and downloads

9/22/2020 • 2 minutes to read • Edit Online

Azure has several interfaces for provisioning and managing resources, such as the Azure portal, Azure CLI, Azure PowerShell, Azure Cloud Shell, and Visual Studio. This guide takes a minimalist approach and uses the Azure Cloud Shell whenever possible to reduce the steps required. However, the Azure portal must be used for some portions.

Prerequisites

The following subscriptions are required:

- Azure — If you don't have an account, get a free trial.
- Azure DevOps Services — your Azure DevOps subscription and organization is created in Chapter 4.
- GitHub — If you don't have an account, sign up for free.

The following tools are required:

- Git — A fundamental understanding of Git is recommended for this guide. Review the Git documentation, specifically git remote and git push.
- .NET Core SDK — Version 2.1.300 or later is required to build and run the sample app. If Visual Studio is installed with the **.NET Core cross-platform development** workload, the .NET Core SDK is already installed.

  Verify your .NET Core SDK installation. Open a command shell, and run the following command:

  ```
  dotnet --version
  ```

Recommended tools (Windows only)

- Visual Studio's robust Azure tools provide a GUI for most of the functionality described in this guide. Any edition of Visual Studio will work, including the free Visual Studio Community Edition. The tutorials are written to demonstrate development, deployment, and DevOps both with and without Visual Studio.

  Confirm that Visual Studio has the following workloads installed:

  - ASP.NET and web development
  - Azure development
  - .NET Core cross-platform development

Deploy an app to App Service

9/22/2020 • 7 minutes to read • Edit Online

Azure App Service is Azure's web hosting platform. Deploying a web app to Azure App Service can be done manually or by an automated process. This section of the guide discusses deployment methods that can be triggered manually or by script using the command line, or triggered manually using Visual Studio.

In this section, you'll accomplish the following tasks:

- Download and build the sample app.
- Create an Azure App Service Web App using the Azure Cloud Shell.
- Deploy the sample app to Azure using Git.
- Deploy a change to the app using Visual Studio.
- Add a staging slot to the web app.
- Deploy an update to the staging slot.
- Swap the staging and production slots.

Download and test the app

The app used in this guide is a pre-built ASP.NET Core app, Simple Feed Reader. It's a Razor Pages app that uses the `Microsoft.SyndicationFeed.ReaderWriter` API to retrieve an RSS/Atom feed and display the news items in a list.

Feel free to review the code, but it's important to understand that there's nothing special about this app. It's just a simple ASP.NET Core app for illustrative purposes.

From a command shell, download the code, build the project, and run it as follows.

> *Note: Linux/macOS users should make appropriate changes for paths, e.g., using forward slash (`/`) rather than back slash (`\`).*

1. Clone the code to a folder on your local machine.

   ```
   git clone https://github.com/Azure-Samples/simple-feed-reader/
   ```

2. Change your working folder to the *simple-feed-reader* folder that was created.

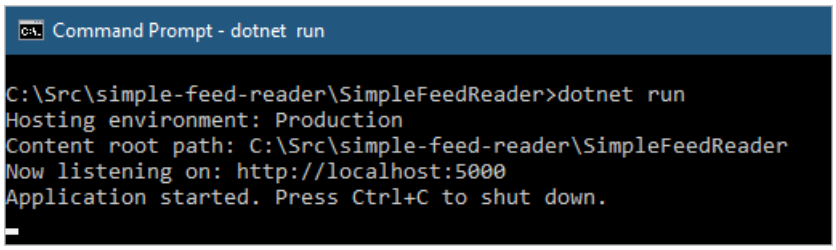
   ```
   cd .\simple-feed-reader\SimpleFeedReader
   ```

3. Restore the packages, and build the solution.

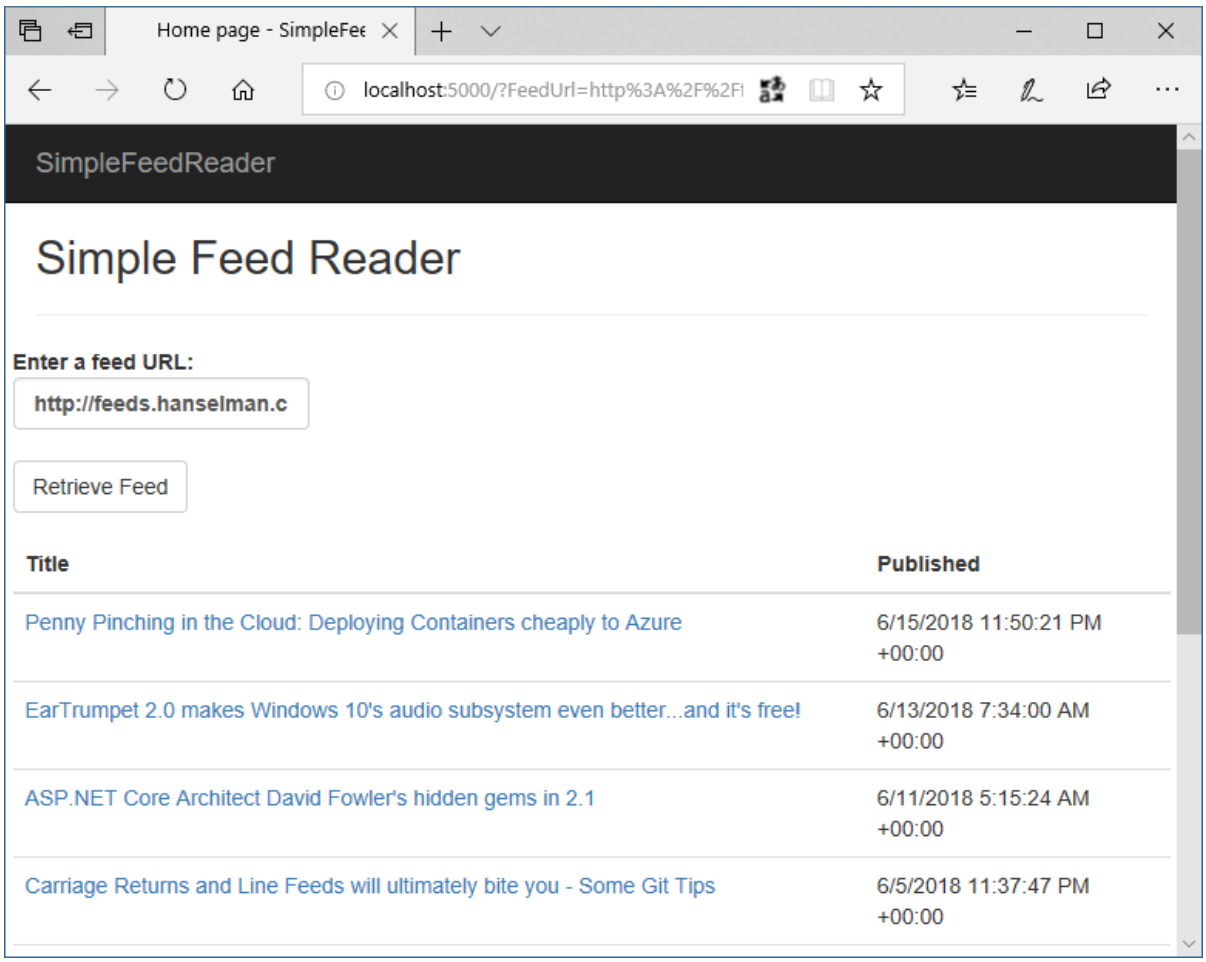
   ```
   dotnet build
   ```

4. Run the app.

   ```
   dotnet run
   ```



5. Open a browser and navigate to `http://localhost:5000`. The app allows you to type or paste a syndication feed URL and view a list of news items.



6. Once you're satisfied the app is working correctly, shut it down by pressing **Ctrl**+**C** in the command shell.

Create the Azure App Service Web App

To deploy the app, you'll need to create an App Service Web App. After creation of the Web App, you'll deploy to it from your local machine using Git.

1. Sign in to the Azure Cloud Shell. Note: When you sign in for the first time, Cloud Shell prompts to create a storage account for configuration files. Accept the defaults or provide a unique name.

2. Use the Cloud Shell for the following steps.

   a. Declare a variable to store your web app's name. The name must be unique to be used in the default URL. Using the `$RANDOM` Bash function to construct the name guarantees uniqueness and results in the format `webappname99999`.

   ```
   webappname=mywebapp$RANDOM
   ```

   b. Create a resource group. Resource groups provide a means to aggregate Azure resources to be managed

as a group.

```
az group create --location centralus --name AzureTutorial
```

The `az` command invokes the Azure CLI. The CLI can be run locally, but using it in the Cloud Shell saves time and configuration.

c. Create an App Service plan in the S1 tier. An App Service plan is a grouping of web apps that share the same pricing tier. The S1 tier isn't free, but it's required for the staging slots feature.

```
az appservice plan create --name $webappname --resource-group AzureTutorial --sku S1
```

d. Create the web app resource using the App Service plan in the same resource group.

```
az webapp create --name $webappname --resource-group AzureTutorial --plan $webappname
```

e. Set the deployment credentials. These deployment credentials apply to all the web apps in your subscription. Don't use special characters in the user name.

```
az webapp deployment user set --user-name REPLACE_WITH_USER_NAME --password REPLACE_WITH_PASSWORD
```

f. Configure the web app to accept deployments from local Git and display the *Git deployment URL*. **Note this URL for reference later**.

```
echo Git deployment URL: $(az webapp deployment source config-local-git --name $webappname --resource-
group AzureTutorial --query url --output tsv)
```

g. Display the *web app URL*. Browse to this URL to see the blank web app. **Note this URL for reference later**.

```
echo Web app URL: http://$webappname.azurewebsites.net
```

3. Using a command shell on your local machine, navigate to the web app's project folder (for example, `.\simple-feed-reader\SimpleFeedReader`). Execute the following commands to set up Git to push to the deployment URL:

   a. Add the remote URL to the local repository.

   ```
   git remote add azure-prod GIT_DEPLOYMENT_URL
   ```

   b. Push the local *master* branch to the *azure-prod* remote's *master* branch.

   ```
   git push azure-prod master
   ```

   You'll be prompted for the deployment credentials you created earlier. Observe the output in the command shell. Azure builds the ASP.NET Core app remotely.

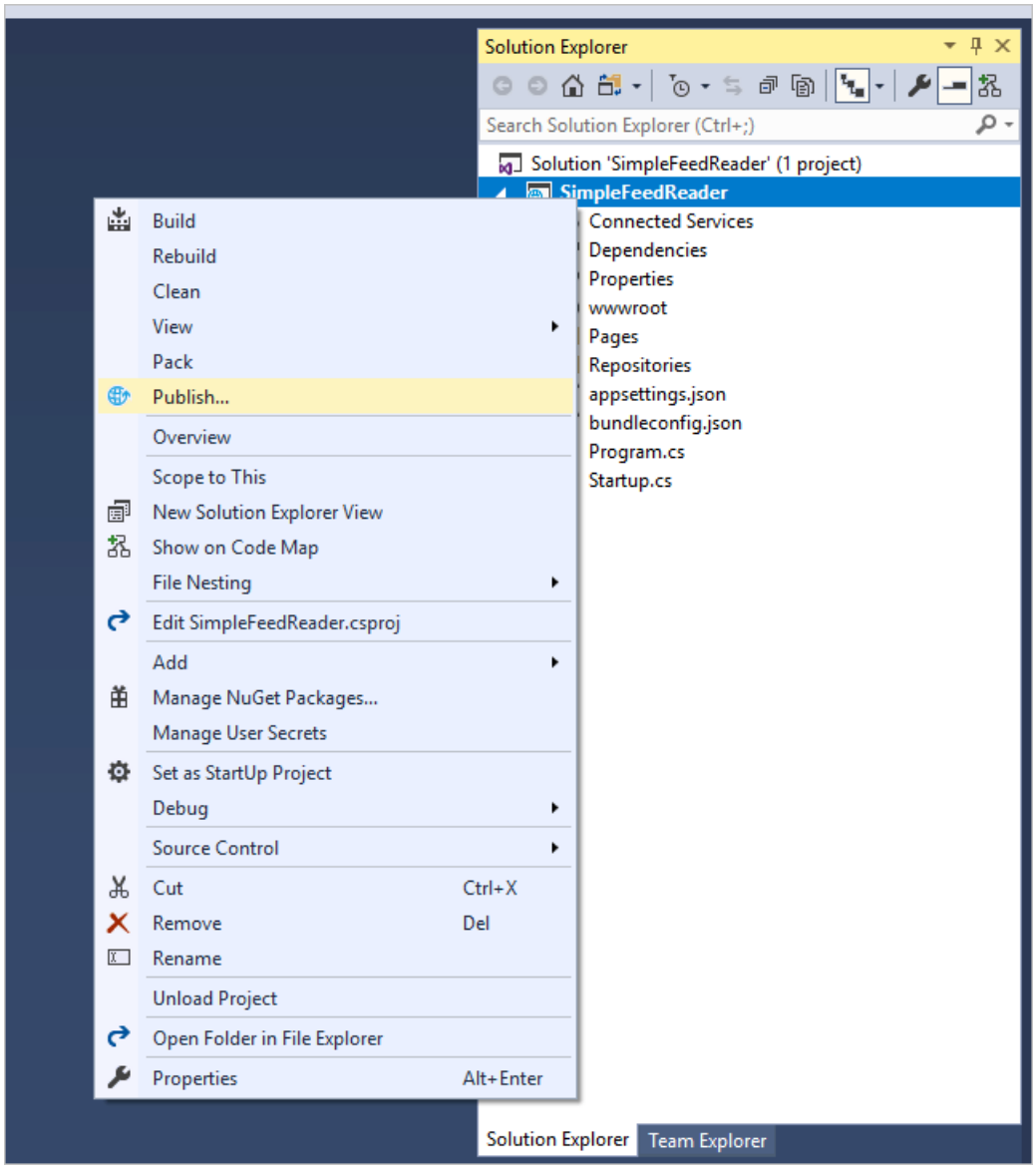
4. In a browser, navigate to the *Web app URL* and note the app has been built and deployed. Additional changes can be committed to the local Git repository with `git commit`. These changes are pushed to Azure with the preceding `git push` command.

Deployment with Visual Studio

> *Note: This section applies to Windows only. Linux and macOS users should make the change described in step 2 below. Save the file, and commit the change to the local repository with* `git commit`*. Finally, push the change with* `git push`*, as in the first section.*

The app has already been deployed from the command shell. Let's use Visual Studio's integrated tools to deploy an update to the app. Behind the scenes, Visual Studio accomplishes the same thing as the command line tooling, but within Visual Studio's familiar UI.

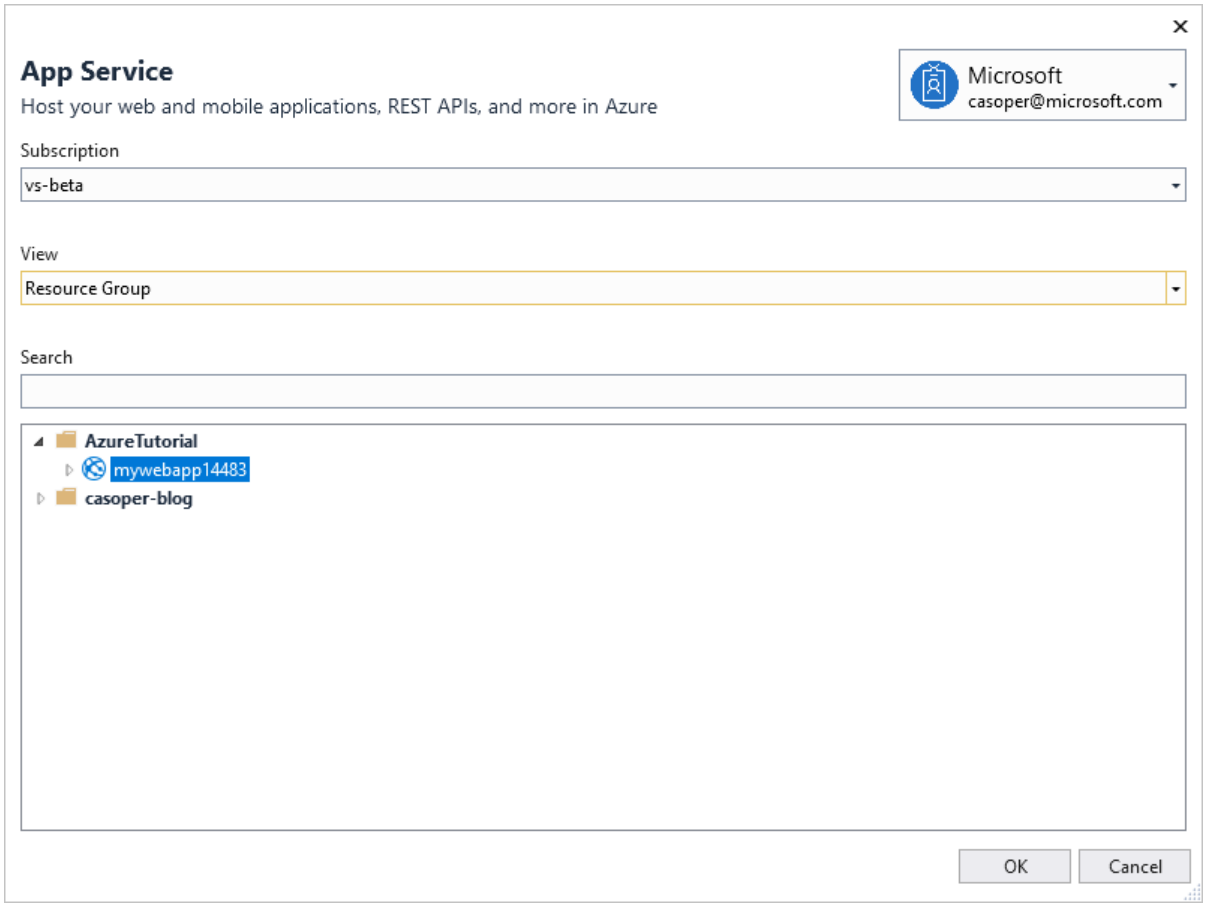
1. Open *SimpleFeedReader.sln* in Visual Studio.
2. In Solution Explorer, open *Pages\Index.cshtml*. Change `<h2>Simple Feed Reader</h2>` to `<h2>Simple Feed Reader - V2</h2>`.
3. Press **Ctrl**+**Shift**+**B** to build the app.
4. In Solution Explorer, right-click on the project and click **Publish**.



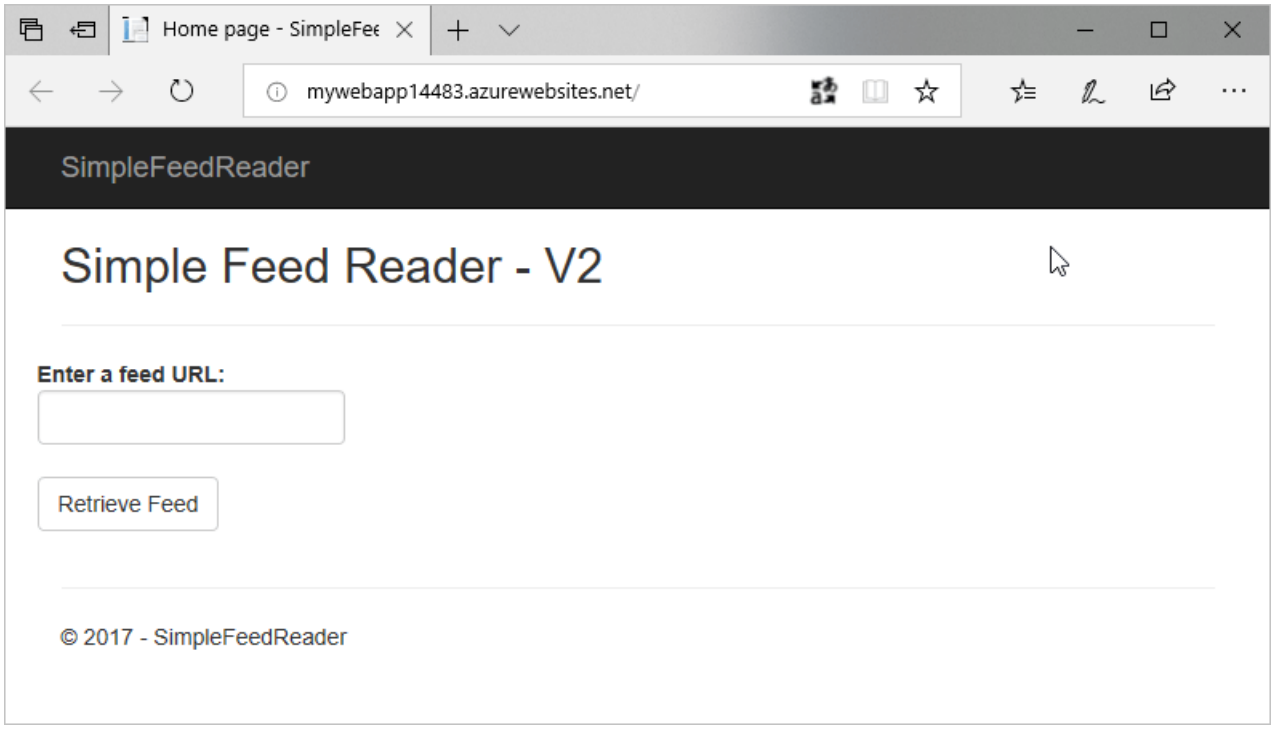
5. Visual Studio can create a new App Service resource, but this update will be published over the existing deployment. In the **Pick a publish target** dialog, select **App Service** from the list on the left, and then select **Select Existing**. Click **Publish**.
6. In the **App Service** dialog, confirm that the Microsoft or Organizational account used to create your Azure

subscription is displayed in the upper right. If it's not, click the drop-down and add it.

7. Confirm that the correct Azure **Subscription** is selected. For **View**, select **Resource Group**. Expand the **AzureTutorial** resource group and then select the existing web app. Click **OK**.



Visual Studio builds and deploys the app to Azure. Browse to the web app URL. Validate that the `<h2>` element modification is live.



Deployment slots

Deployment slots support the staging of changes without impacting the app running in production. Once the

staged version of the app is validated by a quality assurance team, the production and staging slots can be swapped. The app in staging is promoted to production in this manner. The following steps create a staging slot, deploy some changes to it, and swap the staging slot with production after verification.

1. Sign in to the Azure Cloud Shell, if not already signed in.

2. Create the staging slot.

   a. Create a deployment slot with the name *staging*.

   ```
   az webapp deployment slot create --name $webappname --resource-group AzureTutorial --slot staging
   ```

   b. Configure the staging slot to use deployment from local Git and get the **staging** deployment URL. **Note this URL for reference later**.

   ```
   echo Git deployment URL for staging: $(az webapp deployment source config-local-git --name $webappname --resource-group AzureTutorial --slot staging --query url --output tsv)
   ```

   c. Display the staging slot's URL. Browse to the URL to see the empty staging slot. **Note this URL for reference later**.

   ```
   echo Staging web app URL: http://$webappname-staging.azurewebsites.net
   ```

3. In a text editor or Visual Studio, modify *Pages/Index.cshtml* again so that the `<h2>` element reads `<h2>Simple Feed Reader - V3</h2>` and save the file.

4. Commit the file to the local Git repository, using either the **Changes** page in Visual Studio's *Team Explorer* tab, or by entering the following using the local machine's command shell:

   ```
   git commit -a -m "upgraded to V3"
   ```

5. Using the local machine's command shell, add the staging deployment URL as a Git remote and push the committed changes:

   a. Add the remote URL for staging to the local Git repository.

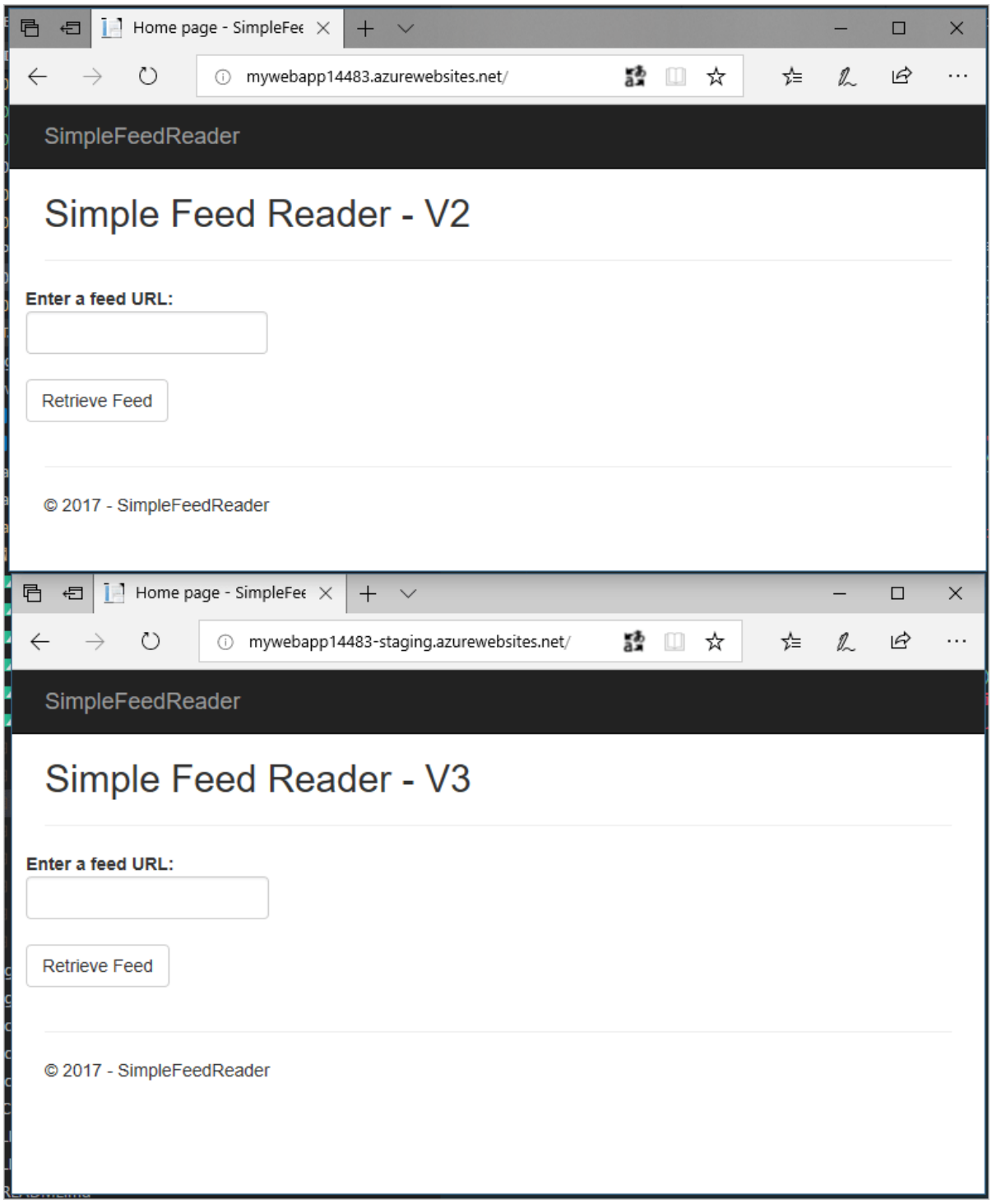
   ```
   git remote add azure-staging <Git_staging_deployment_URL>
   ```

   b. Push the local *master* branch to the *azure-staging* remote's *master* branch.

   ```
   git push azure-staging master
   ```

   Wait while Azure builds and deploys the app.

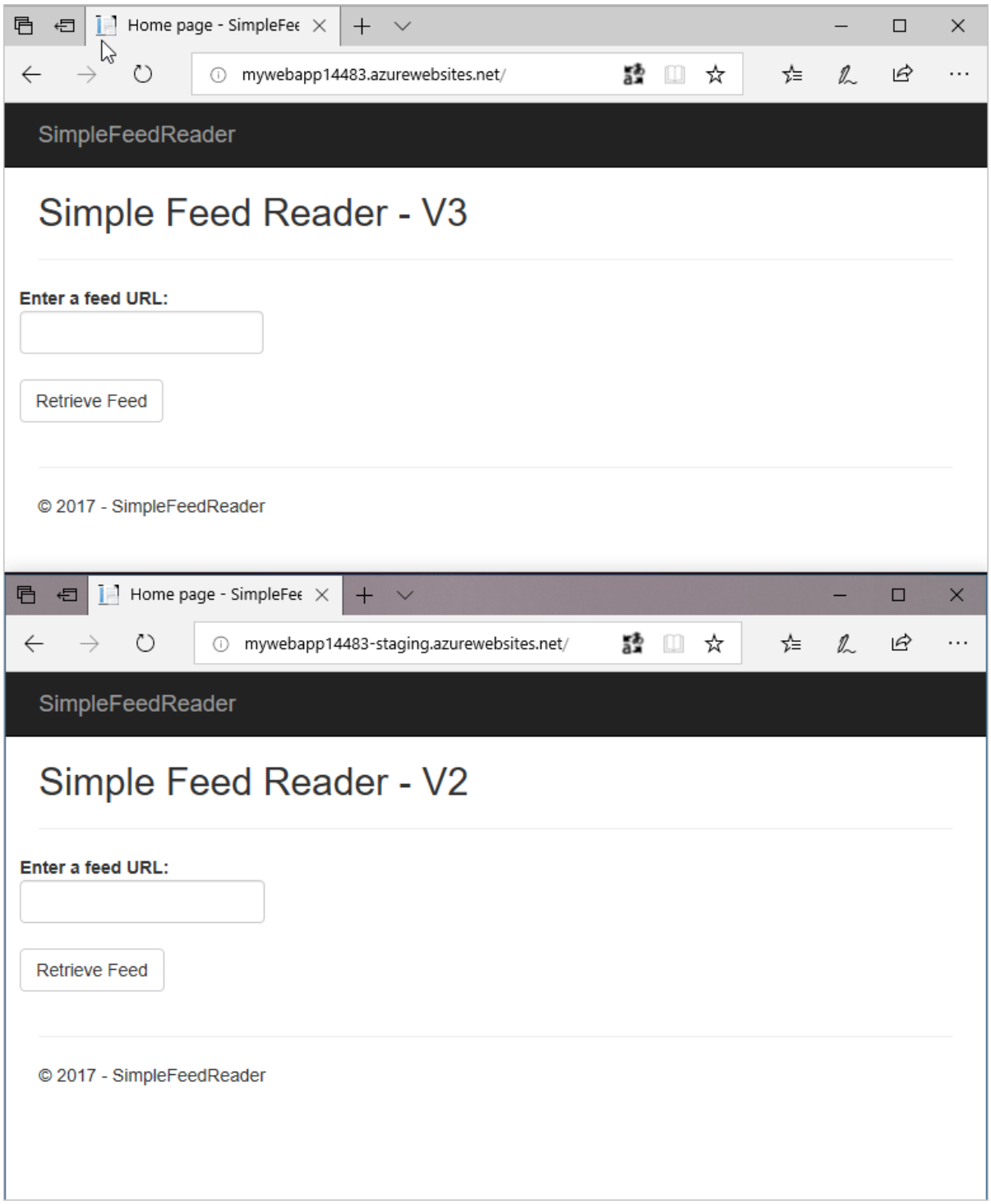
6. To verify that V3 has been deployed to the staging slot, open two browser windows. In one window, navigate to the original web app URL. In the other window, navigate to the staging web app URL. The production URL serves V2 of the app. The staging URL serves V3 of the app.



7. In the Cloud Shell, swap the verified/warmed-up staging slot into production.

```
az webapp deployment slot swap --name $webappname --resource-group AzureTutorial --slot staging
```

8. Verify that the swap occurred by refreshing the two browser windows.



Summary

In this section, the following tasks were completed:

- Downloaded and built the sample app.
- Created an Azure App Service Web App using the Azure Cloud Shell.
- Deployed the sample app to Azure using Git.
- Deployed a change to the app using Visual Studio.
- Added a staging slot to the web app.
- Deployed an update to the staging slot.
- Swapped the staging and production slots.

In the next section, you'll learn how to build a DevOps pipeline with Azure Pipelines.

Additional reading

- Web Apps overview
- Build a .NET Core and SQL Database web app in Azure App Service
- Configure deployment credentials for Azure App Service
- Set up staging environments in Azure App Service

Continuous integration and deployment

9/22/2020 • 10 minutes to read • Edit Online

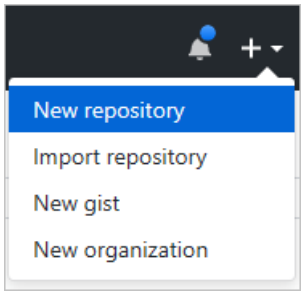
In the previous chapter, you created a local Git repository for the Simple Feed Reader app. In this chapter, you'll publish that code to a GitHub repository and construct an Azure DevOps Services pipeline using Azure Pipelines. The pipeline enables continuous builds and deployments of the app. Any commit to the GitHub repository triggers a build and a deployment to the Azure Web App's staging slot.

In this section, you'll complete the following tasks:

- Publish the app's code to GitHub
- Disconnect local Git deployment
- Create an Azure DevOps organization
- Create a team project in Azure DevOps Services
- Create a build definition
- Create a release pipeline
- Commit changes to GitHub and automatically deploy to Azure
- Examine the Azure Pipelines pipeline

Publish the app's code to GitHub

1. Open a browser window, and navigate to `https://github.com`.
2. Click the **+** drop-down in the header, and select **New repository**:



3. Select your account in the **Owner** drop-down, and enter *simple-feed-reader* in the **Repository name** textbox.
4. Click the **Create repository** button.
5. Open your local machine's command shell. Navigate to the directory in which the *simple-feed-reader* Git repository is stored.
6. Rename the existing *origin* remote to *upstream*. Execute the following command:

```
git remote rename origin upstream
```

7. Add a new *origin* remote pointing to your copy of the repository on GitHub. Execute the following command:

```
git remote add origin https://github.com/<GitHub_username>/simple-feed-reader/
```

8. Publish your local Git repository to the newly created GitHub repository. Execute the following command:

```
git push -u origin master
```

9. Open a browser window, and navigate to `https://github.com/<GitHub_username>/simple-feed-reader/`. Validate that your code appears in the GitHub repository.

Disconnect local Git deployment

Remove the local Git deployment with the following steps. Azure Pipelines (an Azure DevOps service) both replaces and augments that functionality.

1. Open the Azure portal, and navigate to the *staging (mywebapp<unique_number>/staging)* Web App. The Web App can be quickly located by entering *staging* in the portal's search box:

2. Click **Deployment Center**. A new panel appears. Click **Disconnect** to remove the local Git source control configuration that was added in the previous chapter. Confirm the removal operation by clicking the **Yes** button.

3. Navigate to the *mywebapp<unique_number>* App Service. As a reminder, the portal's search box can be used to quickly locate the App Service.

4. Click **Deployment Center**. A new panel appears. Click **Disconnect** to remove the local Git source control configuration that was added in the previous chapter. Confirm the removal operation by clicking the **Yes** button.

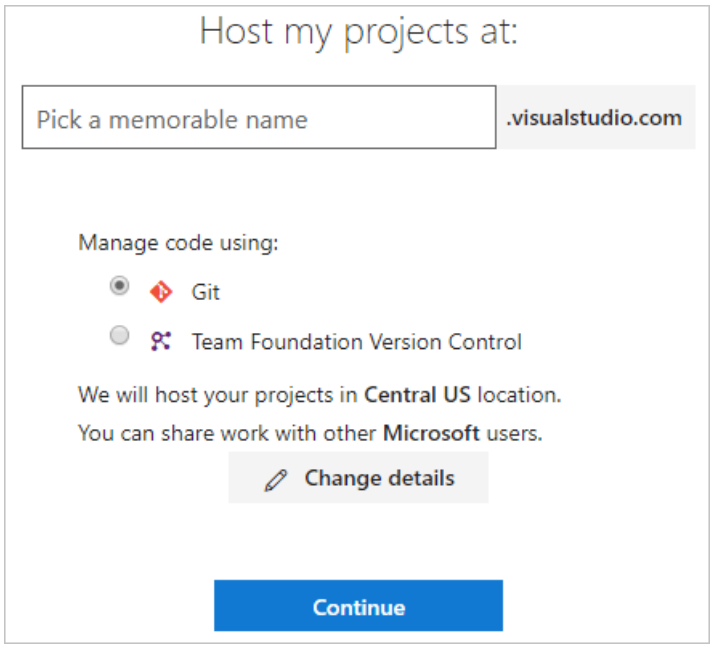
Create an Azure DevOps organization

1. Open a browser, and navigate to the Azure DevOps organization creation page.

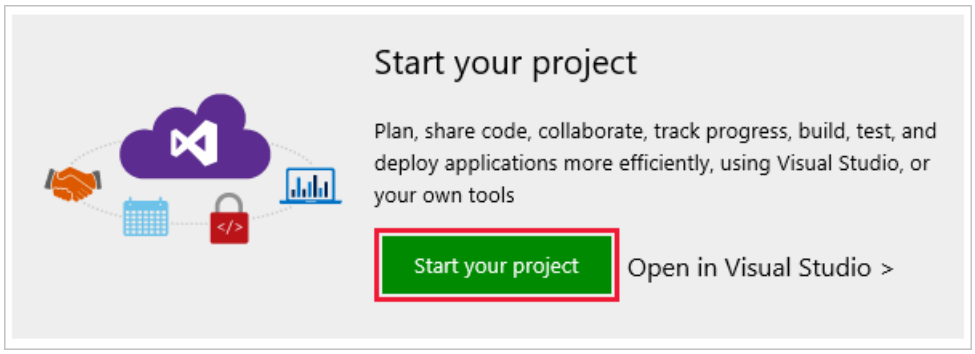
2. Type a unique name into the **Pick a memorable name** textbox to form the URL for accessing your Azure DevOps organization.

3. Select the **Git** radio button, since the code is hosted in a GitHub repository.

4. Click the **Continue** button. After a short wait, an account and a team project, named *MyFirstProject*, are created.



5. Open the confirmation email indicating that the Azure DevOps organization and project are ready for use. Click the **Start your project** button:



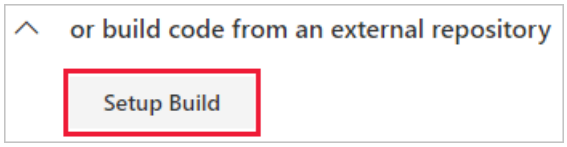
6. A browser opens to *<account_name>.visualstudio.com*. Click the *MyFirstProject* link to begin configuring the project's DevOps pipeline.

Configure the Azure Pipelines pipeline

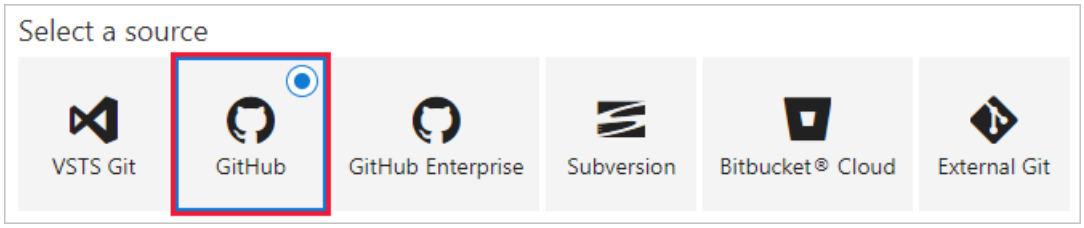
There are three distinct steps to complete. Completing the steps in the following three sections results in an operational DevOps pipeline.

Grant Azure DevOps access to the GitHub repository

1. Expand the **or build code from an external repository** accordion. Click the **Setup Build** button:

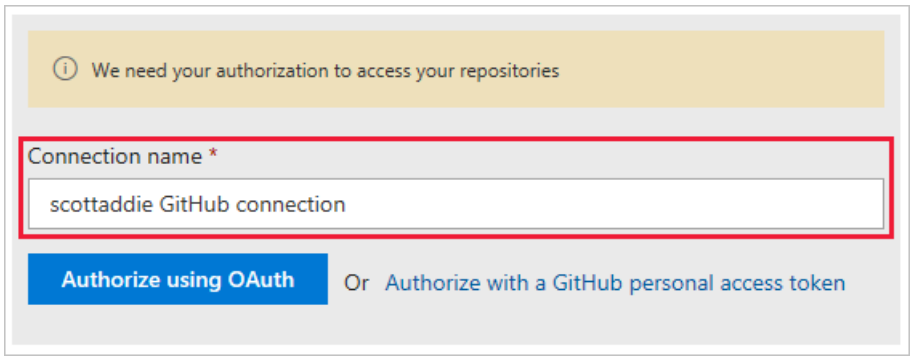


2. Select the **GitHub** option from the **Select a source** section:



3. Authorization is required before Azure DevOps can access your GitHub repository. Enter

<GitHub_username> GitHub connection in the **Connection name** textbox. For example:



4. If two-factor authentication is enabled on your GitHub account, a personal access token is required. In that case, click the **Authorize with a GitHub personal access token** link. See the official GitHub personal access token creation instructions for help. Only the *repo* scope of permissions is needed. Otherwise, click the **Authorize using OAuth** button.

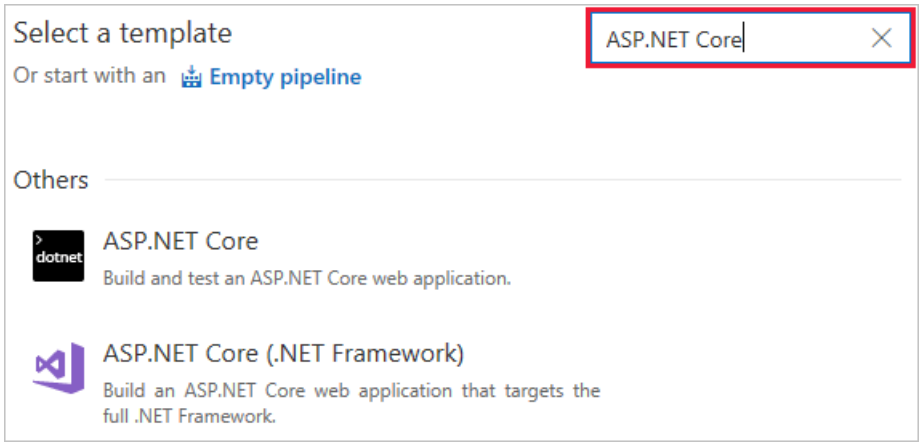
5. When prompted, sign in to your GitHub account. Then select Authorize to grant access to your Azure DevOps organization. If successful, a new service endpoint is created.

6. Click the ellipsis button next to the **Repository** button. Select the *<GitHub_username>/simple-feed-reader* repository from the list. Click the **Select** button.

7. Select the *master* branch from the **Default branch for manual and scheduled builds** drop-down. Click the **Continue** button. The template selection page appears.

Create the build definition

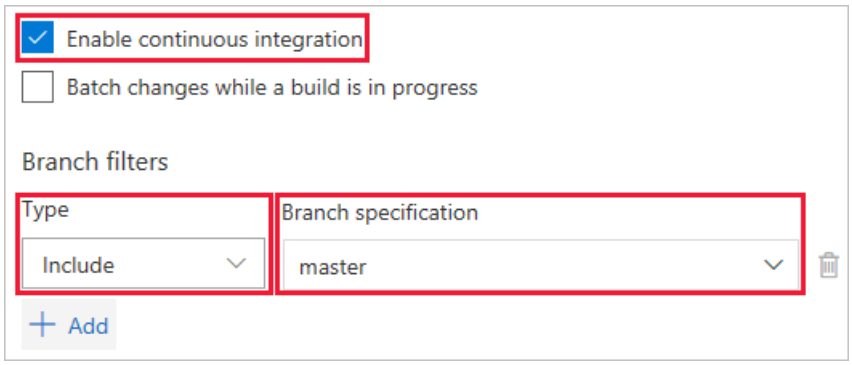
1. From the template selection page, enter *ASP.NET Core* in the search box:



2. The template search results appear. Hover over the **ASP.NET Core** template, and click the **Apply** button.

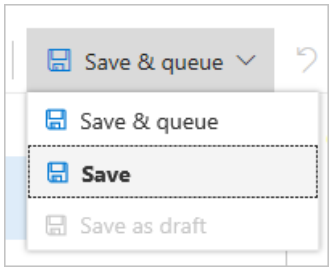
3. The **Tasks** tab of the build definition appears. Click the **Triggers** tab.

4. Check the **Enable continuous integration** box. Under the **Branch filters** section, confirm that the **Type** drop-down is set to *Include*. Set the **Branch specification** drop-down to *master*.

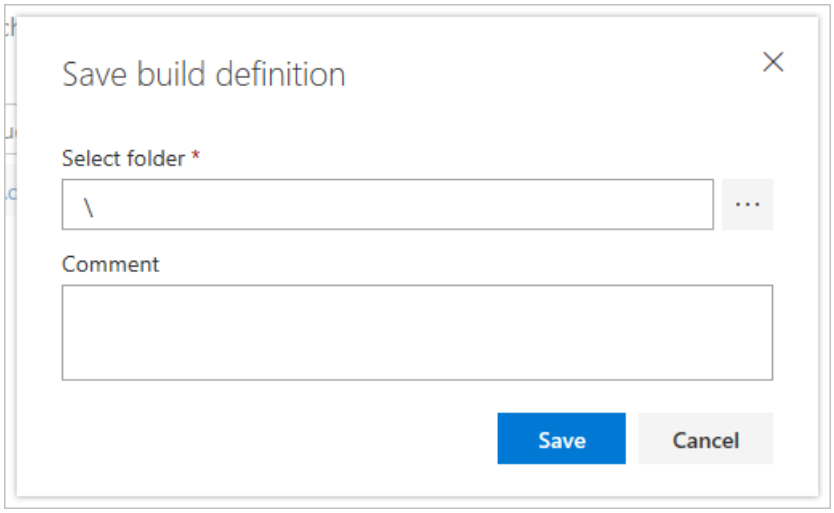


These settings cause a build to trigger when any change is pushed to the *master* branch of the GitHub repository. Continuous integration is tested in the Commit changes to GitHub and automatically deploy to Azure section.

5. Click the **Save & queue** button, and select the **Save** option:



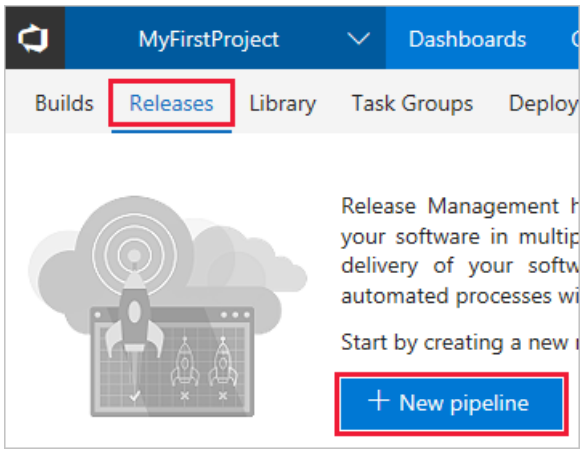
6. The following modal dialog appears:



Use the default folder of \, and click the **Save** button.

Create the release pipeline

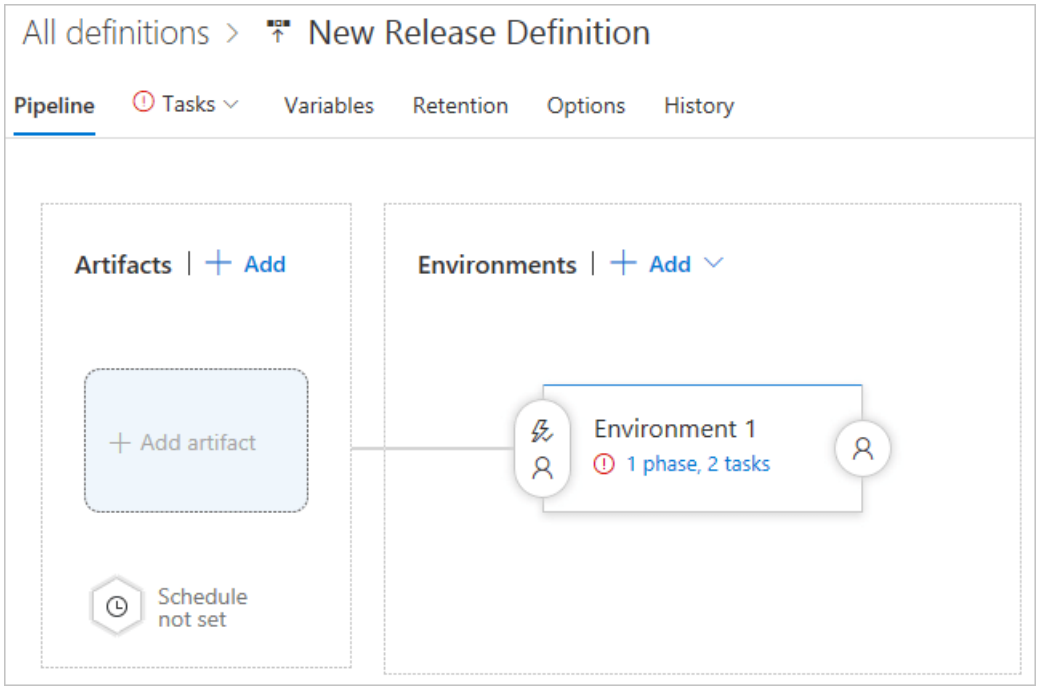
1. Click the **Releases** tab of your team project. Click the **New pipeline** button.



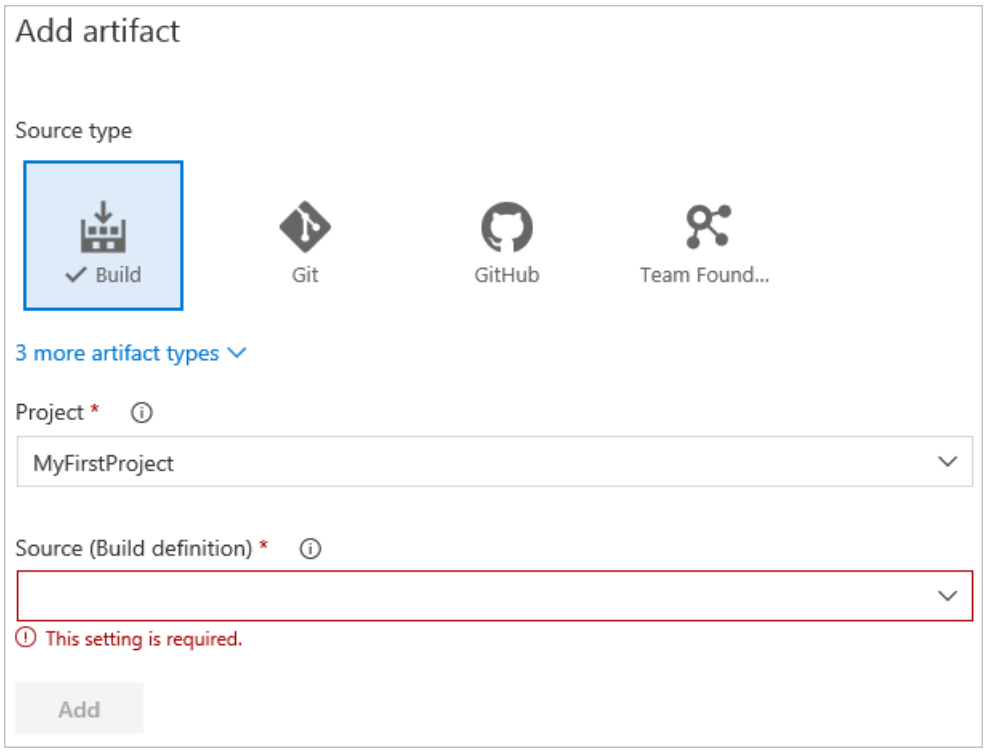
The template selection pane appears.

2. From the template selection page, enter *App Service* in the search box:

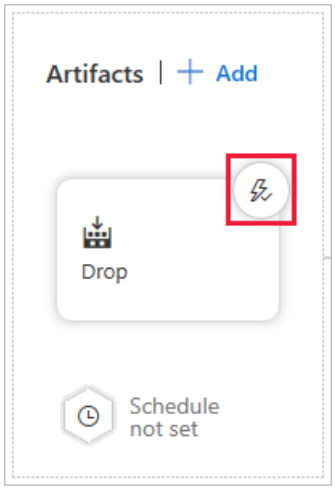
3. The template search results appear. Hover over the **Azure App Service Deployment with Slot** template, and click the **Apply** button. The **Pipeline** tab of the release pipeline appears.



4. Click the **Add** button in the **Artifacts** box. The **Add artifact** panel appears:



5. Select the **Build** tile from the **Source type** section. This type allows for the linking of the release pipeline to the build definition.
6. Select *MyFirstProject* from the **Project** drop-down.
7. Select the build definition name, *MyFirstProject-ASP.NET Core-CI*, from the **Source (Build definition)** drop-down.
8. Select *Latest* from the **Default version** drop-down. This option builds the artifacts produced by the latest run of the build definition.
9. Replace the text in the **Source alias** textbox with *Drop*.
10. Click the **Add** button. The **Artifacts** section updates to display the changes.
11. Click the lightning bolt icon to enable continuous deployments:



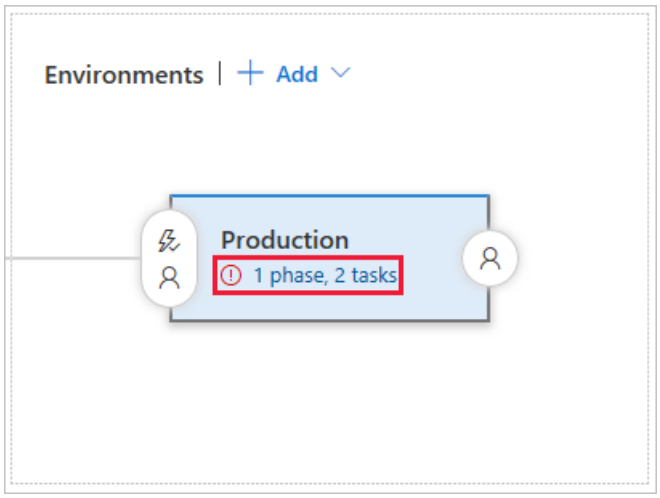
With this option enabled, a deployment occurs each time a new build is available.

12. A **Continuous deployment trigger** panel appears to the right. Click the toggle button to enable the feature. It isn't necessary to enable the **Pull request trigger**.
13. Click the **Add** drop-down in the **Build branch filters** section. Choose the **Build Definition's default branch** option. This filter causes the release to trigger only for a build from the GitHub repository's *master*

branch.

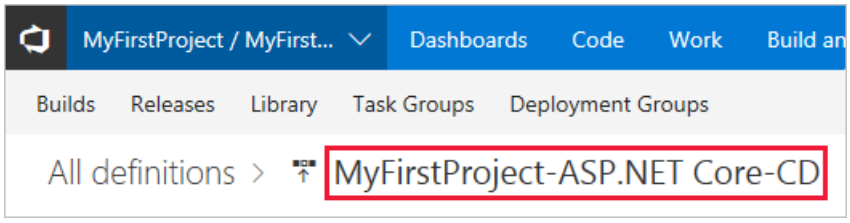
14. Click the **Save** button. Click the **OK** button in the resulting **Save** modal dialog.
15. Click the **Environment 1** box. An **Environment** panel appears to the right. Change the *Environment 1* text in the **Environment name** textbox to *Production*.

16. Click the **1 phase, 2 tasks** link in the **Production** box:



The **Tasks** tab of the environment appears.

17. Click the **Deploy Azure App Service to Slot** task. Its settings appear in a panel to the right.
18. Select the Azure subscription associated with the App Service from the **Azure subscription** drop-down. Once selected, click the **Authorize** button.
19. Select *Web App* from the **App type** drop-down.
20. Select *mywebapp/<unique_number/>* from the **App service name** drop-down.
21. Select *AzureTutorial* from the **Resource group** drop-down.
22. Select *staging* from the **Slot** drop-down.
23. Click the **Save** button.
24. Hover over the default release pipeline name. Click the pencil icon to edit it. Use *MyFirstProject-ASP.NET Core-CD* as the name.



25. Click the **Save** button.

Commit changes to GitHub and automatically deploy to Azure

1. Open *SimpleFeedReader.sln* in Visual Studio.

2. In Solution Explorer, open *Pages\Index.cshtml*. Change `<h2>Simple Feed Reader - V3</h2>` to `<h2>Simple Feed Reader - V4</h2>`.

3. Press **Ctrl**+**Shift**+**B** to build the app.

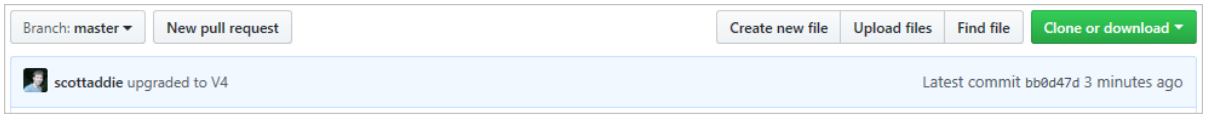
4. Commit the file to the GitHub repository. Use either the **Changes** page in Visual Studio's *Team Explorer* tab, or execute the following using the local machine's command shell:

   ```
   git commit -a -m "upgraded to V4"
   ```

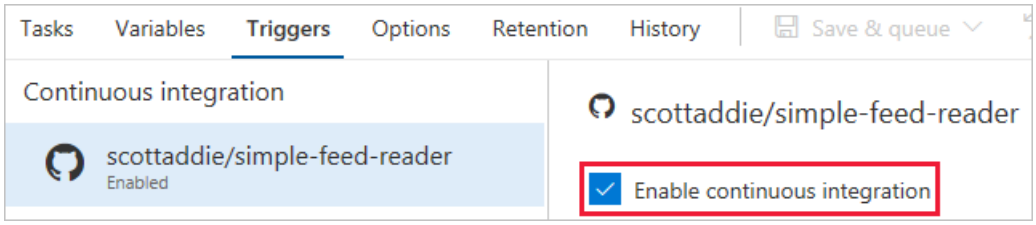
5. Push the change in the *master* branch to the *origin* remote of your GitHub repository:

   ```
   git push origin master
   ```

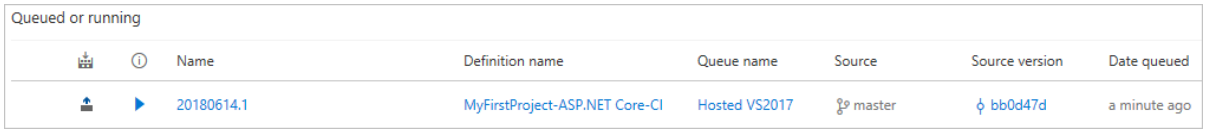
   The commit appears in the GitHub repository's *master* branch:

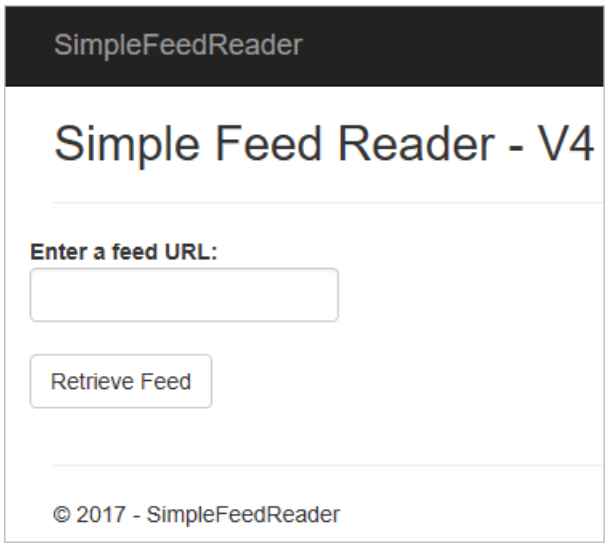
   The build is triggered, since continuous integration is enabled in the build definition's **Triggers** tab:

6. Navigate to the **Queued** tab of the **Azure Pipelines** > **Builds** page in Azure DevOps Services. The queued build shows the branch and commit that triggered the build:

7. Once the build succeeds, a deployment to Azure occurs. Navigate to the app in the browser. Notice that the "V4" text appears in the heading:

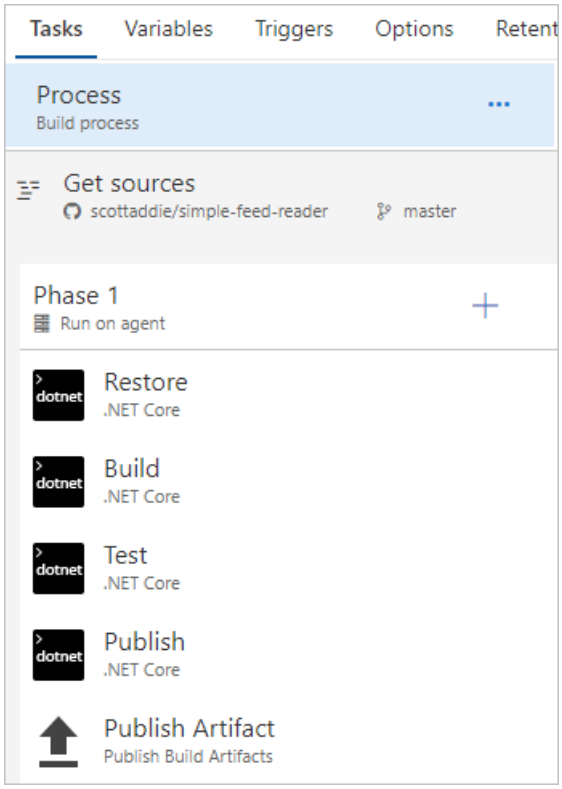


Examine the Azure Pipelines pipeline

Build definition

A build definition was created with the name *MyFirstProject-ASP.NET Core-CI*. Upon completion, the build produces a *.zip* file including the assets to be published. The release pipeline deploys those assets to Azure.

The build definition's **Tasks** tab lists the individual steps being used. There are five build tasks.



1. **Restore** — Executes the `dotnet restore` command to restore the app's NuGet packages. The default package feed used is nuget.org.
2. **Build** — Executes the `dotnet build --configuration release` command to compile the app's code. This `--configuration` option is used to produce an optimized version of the code, which is suitable for deployment to a production environment. Modify the *BuildConfiguration* variable on the build definition's **Variables** tab if, for example, a debug configuration is needed.
3. **Test** — Executes the `dotnet test --configuration release --logger trx --results-directory <local_path_on_build_agent>`

command to run the app's unit tests. Unit tests are executed within any C# project matching the `**/*Tests/*.csproj` glob pattern. Test results are saved in a *.trx* file at the location specified by the `--results-directory` option. If any tests fail, the build fails and isn't deployed.

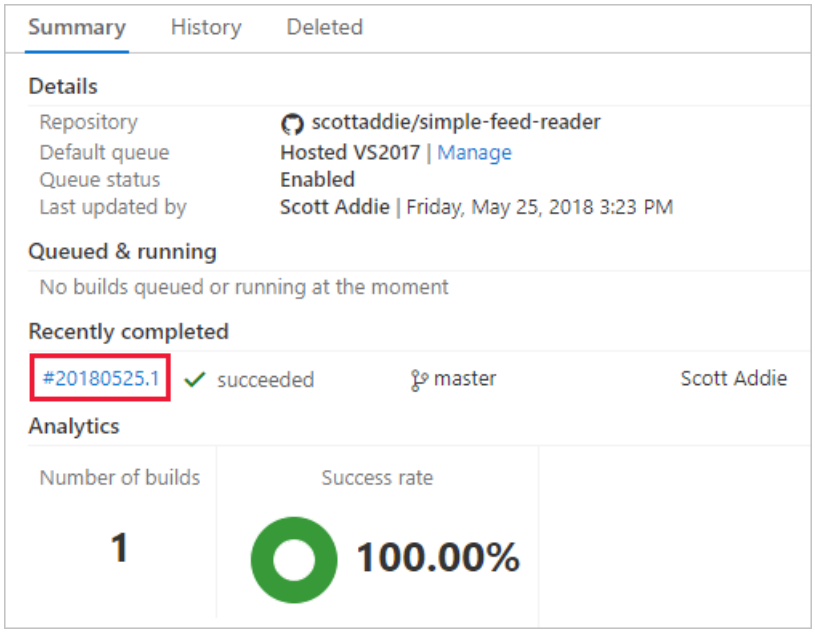
> **NOTE**
>
> To verify the unit tests work, modify *SimpleFeedReader.Tests\Services\NewsServiceTests.cs* to purposefully break one of the tests. For example, change `Assert.True(result.Count > 0);` to `Assert.False(result.Count > 0);` in the `Returns_News_Stories_Given_Valid_Uri` method. Commit and push the change to GitHub. The build is triggered and fails. The build pipeline status changes to **failed**. Revert the change, commit, and push again. The build succeeds.

4. **Publish** — Executes the `dotnet publish --configuration release --output <local_path_on_build_agent>` command to produce a *.zip* file with the artifacts to be deployed. The `--output` option specifies the publish location of the *.zip* file. That location is specified by passing a predefined variable named `$(build.artifactstagingdirectory)`. That variable expands to a local path, such as *c:\agent_work\1\a*, on the build agent.

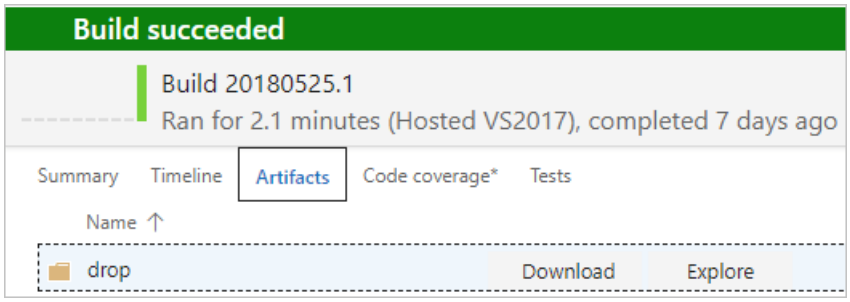
5. **Publish Artifact** — Publishes the *.zip* file produced by the **Publish** task. The task accepts the *.zip* file location as a parameter, which is the predefined variable `$(build.artifactstagingdirectory)`. The *.zip* file is published as a folder named *drop*.

Click the build definition's **Summary** link to view a history of builds with the definition:

On the resulting page, click the link corresponding to the unique build number:



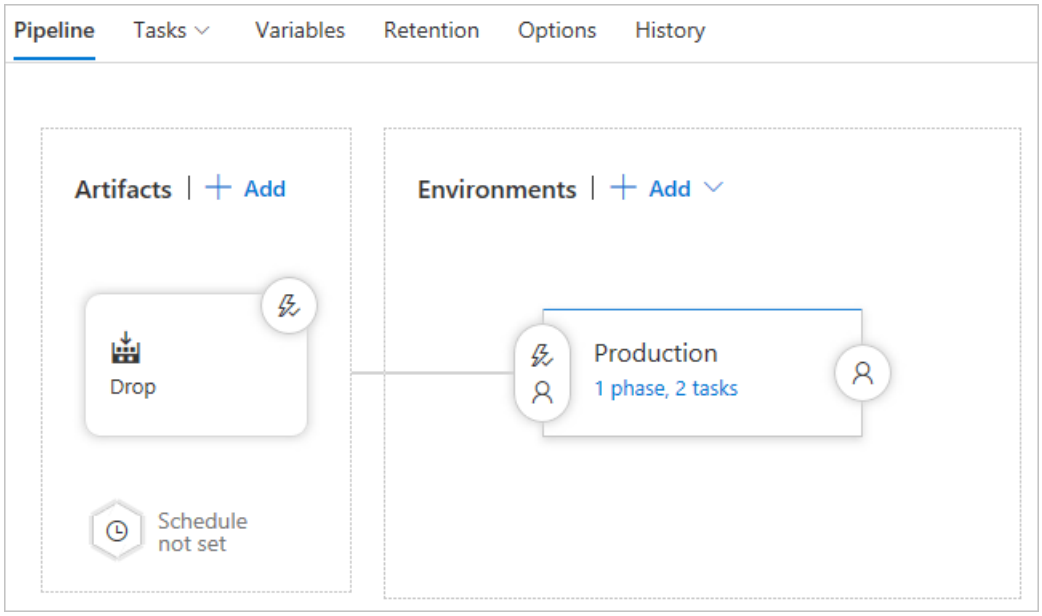
A summary of this specific build is displayed. Click the **Artifacts** tab, and notice the *drop* folder produced by the build is listed:



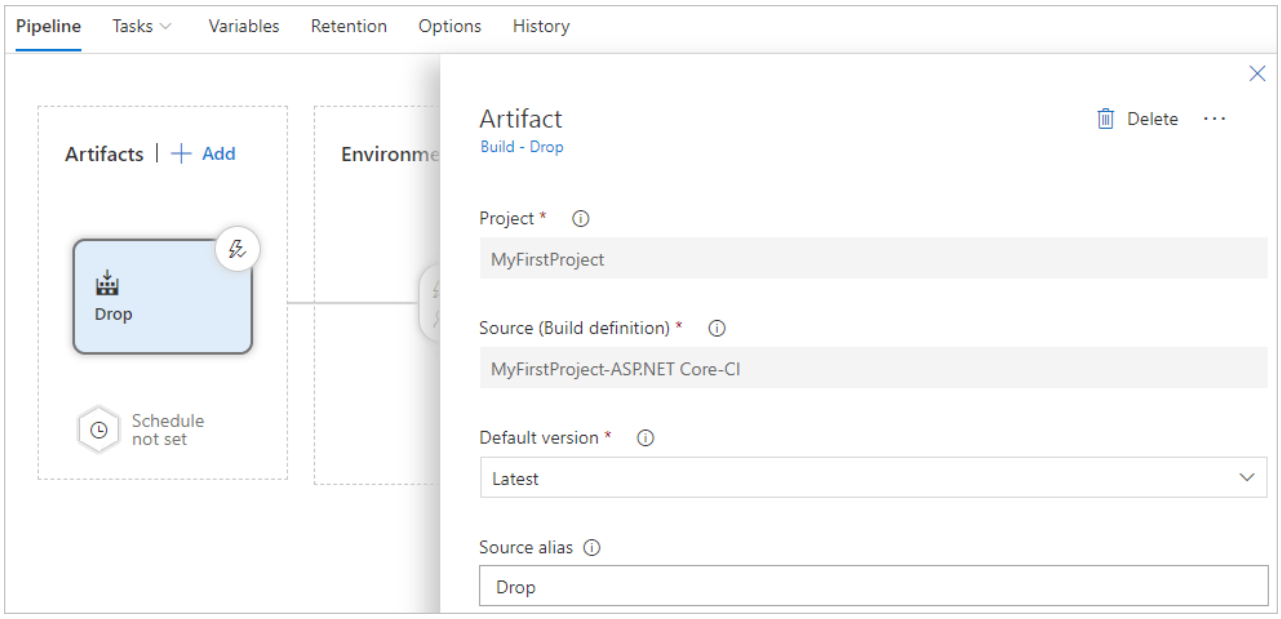
Use the **Download** and **Explore** links to inspect the published artifacts.

Release pipeline

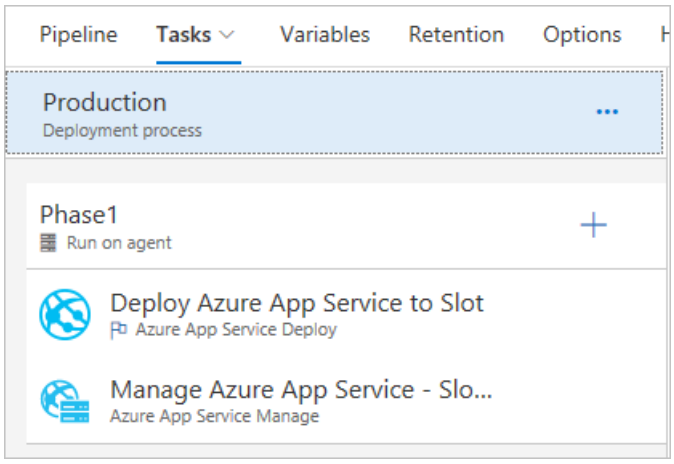
A release pipeline was created with the name *MyFirstProject-ASP.NET Core-CD*.



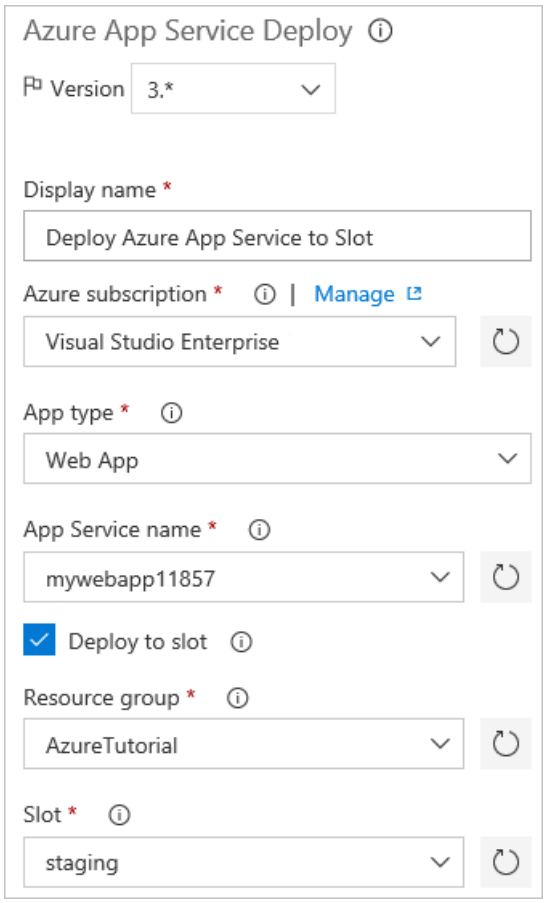
The two major components of the release pipeline are the **Artifacts** and the **Environments**. Clicking the box in the **Artifacts** section reveals the following panel:



The **Source (Build definition)** value represents the build definition to which this release pipeline is linked. The *.zip* file produced by a successful run of the build definition is provided to the *Production* environment for deployment to Azure. Click the *1 phase, 2 tasks* link in the *Production* environment box to view the release pipeline tasks:

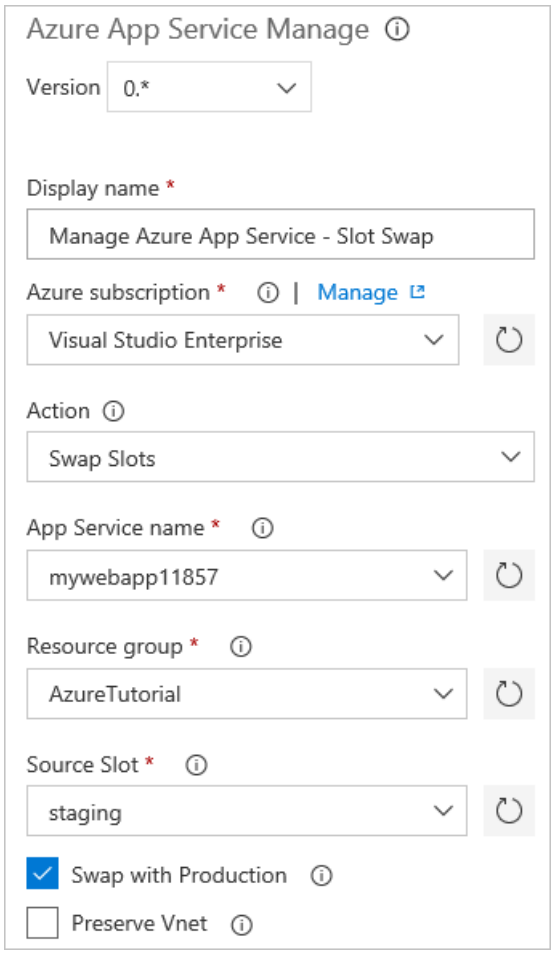


The release pipeline consists of two tasks: *Deploy Azure App Service to Slot* and *Manage Azure App Service - Slot Swap*. Clicking the first task reveals the following task configuration:



The Azure subscription, service type, web app name, resource group, and deployment slot are defined in the deployment task. The **Package or folder** textbox holds the *.zip* file path to be extracted and deployed to the *staging* slot of the *mywebapp<unique_number>* web app.

Clicking the slot swap task reveals the following task configuration:



The subscription, resource group, service type, web app name, and deployment slot details are provided. The **Swap with Production** check box is checked. Consequently, the bits deployed to the *staging* slot are swapped into the production environment.

Additional reading

- Create your first pipeline with Azure Pipelines
- Build and .NET Core project
- Deploy a web app with Azure Pipelines

Monitor and debug

9/22/2020 • 4 minutes to read • Edit Online

Having deployed the app and built a DevOps pipeline, it's important to understand how to monitor and troubleshoot the app.

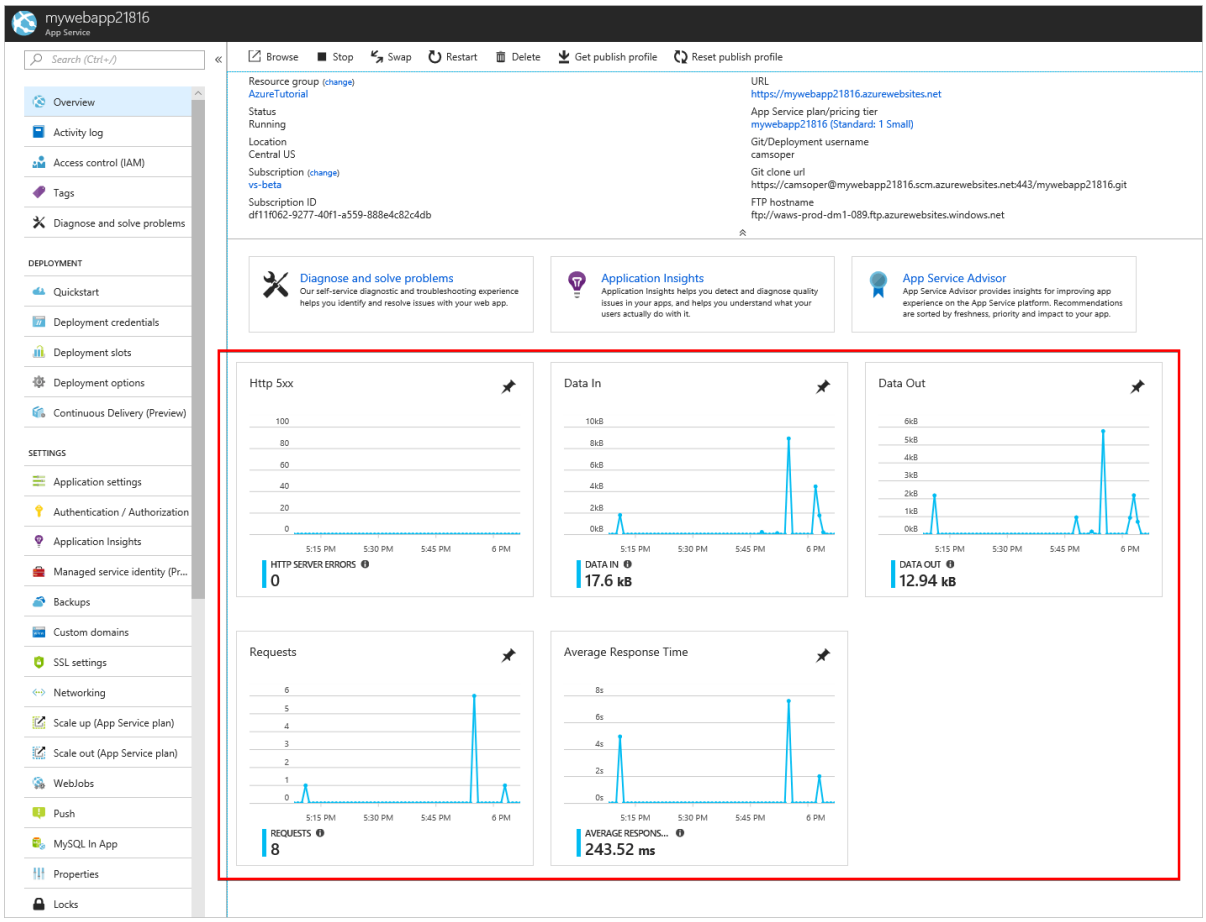
In this section, you'll complete the following tasks:

- Find basic monitoring and troubleshooting data in the Azure portal
- Learn how Azure Monitor provides a deeper look at metrics across all Azure services
- Connect the web app with Application Insights for app profiling
- Turn on logging and learn where to download logs
- Stream logs in real time
- Learn where to set up alerts
- Learn about remote debugging Azure App Service web apps.

Basic monitoring and troubleshooting

App Service web apps are easily monitored in real time. The Azure portal renders metrics in easy-to-understand charts and graphs.

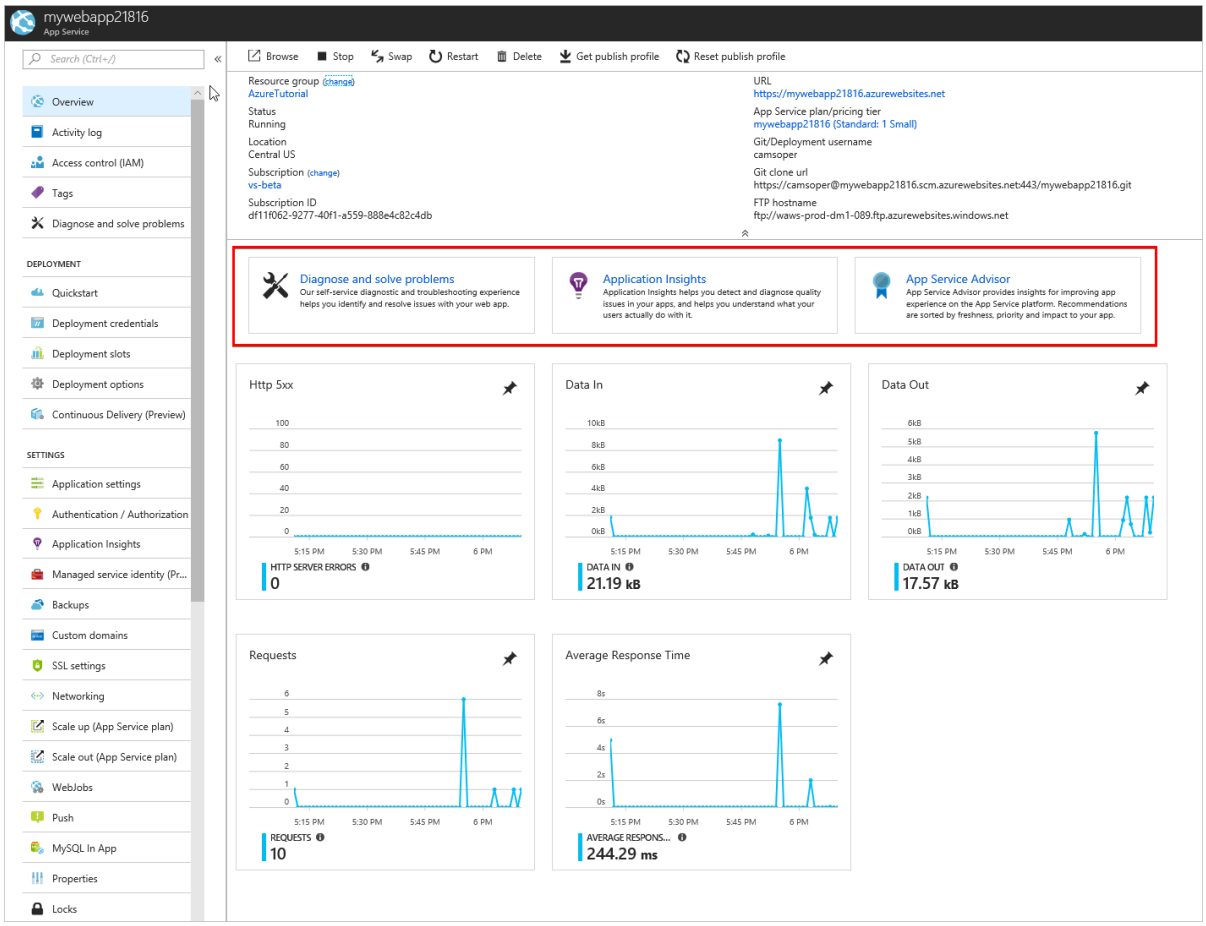
1. Open the Azure portal, and then navigate to the *mywebapp<unique_number>* App Service.
2. The **Overview** tab displays useful "at-a-glance" information, including graphs displaying recent metrics.



- **Http 5xx**: Count of server-side errors, usually exceptions in ASP.NET Core code.

- **Data In**: Data ingress coming into your web app.
- **Data Out**: Data egress from your web app to clients.
- **Requests**: Count of HTTP requests.
- **Average Response Time**: Average time for the web app to respond to HTTP requests.

Several self-service tools for troubleshooting and optimization are also found on this page.



- **Diagnose and solve problems** is a self-service troubleshooter.
- **Application Insights** is for profiling performance and app behavior, and is discussed later in this section.
- **App Service Advisor** makes recommendations to tune your app experience.

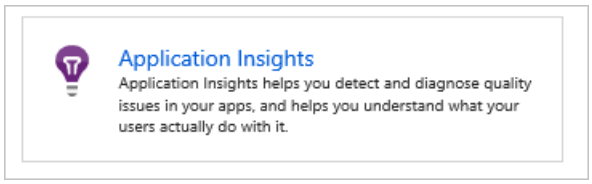
Advanced monitoring

Azure Monitor is the centralized service for monitoring all metrics and setting alerts across Azure services. Within Azure Monitor, administrators can granularly track performance and identify trends. Each Azure service offers its own set of metrics to Azure Monitor.

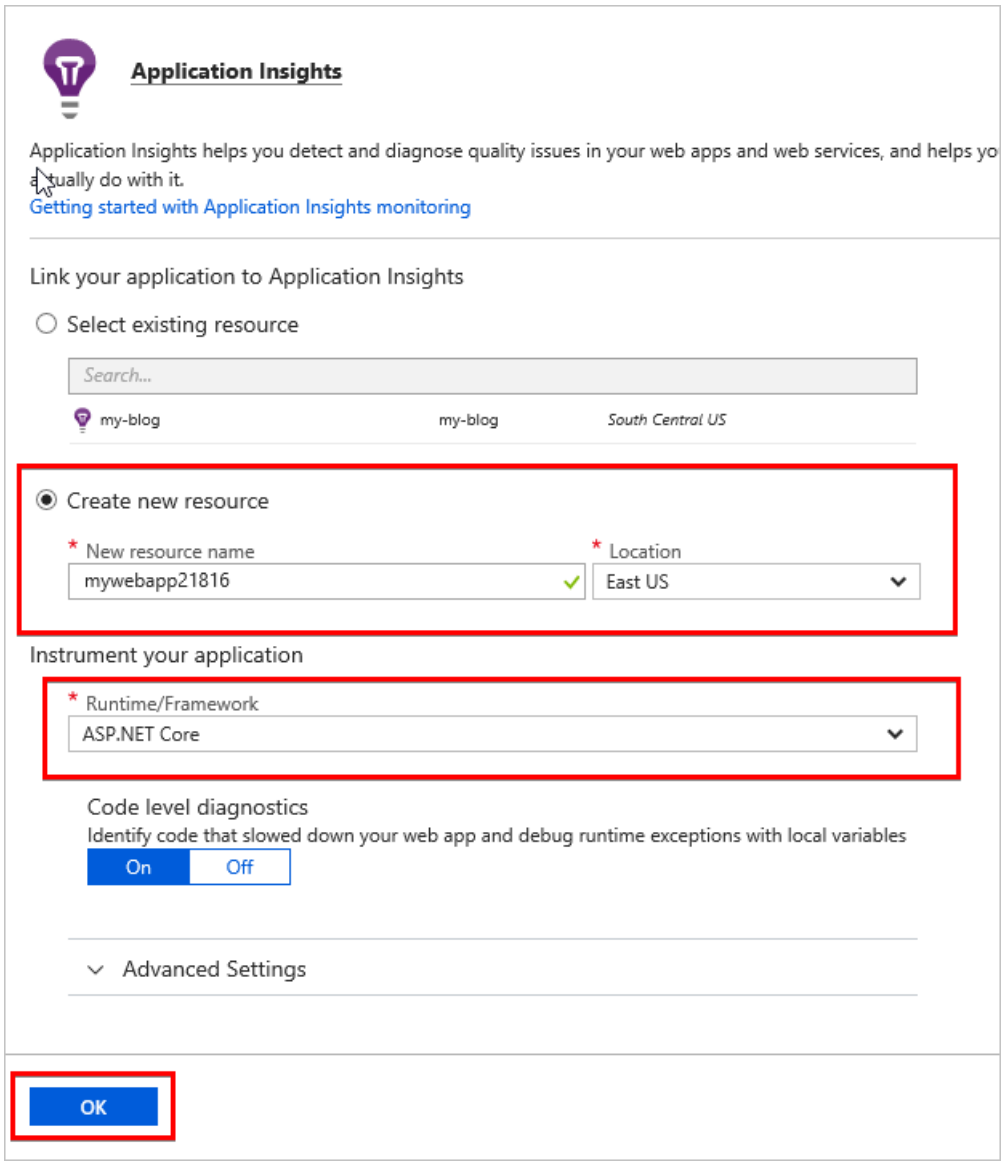
Profile with Application Insights

Application Insights is an Azure service for analyzing the performance and stability of web apps and how users use them. The data from Application Insights is broader and deeper than that of Azure Monitor. The data can provide developers and administrators with key information for improving apps. Application Insights can be added to an Azure App Service resource without code changes.

1. Open the Azure portal, and then navigate to the *mywebapp<unique_number>* App Service.
2. From the **Overview** tab, click the **Application Insights** tile.



3. Select the **Create new resource** radio button. Use the default resource name, and select the location for the Application Insights resource. The location doesn't need to match that of your web app.

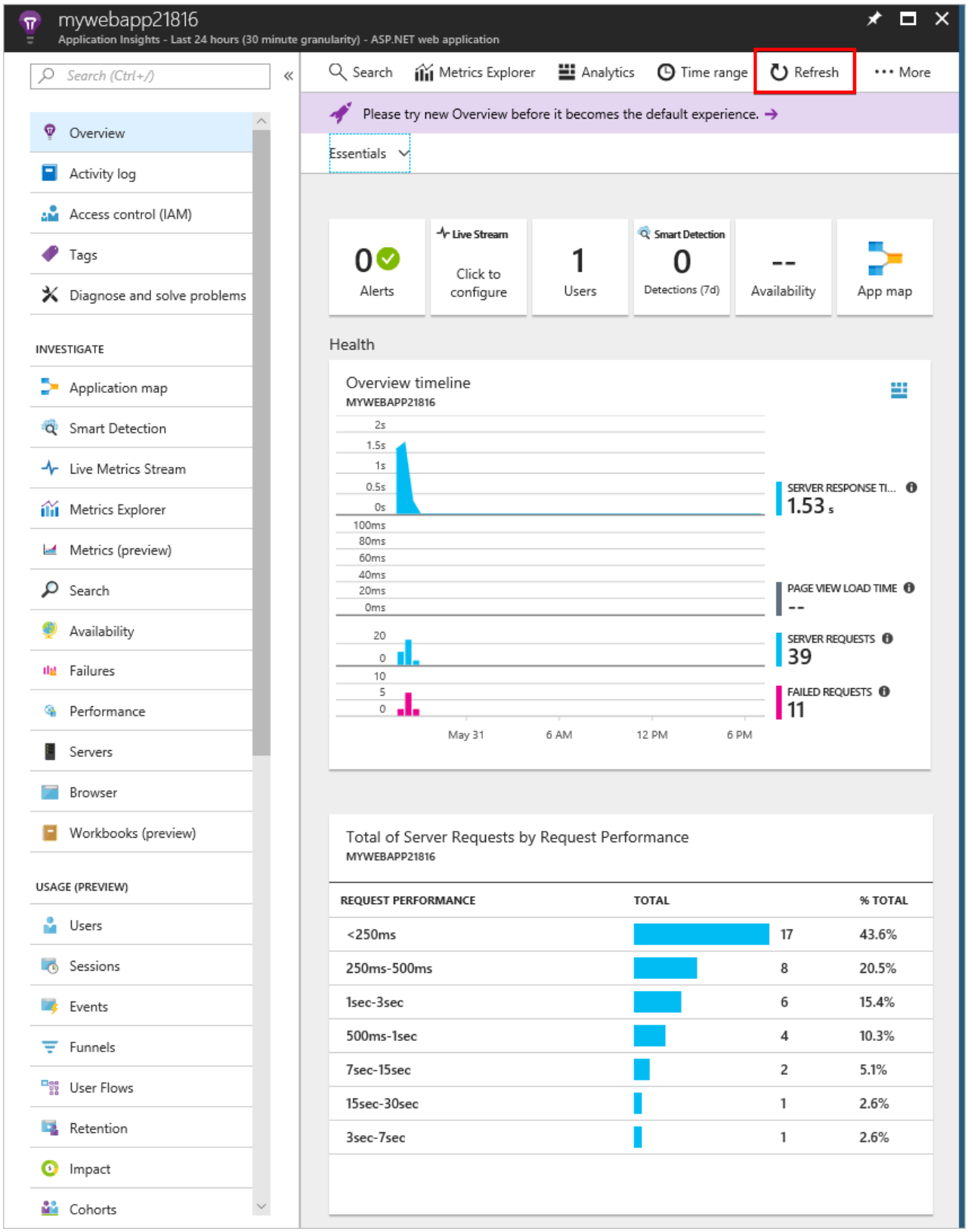


4. For **Runtime/Framework**, select **ASP.NET Core**. Accept the default settings.

5. Select **OK**. If prompted to confirm, select **Continue**.

6. After the resource has been created, click the name of Application Insights resource to navigate directly to the Application Insights page.

As the app is used, data accumulates. Select **Refresh** to reload the blade with new data.

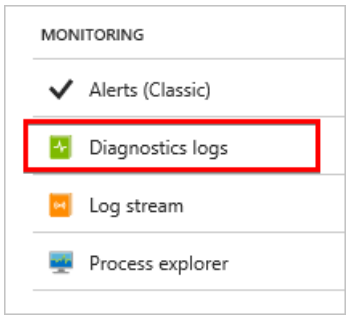


Application Insights provides useful server-side information with no additional configuration. To get the most value from Application Insights, instrument your app with the Application Insights SDK. When properly configured, the service provides end-to-end monitoring across the web server and browser, including client-side performance. For more information, see the Application Insights documentation.

Logging

Web server and app logs are disabled by default in Azure App Service. Enable the logs with the following steps:

1. Open the Azure portal, and navigate to the *mywebapp<unique_number>* App Service.
2. In the menu to the left, scroll down to the **Monitoring** section. Select **Diagnostics logs**.



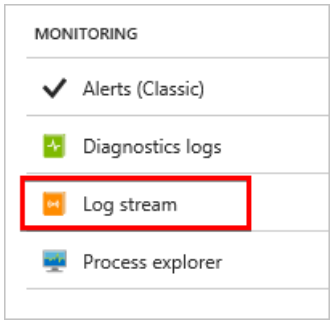
3. Turn on **Application Logging (Filesystem)**. If prompted, click the box to install the extensions to enable app logging in the web app.
4. Set **Web server logging** to **File System**.
5. Enter the **Retention Period** in days. For example, 30.
6. Click **Save**.

ASP.NET Core and web server (App Service) logs are generated for the web app. They can be downloaded using the FTP/FTPS information displayed. The password is the same as the deployment credentials created earlier in this guide. The logs can be streamed directly to your local machine with PowerShell or Azure CLI. Logs can also be viewed in Application Insights.

Log streaming

App and web server logs can be streamed in real time through the portal.

1. Open the Azure portal, and navigate to the *mywebapp<unique_number>* App Service.
2. In the menu to the left, scroll down to the **Monitoring** section and select **Log stream**.



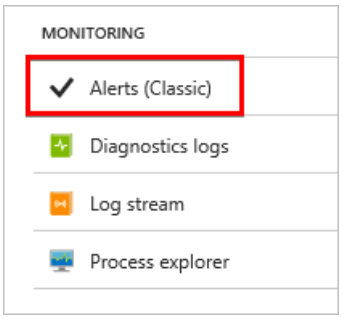
Logs can also be streamed via Azure CLI or Azure PowerShell, including through the Cloud Shell.

Alerts

Azure Monitor also provides real time alerts based on metrics, administrative events, and other criteria.

> *Note: Currently alerting on web app metrics is only available in the Alerts (classic) service.*

The Alerts (classic) service can be found in Azure Monitor or under the **Monitoring** section of the App Service settings.



Live debugging

Azure App Service can be debugged remotely with Visual Studio when logs don't provide enough information. However, remote debugging requires the app to be compiled with debug symbols. Debugging shouldn't be done in production, except as a last resort.

Conclusion

In this section, you completed the following tasks:

- Find basic monitoring and troubleshooting data in the Azure portal
- Learn how Azure Monitor provides a deeper look at metrics across all Azure services
- Connect the web app with Application Insights for app profiling
- Turn on logging and learn where to download logs
- Stream logs in real time
- Learn where to set up alerts
- Learn about remote debugging Azure App Service web apps.

Additional reading

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Monitor Azure web app performance with Application Insights
- Enable diagnostics logging for web apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Create classic metric alerts in Azure Monitor for Azure services - Azure portal

Next steps

9/22/2020 • 2 minutes to read • Edit Online

In this guide, you created a DevOps pipeline for an ASP.NET Core sample app. Congratulations! We hope you enjoyed learning to publish ASP.NET Core web apps to Azure App Service and automate the continuous integration of changes.

Beyond web hosting and DevOps, Azure has a wide array of Platform-as-a-Service (PaaS) services useful to ASP.NET Core developers. This section gives a brief overview of some of the most commonly used services.

Storage and databases

Redis Cache is high-throughput, low-latency data caching available as a service. It can be used for caching page output, reducing database requests, and providing ASP.NET Core session state across multiple instances of an app.

Azure Storage is Azure's massively scalable cloud storage. Developers can take advantage of Queue Storage for reliable message queuing, and Table Storage is a NoSQL key-value store designed for rapid development using massive, semi-structured data sets.

Azure SQL Database provides familiar relational database functionality as a service using the Microsoft SQL Server Engine.

Cosmos DB globally distributed, multi-model NoSQL database service. Multiple APIs are available, including SQL API (formerly called DocumentDB), Cassandra, and MongoDB.

Identity

Azure Active Directory and Azure Active Directory B2C are both identity services. Azure Active Directory is designed for enterprise scenarios and enables Azure AD B2B (business-to-business) collaboration, while Azure Active Directory B2C is intended business-to-customer scenarios, including social network sign-in.

Mobile

Notification Hubs is a multi-platform, scalable push-notification engine to quickly send millions of messages to apps running on various types of devices.

Web infrastructure

Azure Container Service manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized apps without container orchestration expertise.

Azure Search is used to create an enterprise search solution over private, heterogenous content.

Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices and containers.

Host ASP.NET Core on Windows with IIS

9/22/2020 • 76 minutes to read • Edit Online

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see Publish an ASP.NET Core app to IIS.

Install the .NET Core Hosting Bundle

Supported operating systems

The following operating systems are supported:

- Windows 7 or later
- Windows Server 2012 R2 or later

HTTP.sys server (formerly called WebListener) doesn't work in a reverse proxy configuration with IIS. Use the Kestrel server.

For information on hosting in Azure, see Deploy ASP.NET Core apps to Azure App Service.

For troubleshooting guidance, see Troubleshoot and debug ASP.NET Core projects.

Supported platforms

Apps published for 32-bit (x86) or 64-bit (x64) deployment are supported. Deploy a 32-bit app with a 32-bit (x86) .NET Core SDK unless the app:

- Requires the larger virtual memory address space available to a 64-bit app.
- Requires the larger IIS stack size.
- Has 64-bit native dependencies.

Apps published for 32-bit (x86) must have 32-bit enabled for their IIS Application Pools. For more information, see the Create the IIS site section.

Use a 64-bit (x64) .NET Core SDK to publish a 64-bit app. A 64-bit runtime must be present on the host system.

Hosting models

In-process hosting model

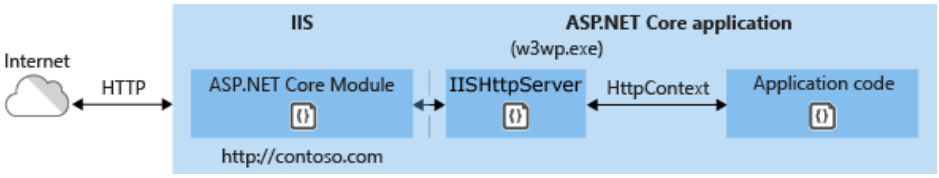
Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the Windows Process Activation Service (WAS).

The ASP.NET Core Module:

- Performs app initialization.
  - Loads the CoreCLR.
  - Calls `Program.Main`.
- Handles the lifetime of the IIS native request.

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted

in-process:



1. A request arrives from the web to the kernel-mode HTTP.sys driver.
2. The driver routes the native request to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS).
3. The ASP.NET Core Module receives the native request and passes it to IIS HTTP Server (`IISHttpServer`). IIS HTTP Server is an in-process server implementation for IIS that converts the request from native to managed.

After the IIS HTTP Server processes the request:

1. The request is sent to the ASP.NET Core middleware pipeline.
2. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic.
3. The app's response is passed back to IIS through IIS HTTP Server.
4. IIS sends the response to the client that initiated the request.

In-process hosting is opt-in for existing apps. The ASP.NET Core web templates use the in-process hosting model.

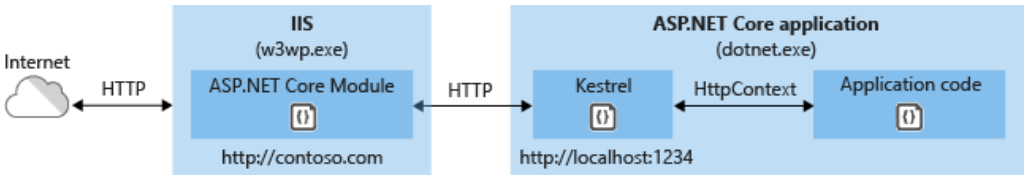
`CreateDefaultBuilder` adds an IServer instance by calling the UseIIS method to boot the CoreCLR and host the app inside of the IIS worker process (*w3wp.exe* or *iisexpress.exe*). Performance tests indicate that hosting a .NET Core app in-process delivers significantly higher request throughput compared to hosting the app out-of-process and proxying requests to Kestrel.

Apps published as a single file executable can't be loaded by the in-process hosting model.

Out-of-process hosting model

Because ASP.NET Core apps run in a process separate from the IIS worker process, the ASP.NET Core Module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



1. Requests arrive from the web to the kernel-mode HTTP.sys driver.
2. The driver routes the requests to IIS on the website's configured port. The configured port is usually 80 (HTTP) or 443 (HTTPS).
3. The module forwards the requests to Kestrel on a random port for the app. The random port isn't 80 or 443.

The ASP.NET Core Module specifies the port via an environment variable at startup. The UseIISIntegration extension configures the server to listen on `http://localhost:{PORT}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding.

Requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is forwarded into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which forwards it back to the HTTP client that initiated the request.

For ASP.NET Core Module configuration guidance, see ASP.NET Core Module.

For more information on hosting, see Host in ASP.NET Core.

Application configuration

Enable the IISIntegration components

When building a host in `CreateHostBuilder` (*Program.cs*), call CreateDefaultBuilder to enable IIS integration:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        ...
```

For more information on `CreateDefaultBuilder`, see .NET Generic Host.

IIS options

In-process hosting model

To configure IIS Server options, include a service configuration for IISServerOptions in ConfigureServices. The following example disables AutomaticAuthentication:

```
services.Configure<IISServerOptions>(options =>
{
    options.AutomaticAuthentication = false;
});
```

OPTION	DEFAULT	SETTING
`AutomaticAuthentication`	`true`	If `true`, IIS Server sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the server only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see Windows Authentication.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`AllowSynchronousIO`	`false`	Whether synchronous I/O is allowed for the `HttpContext.Request` and the `HttpContext.Response`.

OPTION	DEFAULT	SETTING
`MaxRequestBodySize`	`30000000`	Gets or sets the max request body size for the `HttpRequest`. Note that IIS itself has the limit `maxAllowedContentLength` which will be processed before the `MaxRequestBodySize` set in the `IISServerOptions`. Changing the `MaxRequestBodySize` won't affect the `maxAllowedContentLength`. To increase `maxAllowedContentLength`, add an entry in the *web.config* to set `maxAllowedContentLength` to a higher value. For more details, see Configuration.

Out-of-process hosting model

To configure IIS options, include a service configuration for IISOptions in ConfigureServices. The following example prevents the app from populating `HttpContext.Connection.ClientCertificate`:

```
services.Configure<IISOptions>(options =>
{
    options.ForwardClientCertificate = false;
});
```

OPTION	DEFAULT	SETTING
`AutomaticAuthentication`	`true`	If `true`, IIS Integration Middleware sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the middleware only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see the Windows Authentication topic.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`ForwardClientCertificate`	`true`	If `true` and the `MS-ASPNETCORE-CLIENTCERT` request header is present, the `HttpContext.Connection.ClientCertificate` is populated.

Proxy server and load balancer scenarios

The IIS Integration Middleware and the ASP.NET Core Module are configured to forward the:

- Scheme (HTTP/HTTPS).
- Remote IP address where the request originated.

The IIS Integration Middleware configures Forwarded Headers Middleware.

Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

web.config file

The *web.config* file configures the ASP.NET Core Module. Creating, transforming, and publishing the *web.config* file is handled by an MSBuild target (`_TransformWebConfig`) when the project is published. This target is present in the Web SDK targets (`Microsoft.NET.Sdk.Web`). The SDK is set at the top of the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

If a *web.config* file isn't present in the project, the file is created with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output.

If a *web.config* file is present in the project, the file is transformed with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output. The transformation doesn't modify IIS configuration settings in the file.

The *web.config* file may provide additional IIS configuration settings that control active IIS modules. For information on IIS modules that are capable of processing requests with ASP.NET Core apps, see the IIS modules topic.

To prevent the Web SDK from transforming the *web.config* file, use the **<IsTransformWebConfigDisabled>** property in the project file:

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

When disabling the Web SDK from transforming the file, the *processPath* and *arguments* should be manually set by the developer. For more information, see ASP.NET Core Module.

web.config file location

In order to set up the ASP.NET Core Module correctly, the *web.config* file must be present at the content root path (typically the app base path) of the deployed app. This is the same location as the website physical path provided to IIS. The *web.config* file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.

Sensitive files exist on the app's physical path, such as *<assembly>.runtimeconfig.json*, *<assembly>.xml* (XML Documentation comments), and *<assembly>.deps.json*. When the *web.config* file is present and the site starts normally, IIS doesn't serve these sensitive files if they're requested. If the *web.config* file is missing, incorrectly named, or unable to configure the site for normal startup, IIS may serve sensitive files publicly.

The *web.config* file must be present in the deployment at all times, correctly named, and able to configure the site for normal start up. Never remove the *web.config* file from a production deployment.

Transform web.config

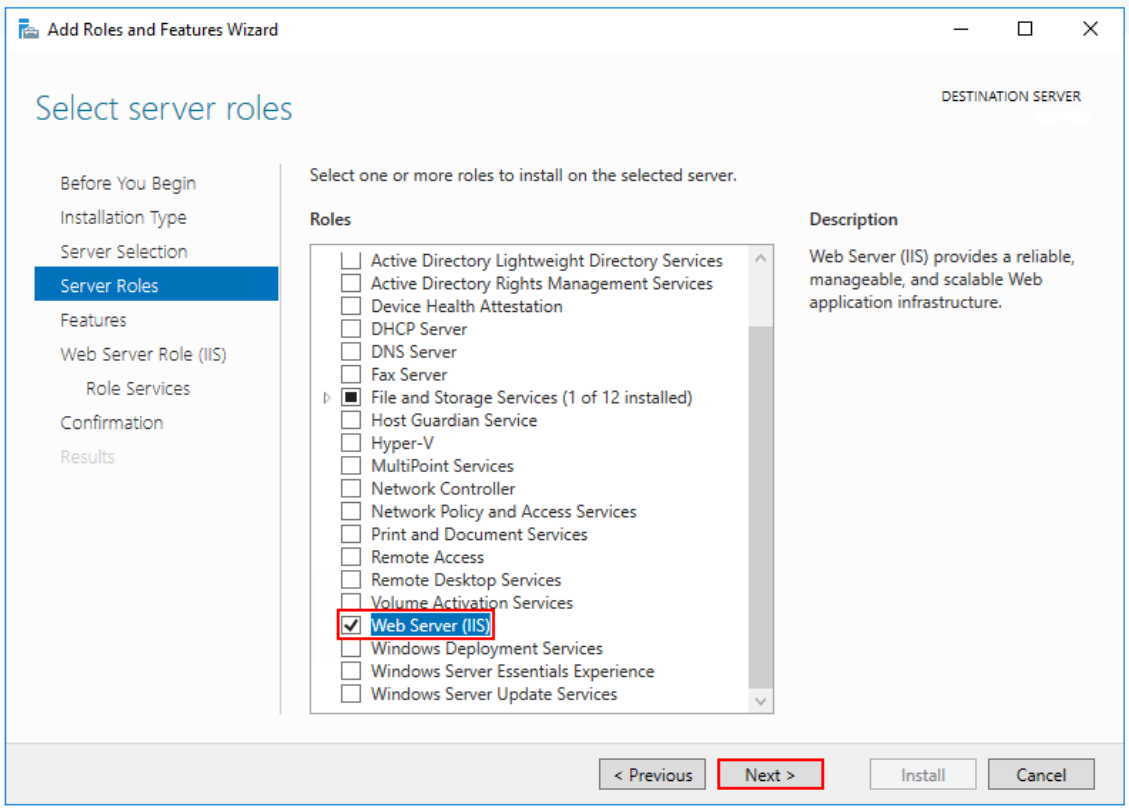
If you need to transform *web.config* on publish, see Transform web.config. You might need to transform *web.config* on publish to set environment variables based on the configuration, profile, or environment.

IIS configuration

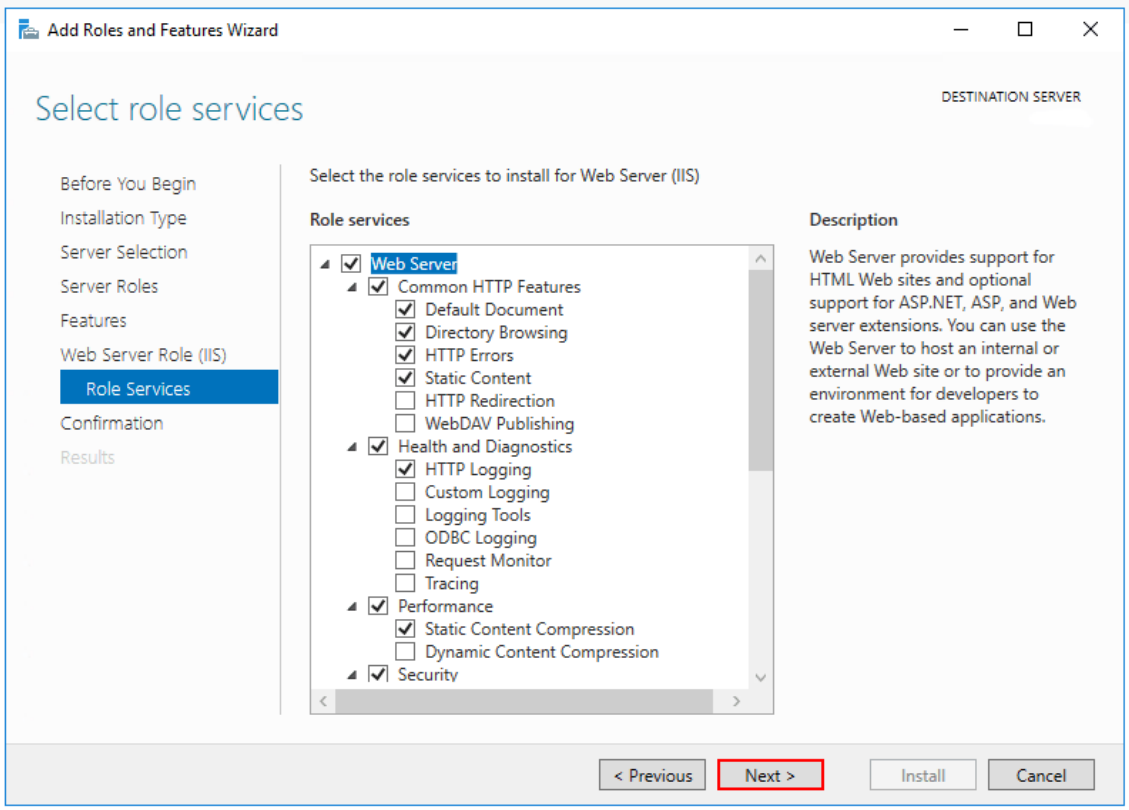
Windows Server operating systems

Enable the **Web Server (IIS)** server role and establish role services.

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



2. After the **Features** step, the **Role services** step loads for Web Server (IIS). Select the IIS role services desired or accept the default role services provided.



Windows Authentication (Optional)

To enable Windows Authentication, expand the following nodes: **Web Server** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication

<windowsAuthentication> and Configure Windows authentication.

WebSockets (Optional)
WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **Web Server** > **Application Development**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

3. Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart isn't required after installing the **Web Server (IIS)** role.

Windows desktop operating systems

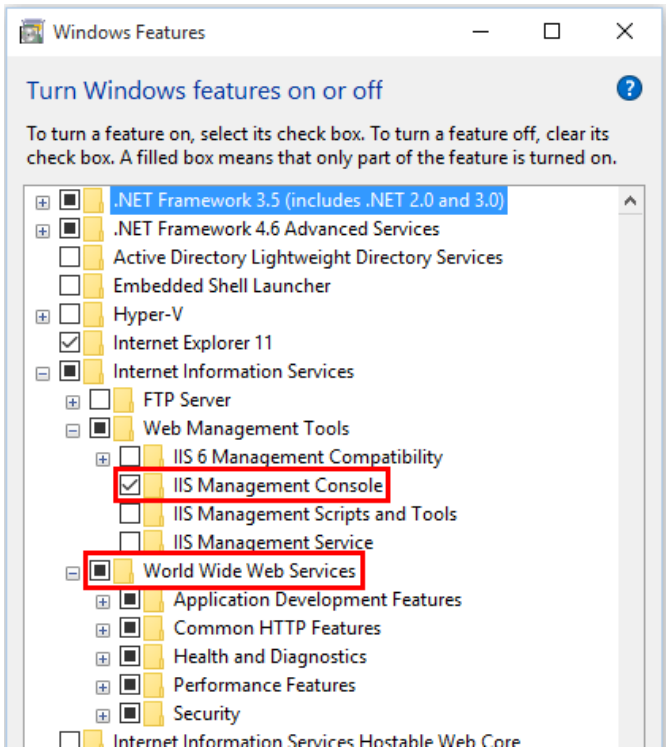
Enable the **IIS Management Console** and **World Wide Web Services**.

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the **Internet Information Services** node. Open the **Web Management Tools** node.
3. Check the box for **IIS Management Console**.
4. Check the box for **World Wide Web Services**.
5. Accept the default features for **World Wide Web Services** or customize the IIS features.

   Windows Authentication (Optional)
   To enable Windows Authentication, expand the following nodes: **World Wide Web Services** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

   WebSockets (Optional)
   WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **World Wide Web Services** > **Application Development Features**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

6. If the IIS installation requires a restart, restart the system.



Install the .NET Core Hosting Bundle

Install the *.NET Core Hosting Bundle* on the hosting system. The bundle installs the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. The module allows ASP.NET Core apps to run behind IIS.

> **IMPORTANT**
>
> If the Hosting Bundle is installed before IIS, the bundle installation must be repaired. Run the Hosting Bundle installer again after installing IIS.
>
> If the Hosting Bundle is installed after installing the 64-bit (x64) version of .NET Core, SDKs might appear to be missing (No .NET Core SDKs were detected). To resolve the problem, see Troubleshoot and debug ASP.NET Core projects.

Direct download (current version)

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

Earlier versions of the installer

To obtain an earlier version of the installer:

1. Navigate to the Download .NET Core page.
2. Select the desired .NET Core version.
3. In the **Run apps - Runtime** column, find the row of the .NET Core runtime version desired.
4. Download the installer using the **Hosting Bundle** link.

> **WARNING**
>
> Some installers contain release versions that have reached their end of life (EOL) and are no longer supported by Microsoft. For more information, see the support policy.

Install the Hosting Bundle

1. Run the installer on the server. The following parameters are available when running the installer from an administrator command shell:

   - `OPT_NO_ANCM=1`: Skip installing the ASP.NET Core Module.
   - `OPT_NO_RUNTIME=1`: Skip installing the .NET Core runtime. Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_SHAREDFX=1`: Skip installing the ASP.NET Shared Framework (ASP.NET runtime). Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_X86=1`: Skip installing x86 runtimes. Use this parameter when you know that you won't be hosting 32-bit apps. If there's any chance that you will host both 32-bit and 64-bit apps in the future, don't use this parameter and install both runtimes.
   - `OPT_NO_SHARED_CONFIG_CHECK=1`: Disable the check for using an IIS Shared Configuration when the shared configuration (*applicationHost.config*) is on the same machine as the IIS installation. *Only available for ASP.NET Core 2.2 or later Hosting Bundler installers.* For more information, see ASP.NET Core Module.

2. Restart the system or execute the following commands in a command shell:

   ```
   net stop was /y
   net start w3svc
   ```

   Restarting IIS picks up a change to the system PATH, which is an environment variable, made by the installer.

ASP.NET Core doesn't adopt roll-forward behavior for patch releases of shared framework packages. After upgrading the shared framework by installing a new hosting bundle, restart the system or execute the following commands in a command shell:

```
net stop was /y
net start w3svc
```

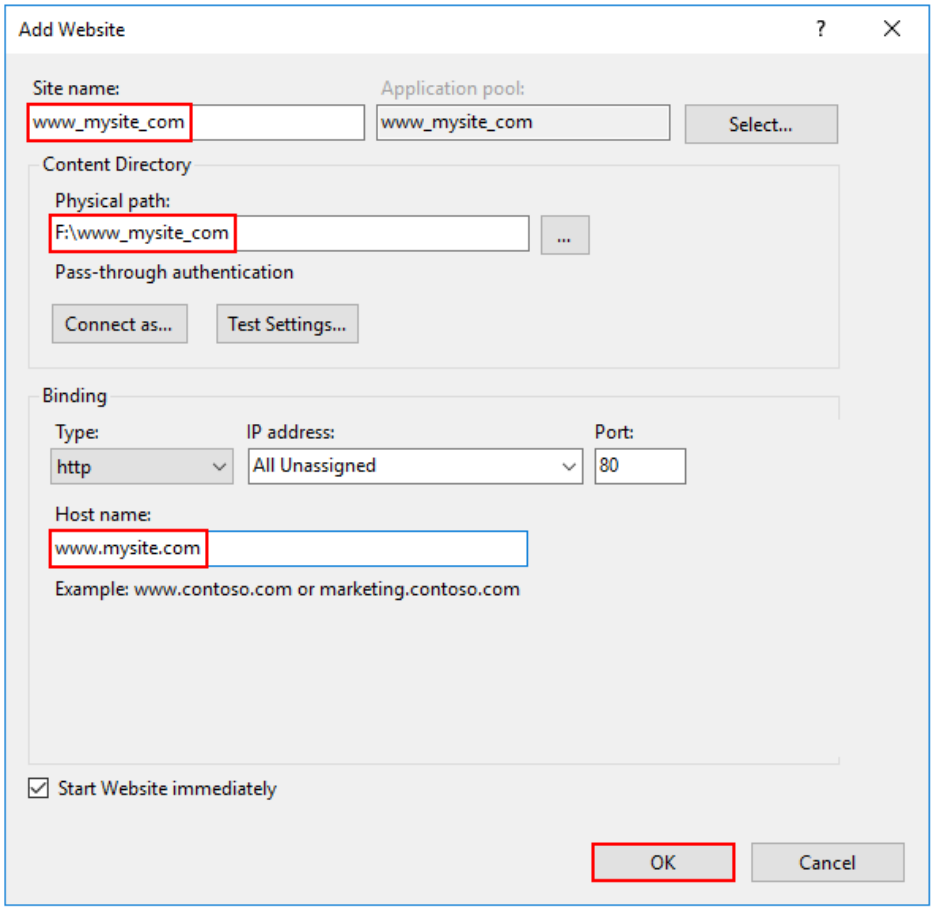
> **NOTE**
> For information on IIS Shared Configuration, see ASP.NET Core Module with IIS Shared Configuration.

Install Web Deploy when publishing with Visual Studio

When deploying apps to servers with Web Deploy, install the latest version of Web Deploy on the server. To install Web Deploy, use the Web Platform Installer (WebPI) or obtain an installer directly from the Microsoft Download Center. The preferred method is to use WebPI. WebPI offers a standalone setup and a configuration for hosting providers.

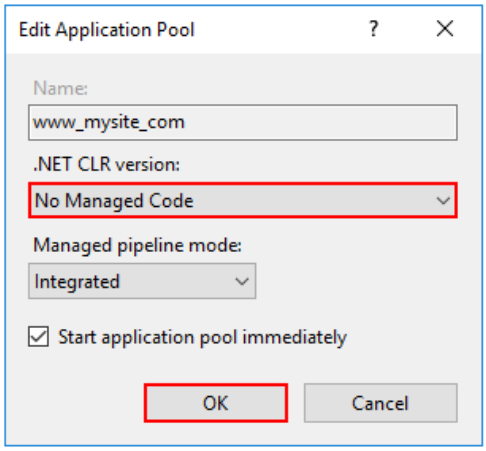
Create the IIS site

1. On the hosting system, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see ASP.NET Core directory structure.
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.
3. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website by selecting **OK**:



> **WARNING**
> Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have this security risk if you control the entire parent domain (as opposed to `*.com` , which is vulnerable). See rfc7230 section-5.4 for more information.

4. Under the server's node, select **Application Pools**.
5. Right-click the site's app pool and select **Basic Settings** from the contextual menu.
6. In the **Edit Application Pool** window, set the **.NET CLR version** to **No Managed Code**:



   ASP.NET Core runs in a separate process and manages the runtime. ASP.NET Core doesn't rely on loading the desktop CLR (.NET CLR). The Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process. Setting the **.NET CLR version** to **No Managed Code** is optional but recommended.

7. *ASP.NET Core 2.2 or later*:

   - For a 32-bit (x86) self-contained deployment published with a 32-bit SDK that uses the in-process hosting model, enable the Application Pool for 32-bit. In IIS Manager, navigate to **Application Pools** in the **Connections** sidebar. Select the app's Application Pool. In the **Actions** sidebar, select **Advanced Settings**. Set **Enable 32-Bit Applications** to `True`.
   - For a 64-bit (x64) self-contained deployment that uses the in-process hosting model, disable the app pool for 32-bit (x86) processes. In IIS Manager, navigate to **Application Pools** in the **Connections** sidebar. Select the app's Application Pool. In the **Actions** sidebar, select **Advanced Settings**. Set **Enable 32-Bit Applications** to `False`.

8. Confirm the process model identity has the proper permissions.

   If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

Windows Authentication configuration (Optional)

For more information, see Configure Windows authentication.

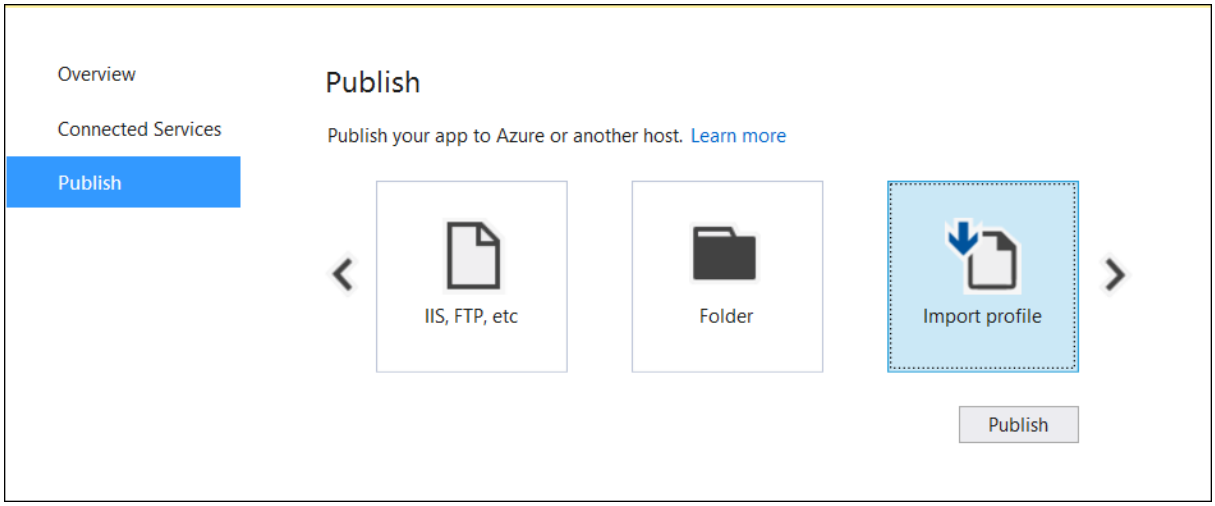
Deploy the app

Deploy the app to the IIS **Physical path** folder that was established in the Create the IIS site section. Web

Deploy is the recommended mechanism for deployment, but several options exist for moving the app from the project's *publish* folder to the hosting system's deployment folder.

Web Deploy with Visual Studio

See the Visual Studio publish profiles for ASP.NET Core app deployment topic to learn how to create a publish profile for use with Web Deploy. If the hosting provider provides a Publish Profile or support for creating one, download their profile and import it using the Visual Studio **Publish** dialog:



Web Deploy outside of Visual Studio

Web Deploy can also be used outside of Visual Studio from the command line. For more information, see Web Deployment Tool.

Alternatives to Web Deploy

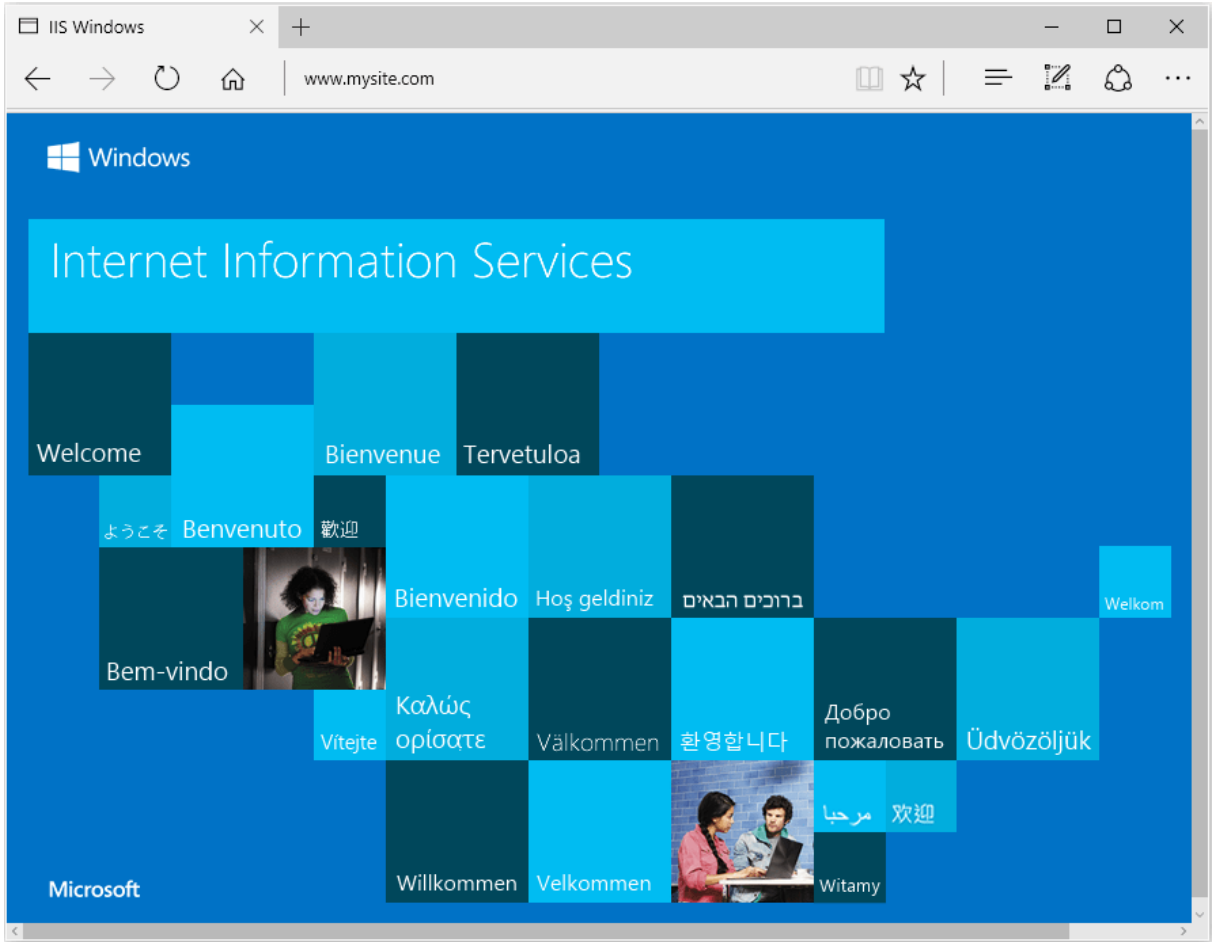
Use any of several methods to move the app to the hosting system, such as manual copy, Xcopy, Robocopy, or PowerShell.

For more information on ASP.NET Core deployment to IIS, see the Deployment resources for IIS administrators section.

Browse the website

After the app is deployed to the hosting system, make a request to one of the app's public endpoints.

In the following example, the site is bound to an IIS **Host name** of `www.mysite.com` on **Port** `80`. A request is made to `http://www.mysite.com`:



Locked deployment files

Files in the deployment folder are locked when the app is running. Locked files can't be overwritten during deployment. To release locked files in a deployment, stop the app pool using **one** of the following approaches:

- Use Web Deploy and reference `Microsoft.NET.Sdk.Web` in the project file. An *app_offline.htm* file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the *app_offline.htm* file during the deployment. For more information, see the ASP.NET Core Module configuration reference.
- Manually stop the app pool in the IIS Manager on the server.
- Use PowerShell to drop *app_offline.htm* (requires PowerShell 5 or later):

```
$pathToApp = 'PATH_TO_APP'

# Stop the AppPool
New-Item -Path $pathToApp app_offline.htm

# Provide script commands here to deploy the app

# Restart the AppPool
Remove-Item -Path $pathToApp app_offline.htm
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including middleware used in authentication. Even if Data Protection APIs aren't called by user code, data protection should be configured with a deployment script or in user code to create a persistent cryptographic key store. If data

protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection under IIS to persist the key ring, use **one** of the following approaches:

- **Create Data Protection Registry Keys**

  Data protection keys used by ASP.NET Core apps are stored in the registry external to the apps. To persist the keys for a given app, create registry keys for the app pool.

  For standalone, non-webfarm IIS installations, the Data Protection Provision-AutoGenKeys.ps1 PowerShell script can be used for each app pool used with an ASP.NET Core app. This script creates a registry key in the HKLM registry that's accessible only to the worker process account of the app's app pool. Keys are encrypted at rest using DPAPI with a machine-wide key.

  In web farm scenarios, an app can be configured to use a UNC path to store its data protection key ring. By default, the data protection keys aren't encrypted. Ensure that the file permissions for the network share are limited to the Windows account the app runs under. An X509 certificate can be used to protect keys at rest. Consider a mechanism to allow users to upload certificates: Place certificates into the user's trusted certificate store and ensure they're available on all machines where the user's app runs. See Configure ASP.NET Core Data Protection for details.

- **Configure the IIS Application Pool to load the user profile**

  This setting is in the **Process Model** section under the **Advanced Settings** for the app pool. Set **Load User Profile** to `True`. When set to `True`, keys are stored in the user profile directory and protected using DPAPI with a key specific to the user account. Keys are persisted to the *%LOCALAPPDATA%/ASP.NET/DataProtection-Keys* folder.

  The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:

  1. Navigate to the *%windir%/system32/inetsrv/config* folder.
  2. Open the *applicationHost.config* file.
  3. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.
  4. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

- **Use the file system as a key ring store**

  Adjust the app code to use the file system as a key ring store. Use an X509 certificate to protect the key ring and ensure the certificate is a trusted certificate. If the certificate is self-signed, place the certificate in the Trusted Root store.

  When using IIS in a web farm:

  - Use a file share that all machines can access.
  - Deploy an X509 certificate to each machine. Configure data protection in code.

- **Set a machine-wide policy for data protection**

  The data protection system has limited support for setting a default machine-wide policy for all apps

that consume the Data Protection APIs. For more information, see ASP.NET Core Data Protection.

Virtual Directories

IIS Virtual Directories aren't supported with ASP.NET Core apps. An app can be hosted as a sub-application.

Sub-applications

An ASP.NET Core app can be hosted as an IIS sub-application (sub-app). The sub-app's path becomes part of the root app's URL.

Static asset links within the sub-app should use tilde-slash (`~/`) notation. Tilde-slash notation triggers a Tag Helper to prepend the sub-app's pathbase to the rendered relative link. For a sub-app at `/subapp_path`, an image linked with `src="~/image.png"` is rendered as `src="/subapp_path/image.png"`. The root app's Static File Middleware doesn't process the static file request. The request is processed by the sub-app's Static File Middleware.

If a static asset's `src` attribute is set to an absolute path (for example, `src="/image.png"`), the link is rendered without the sub-app's pathbase. The root app's Static File Middleware attempts to serve the asset from the root app's web root, which results in a *404 - Not Found* response unless the static asset is available from the root app.

To host an ASP.NET Core app as a sub-app under another ASP.NET Core app:

1. Establish an app pool for the sub-app. Set the **.NET CLR Version** to **No Managed Code** because the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process, not the desktop CLR (.NET CLR).
2. Add the root site in IIS Manager with the sub-app in a folder under the root site.
3. Right-click the sub-app folder in IIS Manager and select **Convert to Application**.
4. In the **Add Application** dialog, use the **Select** button for the **Application Pool** to assign the app pool that you created for the sub-app. Select **OK**.

The assignment of a separate app pool to the sub-app is a requirement when using the in-process hosting model.

For more information on the in-process hosting model and configuring the ASP.NET Core Module, see ASP.NET Core Module.

Configuration of IIS with web.config

IIS configuration is influenced by the `<system.webServer>` section of *web.config* for IIS scenarios that are functional for ASP.NET Core apps with the ASP.NET Core Module. For example, IIS configuration is functional for dynamic compression. If IIS is configured at the server level to use dynamic compression, the `<urlCompression>` element in the app's *web.config* file can disable it for an ASP.NET Core app.

For more information, see the following topics:

- Configuration reference for <system.webServer>
- ASP.NET Core Module
- IIS modules with ASP.NET Core

To set environment variables for individual apps running in isolated app pools (supported for IIS 10.0 or later), see the *AppCmd.exe command* section of the Environment Variables <environmentVariables> topic in the IIS reference documentation.

Configuration sections of web.config

Configuration sections of ASP.NET 4.x apps in *web.config* aren't used by ASP.NET Core apps for configuration:

- `<system.web>`
- `<appSettings>`
- `<connectionStrings>`
- `<location>`

ASP.NET Core apps are configured using other configuration providers. For more information, see Configuration.

Application Pools

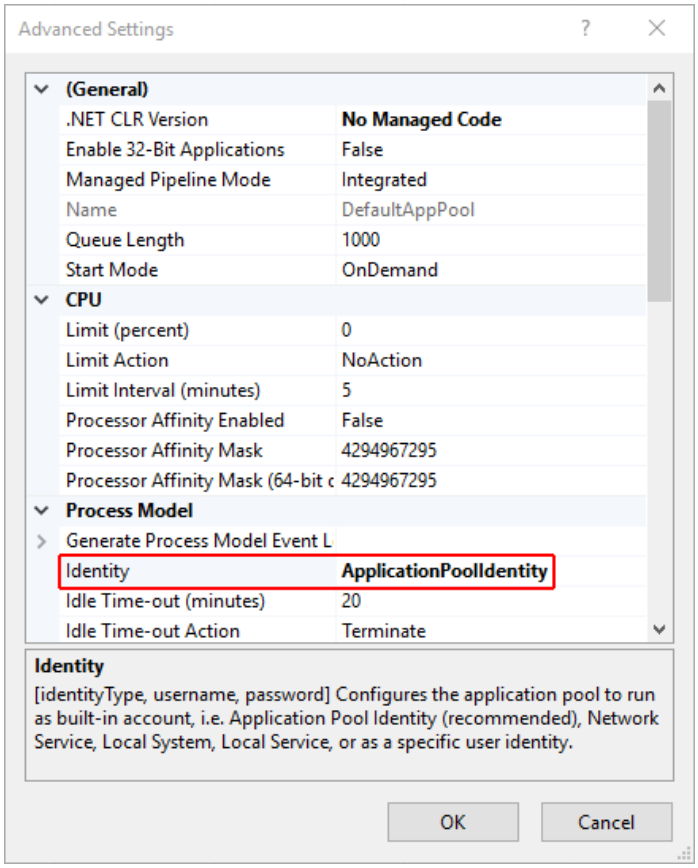
App pool isolation is determined by the hosting model:

- In-process hosting: Apps are required to run in separate app pools.
- Out-of-process hosting: We recommend isolating the apps from each other by running each app in its own app pool.

The IIS **Add Website** dialog defaults to a single app pool per app. When a **Site name** is provided, the text is automatically transferred to the **Application pool** textbox. A new app pool is created using the site name when the site is added.

Application Pool Identity

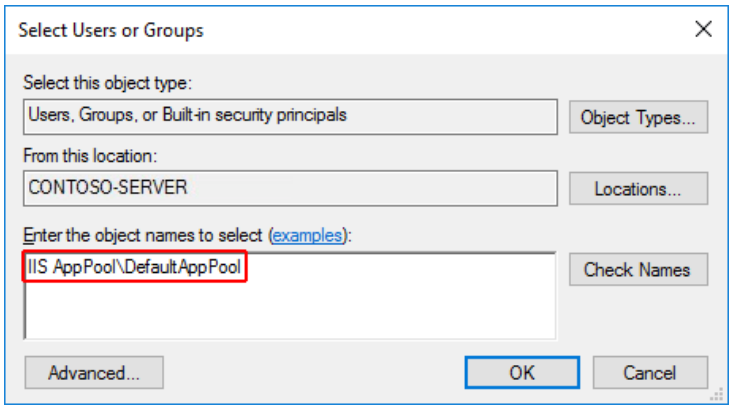
An app pool identity account allows an app to run under a unique account without having to create and manage domains or local accounts. On IIS 8.0 or later, the IIS Admin Worker Process (WAS) creates a virtual account with the name of the new app pool and runs the app pool's worker processes under this account by default. In the IIS Management Console under **Advanced Settings** for the app pool, ensure that the **Identity** is set to use **ApplicationPoolIdentity**:



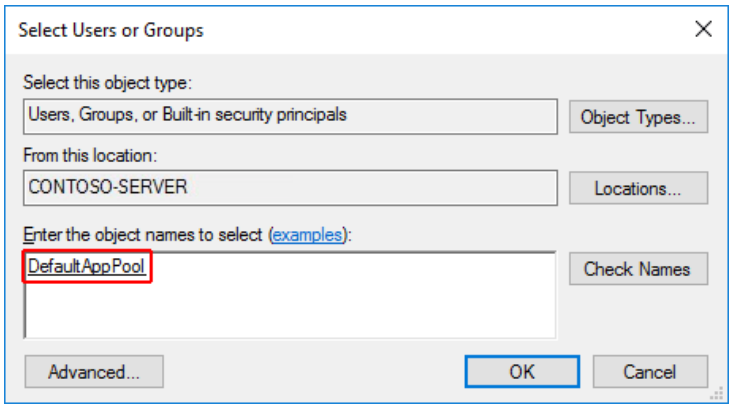
The IIS management process creates a secure identifier with the name of the app pool in the Windows Security System. Resources can be secured using this identity. However, this identity isn't a real user account and doesn't show up in the Windows User Management Console.

If the IIS worker process requires elevated access to the app, modify the Access Control List (ACL) for the directory containing the app:

1. Open Windows Explorer and navigate to the directory.
2. Right-click on the directory and select **Properties**.
3. Under the **Security** tab, select the **Edit** button and then the **Add** button.
4. Select the **Locations** button and make sure the system is selected.
5. Enter **IIS AppPool\<app_pool_name>** in **Enter the object names to select** area. Select the **Check Names** button. For the *DefaultAppPool* check the names using **IIS AppPool\DefaultAppPool**. When the **Check Names** button is selected, a value of **DefaultAppPool** is indicated in the object names area. It isn't possible to enter the app pool name directly into the object names area. Use the **IIS AppPool\<app_pool_name>** format when checking for the object name.



6. Select **OK**.



7. Read & execute permissions should be granted by default. Provide additional permissions as needed.

Access can also be granted at a command prompt using the **ICACLS** tool. Using the *DefaultAppPool* as an example, the following command is used:

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool":F
```

For more information, see the icacls topic.

HTTP/2 support

HTTP/2 is supported with ASP.NET Core in the following IIS deployment scenarios:

- In-process
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - TLS 1.2 or later connection
- Out-of-process
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Public-facing edge server connections use HTTP/2, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.
  - TLS 1.2 or later connection

For an in-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`. For an out-of-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/1.1`.

For more information on the in-process and out-of-process hosting models, see ASP.NET Core Module.

HTTP/2 is enabled by default. Connections fall back to HTTP/1.1 if an HTTP/2 connection isn't established. For more information on HTTP/2 configuration with IIS deployments, see HTTP/2 on IIS.

CORS preflight requests

This section only applies to ASP.NET Core apps that target the .NET Framework.

For an ASP.NET Core app that targets the .NET Framework, OPTIONS requests aren't passed to the app by default in IIS. To learn how to configure the app's IIS handlers in *web.config* to pass OPTIONS requests, see Enable cross-origin requests in ASP.NET Web API 2: How CORS Works.

Application Initialization Module and Idle Timeout

When hosted in IIS by the ASP.NET Core Module version 2:

- Application Initialization Module: App's hosted in-process or out-of-process can be configured to start automatically on a worker process restart or server restart.
- Idle Timeout: App's hosted in-process can be configured not to timeout during periods of inactivity.

Application Initialization Module

Applies to apps hosted in-process and out-of-process.

IIS Application Initialization is an IIS feature that sends an HTTP request to the app when the app pool starts or is recycled. The request triggers the app to start. By default, IIS issues a request to the app's root URL (`/`) to initialize the app (see the additional resources for more details on configuration).

Confirm that the IIS Application Initialization role feature in enabled:

On Windows 7 or later desktop systems when using IIS locally:

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open **Internet Information Services** > **World Wide Web Services** > **Application Development Features**.
3. Select the check box for **Application Initialization**.

On Windows Server 2008 R2 or later:

1. Open the **Add Roles and Features Wizard**.
2. In the **Select role services** panel, open the **Application Development** node.

3. Select the check box for **Application Initialization**.

Use either of the following approaches to enable the Application Initialization Module for the site:

- Using IIS Manager:

  1. Select **Application Pools** in the **Connections** panel.
  2. Right-click the app's app pool in the list and select **Advanced Settings**.
  3. The default **Start Mode** is **OnDemand**. Set the **Start Mode** to **AlwaysRunning**. Select **OK**.
  4. Open the **Sites** node in the **Connections** panel.
  5. Right-click the app and select **Manage Website** > **Advanced Settings**.
  6. The default **Preload Enabled** setting is **False**. Set **Preload Enabled** to **True**. Select **OK**.

- Using *web.config*, add the `<applicationInitialization>` element with `doAppInitAfterRestart` set to `true` to the `<system.webServer>` elements in the app's *web.config* file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <applicationInitialization doAppInitAfterRestart="true" />
    </system.webServer>
  </location>
</configuration>
```

Idle Timeout

Only applies to apps hosted in-process.

To prevent the app from idling, set the app pool's idle timeout using IIS Manager:

1. Select **Application Pools** in the **Connections** panel.
2. Right-click the app's app pool in the list and select **Advanced Settings**.
3. The default **Idle Time-out (minutes)** is **20** minutes. Set the **Idle Time-out (minutes)** to **0** (zero). Select **OK**.
4. Recycle the worker process.

To prevent apps hosted out-of-process from timing out, use either of the following approaches:

- Ping the app from an external service in order to keep it running.
- If the app only hosts background services, avoid IIS hosting and use a Windows Service to host the ASP.NET Core app.

Application Initialization Module and Idle Timeout additional resources

- IIS 8.0 Application Initialization
- Application Initialization <applicationInitialization>.
- Process Model Settings for an Application Pool <processModel>.

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS

- Common errors reference for Azure App Service and IIS with ASP.NET Core

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Introduction to ASP.NET Core
- The Official Microsoft IIS Site
- Windows Server technical content library
- HTTP/2 on IIS
- Transform web.config

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see Publish an ASP.NET Core app to IIS.

Install the .NET Core Hosting Bundle

Supported operating systems

The following operating systems are supported:

- Windows 7 or later
- Windows Server 2008 R2 or later

HTTP.sys server (formerly called WebListener) doesn't work in a reverse proxy configuration with IIS. Use the Kestrel server.

For information on hosting in Azure, see Deploy ASP.NET Core apps to Azure App Service.

For troubleshooting guidance, see Troubleshoot and debug ASP.NET Core projects.

Supported platforms

Apps published for 32-bit (x86) or 64-bit (x64) deployment are supported. Deploy a 32-bit app with a 32-bit (x86) .NET Core SDK unless the app:

- Requires the larger virtual memory address space available to a 64-bit app.
- Requires the larger IIS stack size.
- Has 64-bit native dependencies.

Use a 64-bit (x64) .NET Core SDK to publish a 64-bit app. A 64-bit runtime must be present on the host system.

Hosting models

In-process hosting model

Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because:

- Requests aren't proxied over the loopback adapter. A loopback adapter is a network interface that returns outgoing network traffic back to the same machine.

IIS handles process management with the Windows Process Activation Service (WAS).

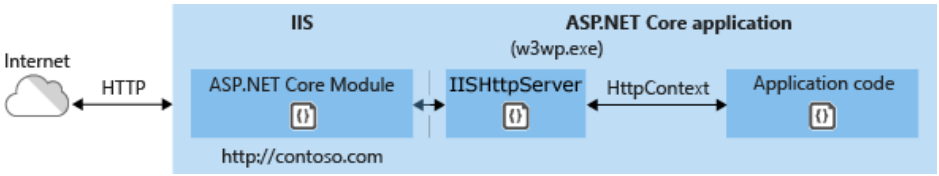
The ASP.NET Core Module:

- Performs app initialization.
  - Loads the CoreCLR.

    - Calls `Program.Main`.
- Handles the lifetime of the IIS native request.

The in-process hosting model isn't supported for ASP.NET Core apps that target the .NET Framework.

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted in-process:



A request arrives from the web to the kernel-mode HTTP.sys driver. The driver routes the native request to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The ASP.NET Core Module receives the native request and passes it to IIS HTTP Server (`IISHttpServer`). IIS HTTP Server is an in-process server implementation for IIS that converts the request from native to managed.

After the IIS HTTP Server processes the request, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. The app's response is passed back to IIS through IIS HTTP Server. IIS sends the response to the client that initiated the request.

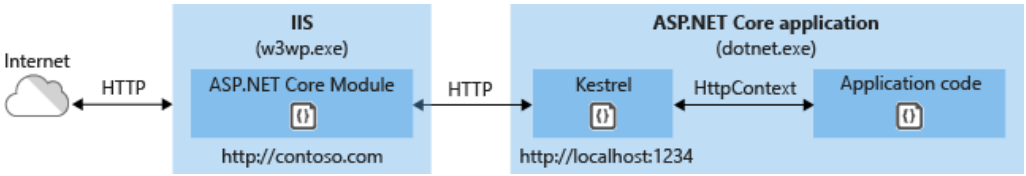
In-process hosting is opt-in for existing apps, but dotnet new templates default to the in-process hosting model for all IIS and IIS Express scenarios.

`CreateDefaultBuilder` adds an IServer instance by calling the UseIIS method to boot the CoreCLR and host the app inside of the IIS worker process (*w3wp.exe* or *iisexpress.exe*). Performance tests indicate that hosting a .NET Core app in-process delivers significantly higher request throughput compared to hosting the app out-of-process and proxying requests to Kestrel server.

Out-of-process hosting model

Because ASP.NET Core apps run in a process separate from the IIS worker process, the ASP.NET Core Module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the UseIISIntegration extension configures the server to listen on `http://localhost:{PORT}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for

forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.

For ASP.NET Core Module configuration guidance, see ASP.NET Core Module.

For more information on hosting, see Host in ASP.NET Core.

Application configuration

Enable the IISIntegration components

When building a host in `CreateWebHostBuilder` (*Program.cs*), call CreateDefaultBuilder to enable IIS integration:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        ...
```

For more information on `CreateDefaultBuilder`, see ASP.NET Core Web Host.

IIS options

In-process hosting model

To configure IIS Server options, include a service configuration for IISServerOptions in ConfigureServices. The following example disables AutomaticAuthentication:

```
services.Configure<IISServerOptions>(options =>
{
    options.AutomaticAuthentication = false;
});
```

OPTION	DEFAULT	SETTING
`AutomaticAuthentication`	`true`	If `true`, IIS Server sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the server only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see Windows Authentication.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.

Out-of-process hosting model

To configure IIS options, include a service configuration for IISOptions in ConfigureServices. The following example prevents the app from populating `HttpContext.Connection.ClientCertificate`:

```
services.Configure<IISOptions>(options =>
{
    options.ForwardClientCertificate = false;
});
```

OPTION	DEFAULT	SETTING
`AutomaticAuthentication`	`true`	If `true`, IIS Integration Middleware sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the middleware only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see the Windows Authentication topic.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`ForwardClientCertificate`	`true`	If `true` and the `MS-ASPNETCORE-CLIENTCERT` request header is present, the `HttpContext.Connection.ClientCertificate` is populated.

Proxy server and load balancer scenarios

The IIS Integration Middleware, which configures Forwarded Headers Middleware, and the ASP.NET Core Module are configured to forward the scheme (HTTP/HTTPS) and the remote IP address where the request originated. Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

web.config file

The *web.config* file configures the ASP.NET Core Module. Creating, transforming, and publishing the *web.config* file is handled by an MSBuild target (`_TransformWebConfig`) when the project is published. This target is present in the Web SDK targets (`Microsoft.NET.Sdk.Web`). The SDK is set at the top of the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

If a *web.config* file isn't present in the project, the file is created with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output.

If a *web.config* file is present in the project, the file is transformed with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output. The transformation doesn't modify IIS configuration settings in the file.

The *web.config* file may provide additional IIS configuration settings that control active IIS modules. For information on IIS modules that are capable of processing requests with ASP.NET Core apps, see the IIS modules topic.

To prevent the Web SDK from transforming the *web.config* file, use the **<IsTransformWebConfigDisabled>** property in the project file:

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

When disabling the Web SDK from transforming the file, the *processPath* and *arguments* should be manually set by the developer. For more information, see ASP.NET Core Module.

web.config file location

In order to set up the ASP.NET Core Module correctly, the *web.config* file must be present at the content root path (typically the app base path) of the deployed app. This is the same location as the website physical path provided to IIS. The *web.config* file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.

Sensitive files exist on the app's physical path, such as *<assembly>.runtimeconfig.json*, *<assembly>.xml* (XML Documentation comments), and *<assembly>.deps.json*. When the *web.config* file is present and the site starts normally, IIS doesn't serve these sensitive files if they're requested. If the *web.config* file is missing, incorrectly named, or unable to configure the site for normal startup, IIS may serve sensitive files publicly.

The *web.config* file must be present in the deployment at all times, correctly named, and able to configure the site for normal start up. Never remove the *web.config* file from a production deployment.

Transform web.config

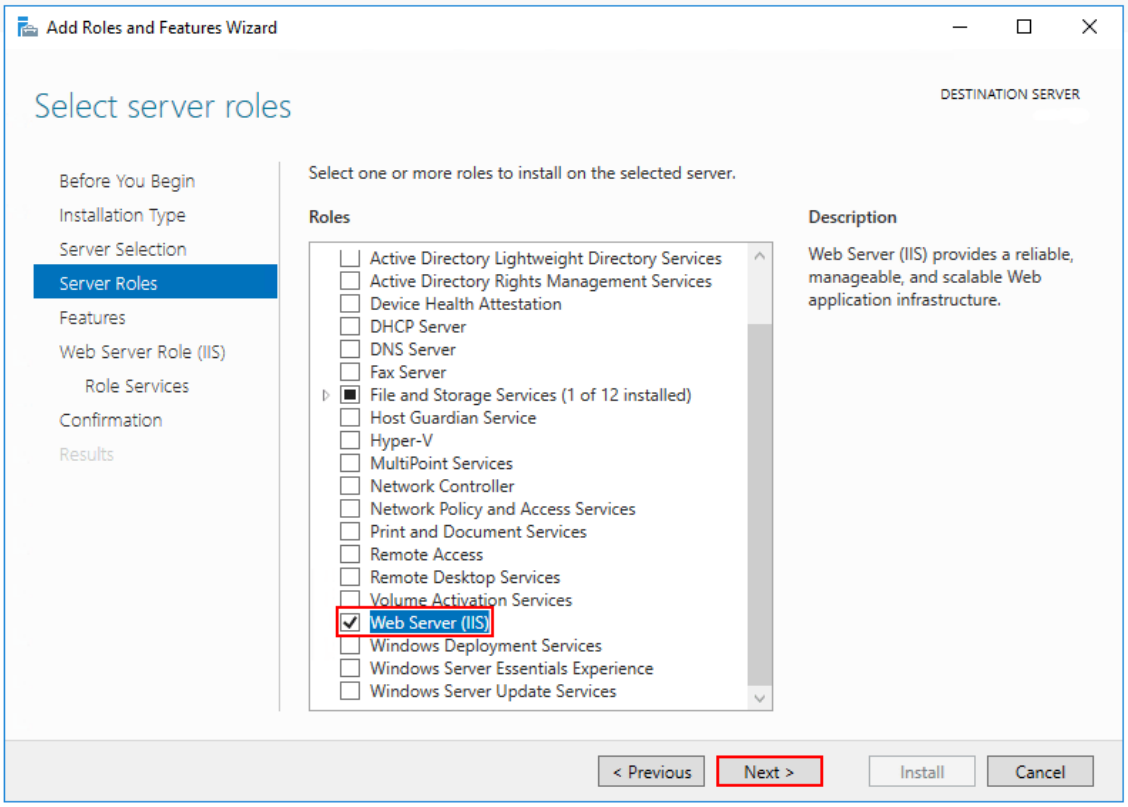
If you need to transform *web.config* on publish (for example, set environment variables based on the configuration, profile, or environment), see Transform web.config.

IIS configuration

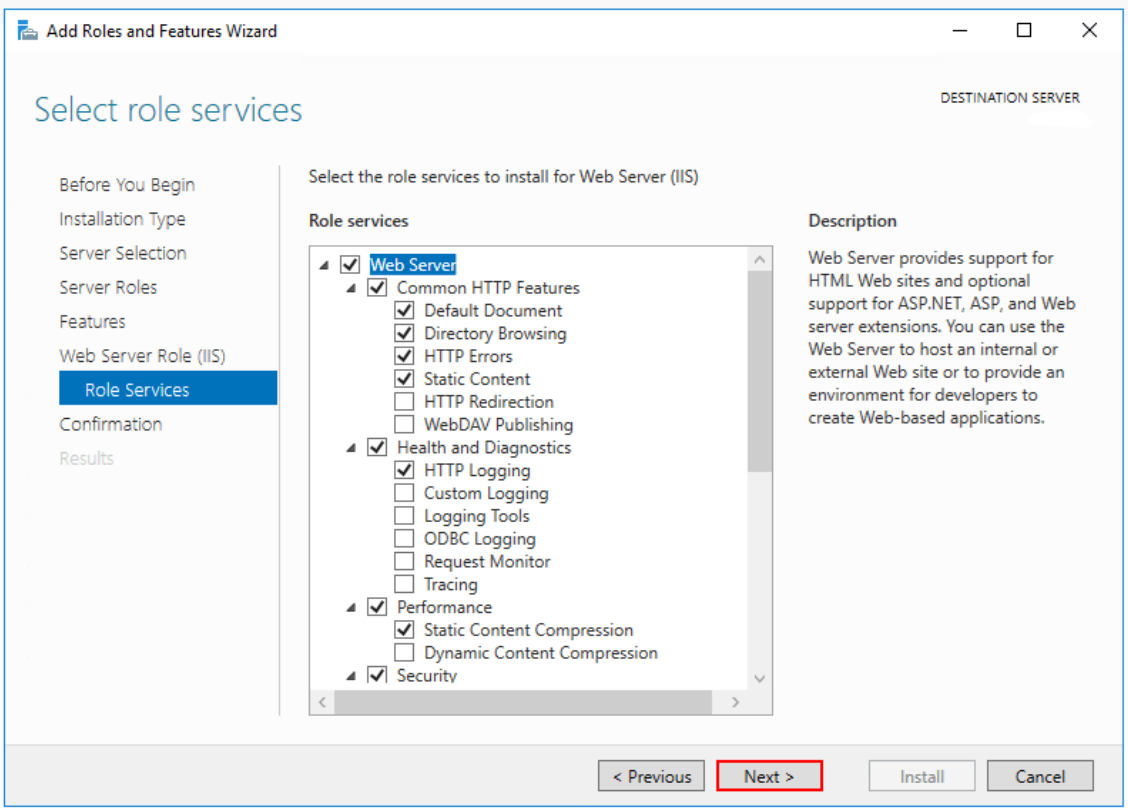
Windows Server operating systems

Enable the **Web Server (IIS)** server role and establish role services.

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



2. After the **Features** step, the **Role services** step loads for Web Server (IIS). Select the IIS role services desired or accept the default role services provided.



Windows Authentication (Optional)

To enable Windows Authentication, expand the following nodes: **Web Server** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

WebSockets (Optional)

WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **Web Server** > **Application Development**. Select the **WebSocket Protocol** feature. For

more information, see WebSockets.

3. Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart isn't required after installing the **Web Server (IIS)** role.

Windows desktop operating systems

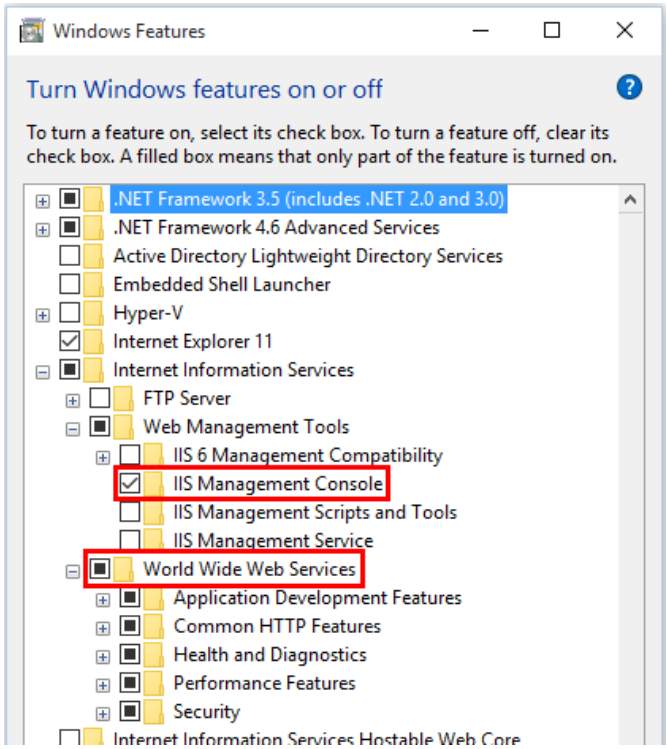
Enable the **IIS Management Console** and **World Wide Web Services**.

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the **Internet Information Services** node. Open the **Web Management Tools** node.
3. Check the box for **IIS Management Console**.
4. Check the box for **World Wide Web Services**.
5. Accept the default features for **World Wide Web Services** or customize the IIS features.

   Windows Authentication (Optional)
   To enable Windows Authentication, expand the following nodes: **World Wide Web Services** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

   WebSockets (Optional)
   WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **World Wide Web Services** > **Application Development Features**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

6. If the IIS installation requires a restart, restart the system.



Install the .NET Core Hosting Bundle

Install the *.NET Core Hosting Bundle* on the hosting system. The bundle installs the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. The module allows ASP.NET Core apps to run behind IIS.

> **IMPORTANT**
>
> If the Hosting Bundle is installed before IIS, the bundle installation must be repaired. Run the Hosting Bundle installer again after installing IIS.
>
> If the Hosting Bundle is installed after installing the 64-bit (x64) version of .NET Core, SDKs might appear to be missing (No .NET Core SDKs were detected). To resolve the problem, see Troubleshoot and debug ASP.NET Core projects.

Download

1. Navigate to the Download .NET Core page.
2. Select the desired .NET Core version.
3. In the **Run apps - Runtime** column, find the row of the .NET Core runtime version desired.
4. Download the installer using the **Hosting Bundle** link.

> **WARNING**
>
> Some installers contain release versions that have reached their end of life (EOL) and are no longer supported by Microsoft. For more information, see the support policy.

Install the Hosting Bundle

1. Run the installer on the server. The following parameters are available when running the installer from an administrator command shell:

   - `OPT_NO_ANCM=1`: Skip installing the ASP.NET Core Module.
   - `OPT_NO_RUNTIME=1`: Skip installing the .NET Core runtime. Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_SHAREDFX=1`: Skip installing the ASP.NET Shared Framework (ASP.NET runtime). Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_X86=1`: Skip installing x86 runtimes. Use this parameter when you know that you won't be hosting 32-bit apps. If there's any chance that you will host both 32-bit and 64-bit apps in the future, don't use this parameter and install both runtimes.
   - `OPT_NO_SHARED_CONFIG_CHECK=1`: Disable the check for using an IIS Shared Configuration when the shared configuration (*applicationHost.config*) is on the same machine as the IIS installation. *Only available for ASP.NET Core 2.2 or later Hosting Bundler installers.* For more information, see ASP.NET Core Module.

2. Restart the system or execute the following commands in a command shell:

   ```
   net stop was /y
   net start w3svc
   ```

   Restarting IIS picks up a change to the system PATH, which is an environment variable, made by the installer.

It isn't necessary to manually stop individual sites in IIS when installing the Hosting Bundle. Hosted apps (IIS sites) restart when IIS restarts. Apps start up again when they receive their first request, including from the Application Initialization Module.

ASP.NET Core adopts roll-forward behavior for patch releases of shared framework packages. When apps hosted by IIS restart with IIS, the apps load with the latest patch releases of their referenced packages when they receive their first request. If IIS isn't restarted, apps restart and exhibit roll-forward behavior when their worker processes are recycled and they receive their first request.

> **NOTE**
>
> For information on IIS Shared Configuration, see ASP.NET Core Module with IIS Shared Configuration.

Install Web Deploy when publishing with Visual Studio

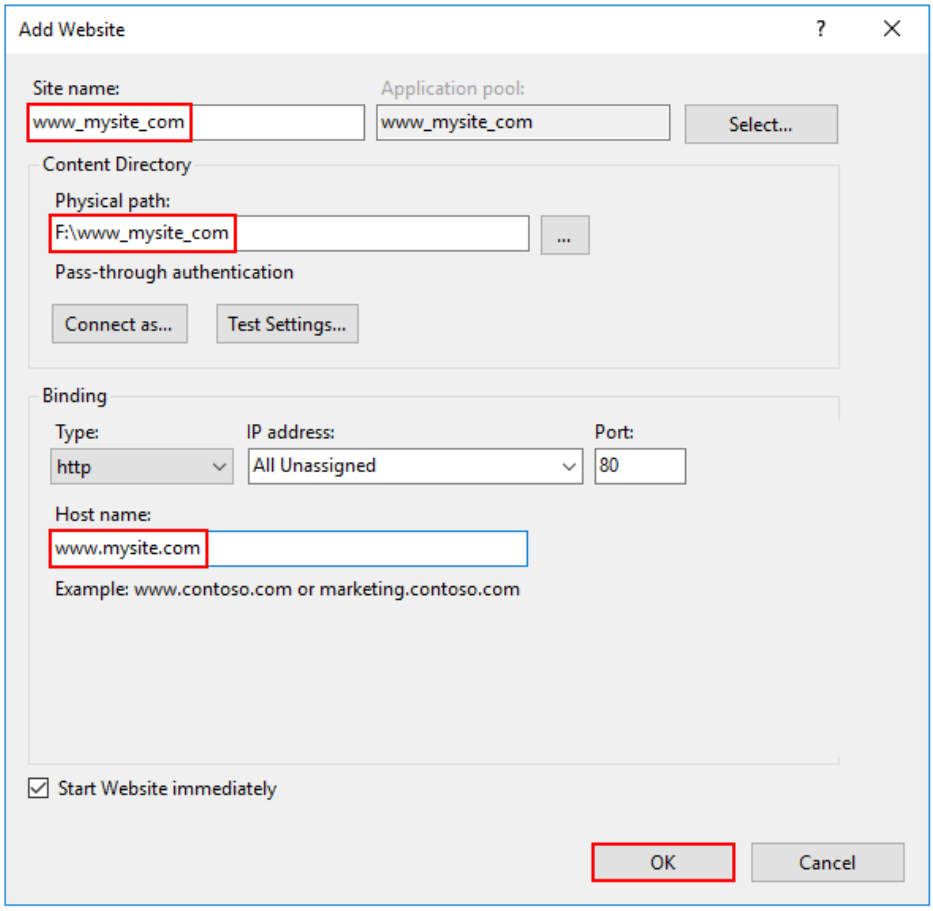
When deploying apps to servers with Web Deploy, install the latest version of Web Deploy on the server. To install Web Deploy, use the Web Platform Installer (WebPI) or obtain an installer directly from the Microsoft Download Center. The preferred method is to use WebPI. WebPI offers a standalone setup and a configuration for hosting providers.

Create the IIS site

1. On the hosting system, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see ASP.NET Core directory structure.

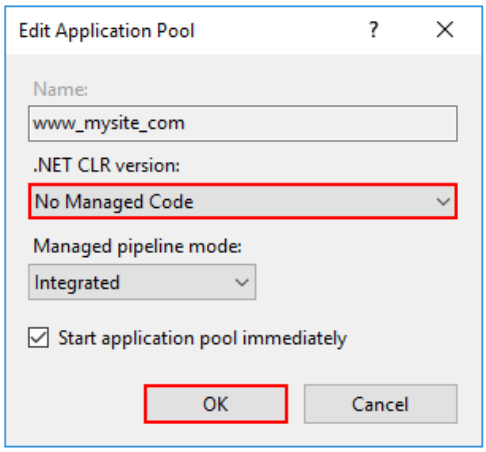
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.

3. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website by selecting **OK**:



> **WARNING**
>
> Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have this security risk if you control the entire parent domain (as opposed to `*.com` , which is vulnerable). See rfc7230 section-5.4 for more information.

4. Under the server's node, select **Application Pools**.
5. Right-click the site's app pool and select **Basic Settings** from the contextual menu.
6. In the **Edit Application Pool** window, set the **.NET CLR version** to **No Managed Code**:



   ASP.NET Core runs in a separate process and manages the runtime. ASP.NET Core doesn't rely on loading the desktop CLR (.NET CLR)—the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process. Setting the **.NET CLR version** to **No Managed Code** is optional but recommended.

7. *ASP.NET Core 2.2 or later*: For a 64-bit (x64) self-contained deployment that uses the in-process hosting model, disable the app pool for 32-bit (x86) processes.

   In the **Actions** sidebar of IIS Manager > **Application Pools**, select **Set Application Pool Defaults** or **Advanced Settings**. Locate **Enable 32-Bit Applications** and set the value to `False` . This setting doesn't affect apps deployed for out-of-process hosting.

8. Confirm the process model identity has the proper permissions.

   If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

Windows Authentication configuration (Optional)

For more information, see Configure Windows authentication.

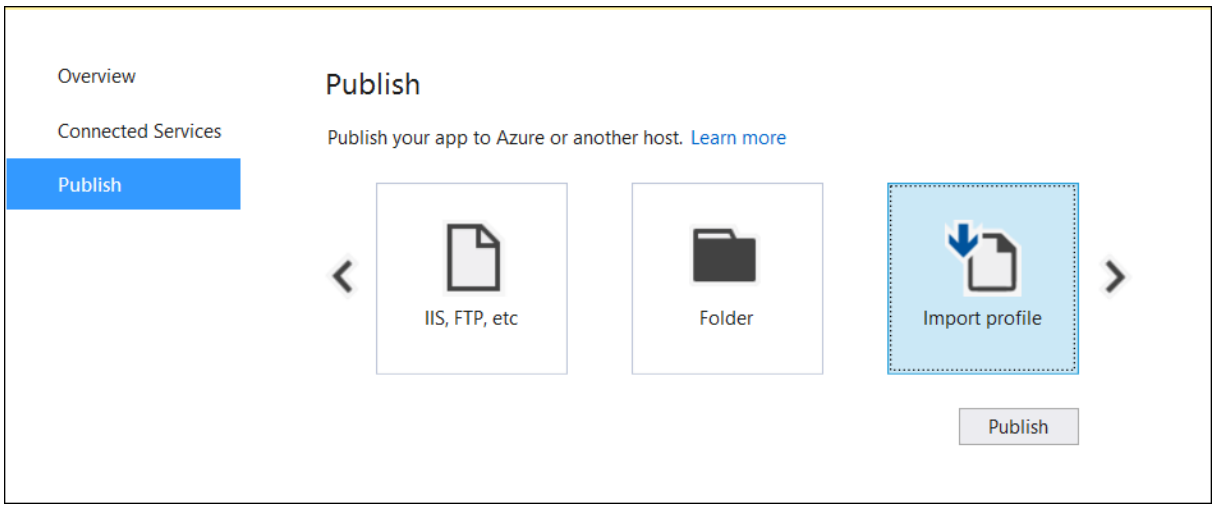
Deploy the app

Deploy the app to the IIS **Physical path** folder that was established in the Create the IIS site section. Web Deploy is the recommended mechanism for deployment, but several options exist for moving the app from the project's *publish* folder to the hosting system's deployment folder.

Web Deploy with Visual Studio

See the Visual Studio publish profiles for ASP.NET Core app deployment topic to learn how to create a publish

profile for use with Web Deploy. If the hosting provider provides a Publish Profile or support for creating one, download their profile and import it using the Visual Studio **Publish** dialog:



Web Deploy outside of Visual Studio

Web Deploy can also be used outside of Visual Studio from the command line. For more information, see Web Deployment Tool.

Alternatives to Web Deploy

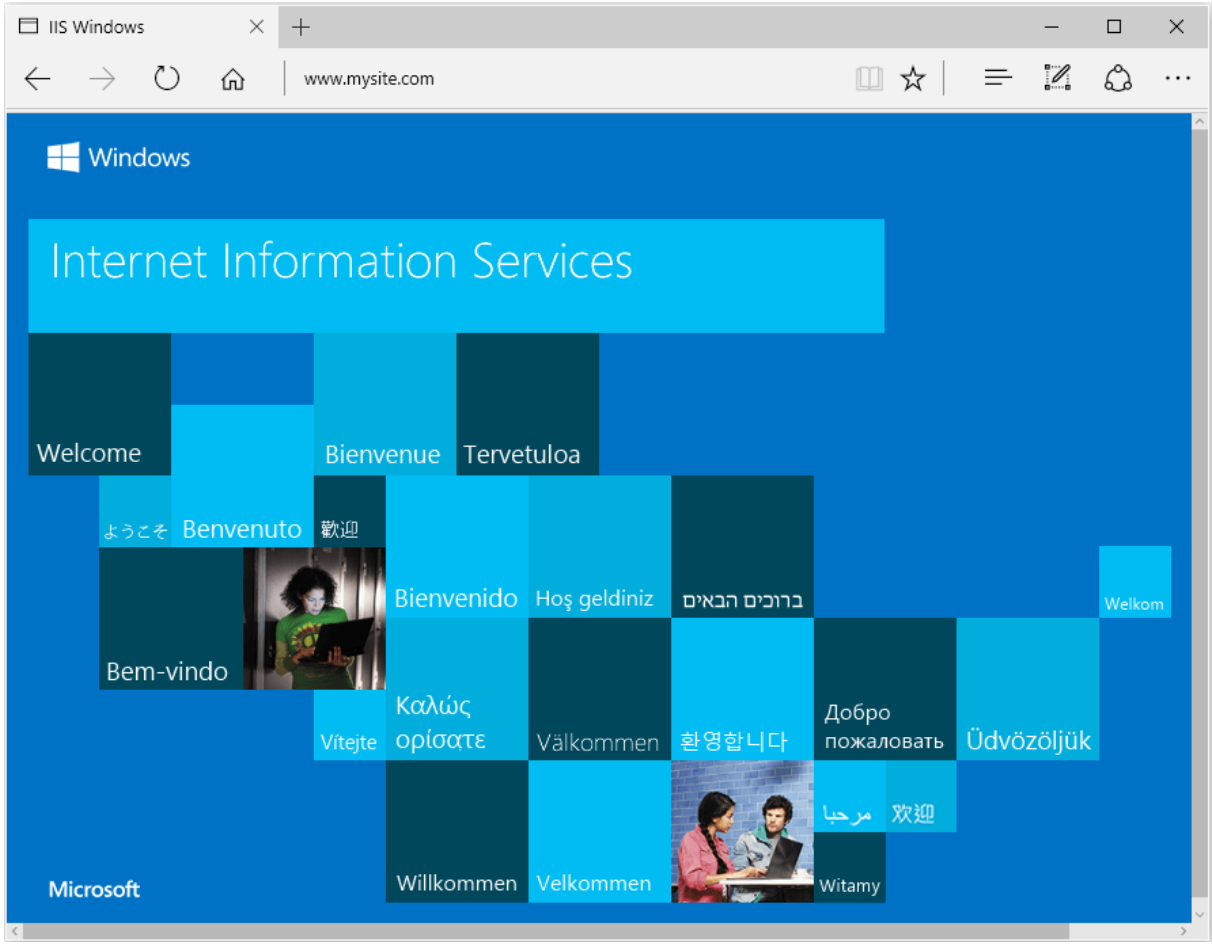
Use any of several methods to move the app to the hosting system, such as manual copy, Xcopy, Robocopy, or PowerShell.

For more information on ASP.NET Core deployment to IIS, see the Deployment resources for IIS administrators section.

Browse the website

After the app is deployed to the hosting system, make a request to one of the app's public endpoints.

In the following example, the site is bound to an IIS **Host name** of `www.mysite.com` on **Port** `80`. A request is made to `http://www.mysite.com`:



Locked deployment files

Files in the deployment folder are locked when the app is running. Locked files can't be overwritten during deployment. To release locked files in a deployment, stop the app pool using **one** of the following approaches:

- Use Web Deploy and reference `Microsoft.NET.Sdk.Web` in the project file. An *app_offline.htm* file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the *app_offline.htm* file during the deployment. For more information, see the ASP.NET Core Module configuration reference.
- Manually stop the app pool in the IIS Manager on the server.
- Use PowerShell to drop *app_offline.htm* (requires PowerShell 5 or later):

```
$pathToApp = 'PATH_TO_APP'

# Stop the AppPool
New-Item -Path $pathToApp app_offline.htm

# Provide script commands here to deploy the app

# Restart the AppPool
Remove-Item -Path $pathToApp app_offline.htm
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including middleware used in authentication. Even if Data Protection APIs aren't called by user code, data protection should be configured with a deployment script or in user code to create a persistent cryptographic key store. If data

protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection under IIS to persist the key ring, use **one** of the following approaches:

- **Create Data Protection Registry Keys**

  Data protection keys used by ASP.NET Core apps are stored in the registry external to the apps. To persist the keys for a given app, create registry keys for the app pool.

  For standalone, non-webfarm IIS installations, the Data Protection Provision-AutoGenKeys.ps1 PowerShell script can be used for each app pool used with an ASP.NET Core app. This script creates a registry key in the HKLM registry that's accessible only to the worker process account of the app's app pool. Keys are encrypted at rest using DPAPI with a machine-wide key.

  In web farm scenarios, an app can be configured to use a UNC path to store its data protection key ring. By default, the data protection keys aren't encrypted. Ensure that the file permissions for the network share are limited to the Windows account the app runs under. An X509 certificate can be used to protect keys at rest. Consider a mechanism to allow users to upload certificates: Place certificates into the user's trusted certificate store and ensure they're available on all machines where the user's app runs. See Configure ASP.NET Core Data Protection for details.

- **Configure the IIS Application Pool to load the user profile**

  This setting is in the **Process Model** section under the **Advanced Settings** for the app pool. Set **Load User Profile** to `True`. When set to `True`, keys are stored in the user profile directory and protected using DPAPI with a key specific to the user account. Keys are persisted to the *%LOCALAPPDATA%/ASP.NET/DataProtection-Keys* folder.

  The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:

  1. Navigate to the *%windir%/system32/inetsrv/config* folder.
  2. Open the *applicationHost.config* file.
  3. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.
  4. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

- **Use the file system as a key ring store**

  Adjust the app code to use the file system as a key ring store. Use an X509 certificate to protect the key ring and ensure the certificate is a trusted certificate. If the certificate is self-signed, place the certificate in the Trusted Root store.

  When using IIS in a web farm:

  - Use a file share that all machines can access.
  - Deploy an X509 certificate to each machine. Configure data protection in code.

- **Set a machine-wide policy for data protection**

  The data protection system has limited support for setting a default machine-wide policy for all apps

that consume the Data Protection APIs. For more information, see ASP.NET Core Data Protection.

Virtual Directories

IIS Virtual Directories aren't supported with ASP.NET Core apps. An app can be hosted as a sub-application.

Sub-applications

An ASP.NET Core app can be hosted as an IIS sub-application (sub-app). The sub-app's path becomes part of the root app's URL.

Static asset links within the sub-app should use tilde-slash (`~/`) notation. Tilde-slash notation triggers a Tag Helper to prepend the sub-app's pathbase to the rendered relative link. For a sub-app at `/subapp_path`, an image linked with `src="~/image.png"` is rendered as `src="/subapp_path/image.png"`. The root app's Static File Middleware doesn't process the static file request. The request is processed by the sub-app's Static File Middleware.

If a static asset's `src` attribute is set to an absolute path (for example, `src="/image.png"`), the link is rendered without the sub-app's pathbase. The root app's Static File Middleware attempts to serve the asset from the root app's web root, which results in a *404 - Not Found* response unless the static asset is available from the root app.

To host an ASP.NET Core app as a sub-app under another ASP.NET Core app:

1. Establish an app pool for the sub-app. Set the **.NET CLR Version** to **No Managed Code** because the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process, not the desktop CLR (.NET CLR).
2. Add the root site in IIS Manager with the sub-app in a folder under the root site.
3. Right-click the sub-app folder in IIS Manager and select **Convert to Application**.
4. In the **Add Application** dialog, use the **Select** button for the **Application Pool** to assign the app pool that you created for the sub-app. Select **OK**.

The assignment of a separate app pool to the sub-app is a requirement when using the in-process hosting model.

For more information on the in-process hosting model and configuring the ASP.NET Core Module, see ASP.NET Core Module.

Configuration of IIS with web.config

IIS configuration is influenced by the `<system.webServer>` section of *web.config* for IIS scenarios that are functional for ASP.NET Core apps with the ASP.NET Core Module. For example, IIS configuration is functional for dynamic compression. If IIS is configured at the server level to use dynamic compression, the `<urlCompression>` element in the app's *web.config* file can disable it for an ASP.NET Core app.

For more information, see the following topics:

- Configuration reference for <system.webServer>
- ASP.NET Core Module
- IIS modules with ASP.NET Core

To set environment variables for individual apps running in isolated app pools (supported for IIS 10.0 or later), see the *AppCmd.exe command* section of the Environment Variables <environmentVariables> topic in the IIS reference documentation.

Configuration sections of web.config

Configuration sections of ASP.NET 4.x apps in *web.config* aren't used by ASP.NET Core apps for configuration:

- `<system.web>`
- `<appSettings>`
- `<connectionStrings>`
- `<location>`

ASP.NET Core apps are configured using other configuration providers. For more information, see Configuration.

Application Pools

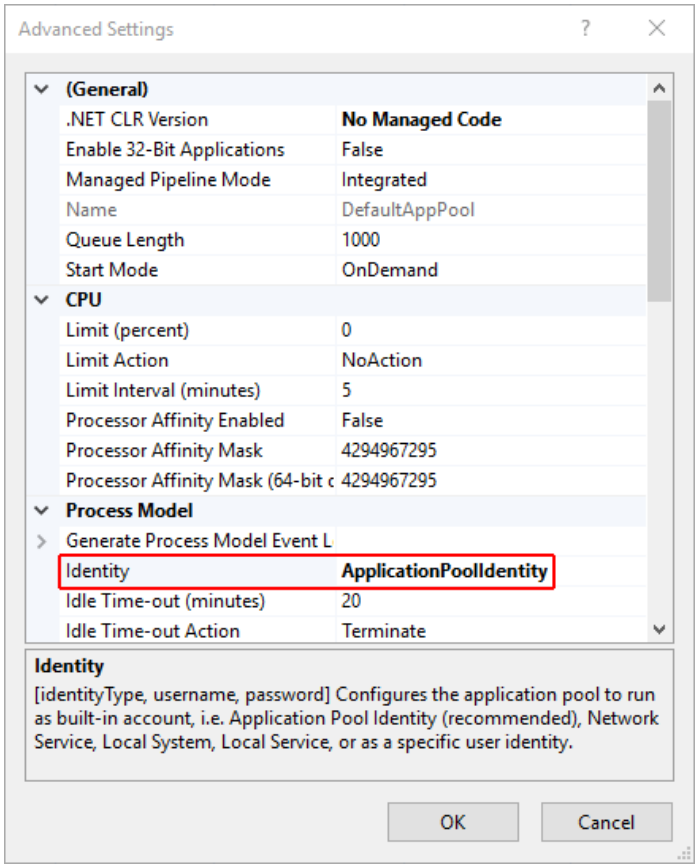
App pool isolation is determined by the hosting model:

- In-process hosting: Apps are required to run in separate app pools.
- Out-of-process hosting: We recommend isolating the apps from each other by running each app in its own app pool.

The IIS **Add Website** dialog defaults to a single app pool per app. When a **Site name** is provided, the text is automatically transferred to the **Application pool** textbox. A new app pool is created using the site name when the site is added.

Application Pool Identity

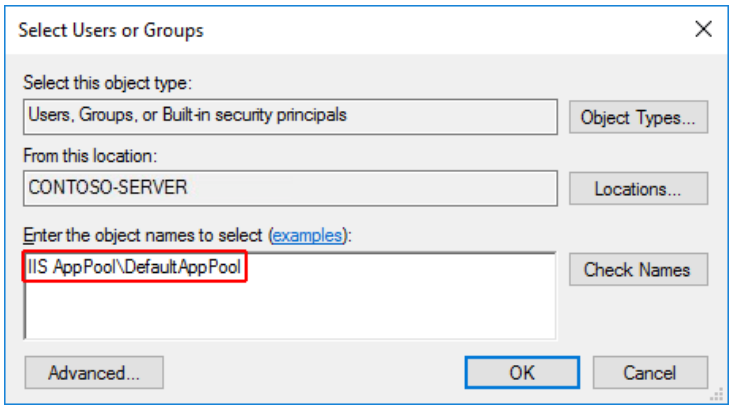
An app pool identity account allows an app to run under a unique account without having to create and manage domains or local accounts. On IIS 8.0 or later, the IIS Admin Worker Process (WAS) creates a virtual account with the name of the new app pool and runs the app pool's worker processes under this account by default. In the IIS Management Console under **Advanced Settings** for the app pool, ensure that the **Identity** is set to use **ApplicationPoolIdentity**:



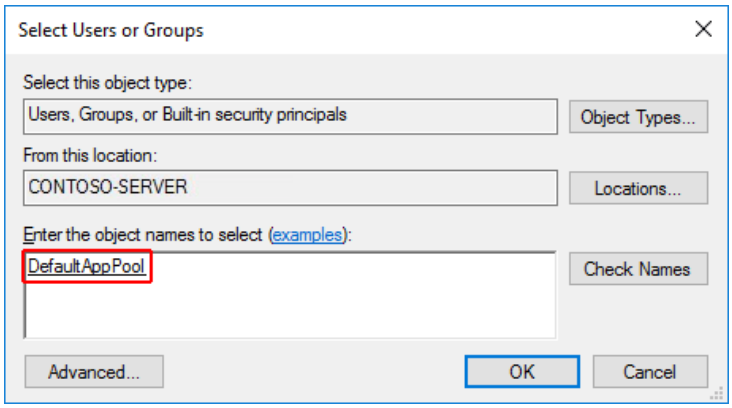
The IIS management process creates a secure identifier with the name of the app pool in the Windows Security System. Resources can be secured using this identity. However, this identity isn't a real user account and doesn't show up in the Windows User Management Console.

If the IIS worker process requires elevated access to the app, modify the Access Control List (ACL) for the directory containing the app:

1. Open Windows Explorer and navigate to the directory.
2. Right-click on the directory and select **Properties**.
3. Under the **Security** tab, select the **Edit** button and then the **Add** button.
4. Select the **Locations** button and make sure the system is selected.
5. Enter **IIS AppPool\<app_pool_name>** in **Enter the object names to select** area. Select the **Check Names** button. For the *DefaultAppPool* check the names using **IIS AppPool\DefaultAppPool**. When the **Check Names** button is selected, a value of **DefaultAppPool** is indicated in the object names area. It isn't possible to enter the app pool name directly into the object names area. Use the **IIS AppPool\<app_pool_name>** format when checking for the object name.



6. Select **OK**.



7. Read & execute permissions should be granted by default. Provide additional permissions as needed.

Access can also be granted at a command prompt using the **ICACLS** tool. Using the *DefaultAppPool* as an example, the following command is used:

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool":F
```

For more information, see the icacls topic.

HTTP/2 support

HTTP/2 is supported with ASP.NET Core in the following IIS deployment scenarios:

- In-process
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - TLS 1.2 or later connection
- Out-of-process
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Public-facing edge server connections use HTTP/2, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.
  - TLS 1.2 or later connection

For an in-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`. For an out-of-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/1.1`.

For more information on the in-process and out-of-process hosting models, see ASP.NET Core Module.

HTTP/2 is enabled by default. Connections fall back to HTTP/1.1 if an HTTP/2 connection isn't established. For more information on HTTP/2 configuration with IIS deployments, see HTTP/2 on IIS.

CORS preflight requests

This section only applies to ASP.NET Core apps that target the .NET Framework.

For an ASP.NET Core app that targets the .NET Framework, OPTIONS requests aren't passed to the app by default in IIS. To learn how to configure the app's IIS handlers in *web.config* to pass OPTIONS requests, see Enable cross-origin requests in ASP.NET Web API 2: How CORS Works.

Application Initialization Module and Idle Timeout

When hosted in IIS by the ASP.NET Core Module version 2:

- Application Initialization Module: App's hosted in-process or out-of-process can be configured to start automatically on a worker process restart or server restart.
- Idle Timeout: App's hosted in-process can be configured not to timeout during periods of inactivity.

Application Initialization Module

Applies to apps hosted in-process and out-of-process.

IIS Application Initialization is an IIS feature that sends an HTTP request to the app when the app pool starts or is recycled. The request triggers the app to start. By default, IIS issues a request to the app's root URL (`/`) to initialize the app (see the additional resources for more details on configuration).

Confirm that the IIS Application Initialization role feature in enabled:

On Windows 7 or later desktop systems when using IIS locally:

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open **Internet Information Services** > **World Wide Web Services** > **Application Development Features**.
3. Select the check box for **Application Initialization**.

On Windows Server 2008 R2 or later:

1. Open the **Add Roles and Features Wizard**.
2. In the **Select role services** panel, open the **Application Development** node.

3. Select the check box for **Application Initialization**.

Use either of the following approaches to enable the Application Initialization Module for the site:

- Using IIS Manager:

  1. Select **Application Pools** in the **Connections** panel.
  2. Right-click the app's app pool in the list and select **Advanced Settings**.
  3. The default **Start Mode** is **OnDemand**. Set the **Start Mode** to **AlwaysRunning**. Select **OK**.
  4. Open the **Sites** node in the **Connections** panel.
  5. Right-click the app and select **Manage Website** > **Advanced Settings**.
  6. The default **Preload Enabled** setting is **False**. Set **Preload Enabled** to **True**. Select **OK**.

- Using *web.config*, add the `<applicationInitialization>` element with `doAppInitAfterRestart` set to `true` to the `<system.webServer>` elements in the app's *web.config* file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <applicationInitialization doAppInitAfterRestart="true" />
    </system.webServer>
  </location>
</configuration>
```

Idle Timeout

Only applies to apps hosted in-process.

To prevent the app from idling, set the app pool's idle timeout using IIS Manager:

1. Select **Application Pools** in the **Connections** panel.
2. Right-click the app's app pool in the list and select **Advanced Settings**.
3. The default **Idle Time-out (minutes)** is **20** minutes. Set the **Idle Time-out (minutes)** to **0** (zero). Select **OK**.
4. Recycle the worker process.

To prevent apps hosted out-of-process from timing out, use either of the following approaches:

- Ping the app from an external service in order to keep it running.
- If the app only hosts background services, avoid IIS hosting and use a Windows Service to host the ASP.NET Core app.

Application Initialization Module and Idle Timeout additional resources

- IIS 8.0 Application Initialization
- Application Initialization <applicationInitialization>.
- Process Model Settings for an Application Pool <processModel>.

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS

- Common errors reference for Azure App Service and IIS with ASP.NET Core

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Introduction to ASP.NET Core
- The Official Microsoft IIS Site
- Windows Server technical content library
- HTTP/2 on IIS
- Transform web.config

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see Publish an ASP.NET Core app to IIS.

Install the .NET Core Hosting Bundle

Supported operating systems

The following operating systems are supported:

- Windows 7 or later
- Windows Server 2008 R2 or later

HTTP.sys server (formerly called WebListener) doesn't work in a reverse proxy configuration with IIS. Use the Kestrel server.

For information on hosting in Azure, see Deploy ASP.NET Core apps to Azure App Service.

For troubleshooting guidance, see Troubleshoot and debug ASP.NET Core projects.

Supported platforms

Apps published for 32-bit (x86) or 64-bit (x64) deployment are supported. Deploy a 32-bit app with a 32-bit (x86) .NET Core SDK unless the app:

- Requires the larger virtual memory address space available to a 64-bit app.
- Requires the larger IIS stack size.
- Has 64-bit native dependencies.

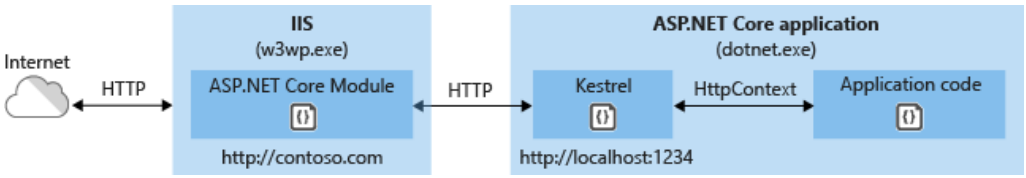
Use a 64-bit (x64) .NET Core SDK to publish a 64-bit app. A 64-bit runtime must be present on the host system.

ASP.NET Core ships with Kestrel server, a default, cross-platform HTTP server.

When using IIS or IIS Express, the app runs in a process separate from the IIS worker process (*out-of-process*) with the Kestrel server.

Because ASP.NET Core apps run in a process separate from the IIS worker process, the module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the IIS Integration Middleware configures the server to listen on `http://localhost:{port}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.

`CreateDefaultBuilder` configures Kestrel server as the web server and enables IIS Integration by configuring the base path and port for the ASP.NET Core Module.

The ASP.NET Core Module generates a dynamic port to assign to the backend process. `CreateDefaultBuilder` calls the UseIISIntegration method. `UseIISIntegration` configures Kestrel to listen on the dynamic port at the localhost IP address (`127.0.0.1`). If the dynamic port is 1234, Kestrel listens at `127.0.0.1:1234`. This configuration replaces other URL configurations provided by:

- `UseUrls`
- Kestrel's Listen API
- Configuration (or command-line --urls option)

Calls to `UseUrls` or Kestrel's `Listen` API aren't required when using the module. If `UseUrls` or `Listen` is called, Kestrel listens on the port specified only when running the app without IIS.

For ASP.NET Core Module configuration guidance, see ASP.NET Core Module.

For more information on hosting, see Host in ASP.NET Core.

Application configuration

Enable the IISIntegration components

When building a host in `CreateWebHostBuilder` (*Program.cs*), call CreateDefaultBuilder to enable IIS integration:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        ...
```

For more information on `CreateDefaultBuilder`, see ASP.NET Core Web Host.

IIS options

OPTION	DEFAULT	SETTING

OPTION	DEFAULT	SETTING
`AutomaticAuthentication`	`true`	If `true`, IIS Server sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the server only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see Windows Authentication.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.

To configure IIS options, include a service configuration for IISOptions in ConfigureServices. The following example prevents the app from populating `HttpContext.Connection.ClientCertificate`:

```
services.Configure<IISOptions>(options =>
{
    options.ForwardClientCertificate = false;
});
```

OPTION	DEFAULT	SETTING
`AutomaticAuthentication`	`true`	If `true`, IIS Integration Middleware sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the middleware only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see the Windows Authentication topic.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`ForwardClientCertificate`	`true`	If `true` and the `MS-ASPNETCORE-CLIENTCERT` request header is present, the `HttpContext.Connection.ClientCertificate` is populated.

Proxy server and load balancer scenarios

The IIS Integration Middleware, which configures Forwarded Headers Middleware, and the ASP.NET Core Module are configured to forward the scheme (HTTP/HTTPS) and the remote IP address where the request originated. Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

web.config file

The *web.config* file configures the ASP.NET Core Module. Creating, transforming, and publishing the *web.config* file is handled by an MSBuild target (`_TransformWebConfig`) when the project is published. This target is present in the Web SDK targets (`Microsoft.NET.Sdk.Web`). The SDK is set at the top of the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

If a *web.config* file isn't present in the project, the file is created with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output.

If a *web.config* file is present in the project, the file is transformed with the correct *processPath* and *arguments* to configure the ASP.NET Core Module and moved to published output. The transformation doesn't modify IIS configuration settings in the file.

The *web.config* file may provide additional IIS configuration settings that control active IIS modules. For information on IIS modules that are capable of processing requests with ASP.NET Core apps, see the IIS modules topic.

To prevent the Web SDK from transforming the *web.config* file, use the **<IsTransformWebConfigDisabled>** property in the project file:

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

When disabling the Web SDK from transforming the file, the *processPath* and *arguments* should be manually set by the developer. For more information, see ASP.NET Core Module.

web.config file location

In order to set up the ASP.NET Core Module correctly, the *web.config* file must be present at the content root path (typically the app base path) of the deployed app. This is the same location as the website physical path provided to IIS. The *web.config* file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.

Sensitive files exist on the app's physical path, such as *<assembly>.runtimeconfig.json*, *<assembly>.xml* (XML Documentation comments), and *<assembly>.deps.json*. When the *web.config* file is present and the site starts normally, IIS doesn't serve these sensitive files if they're requested. If the *web.config* file is missing, incorrectly named, or unable to configure the site for normal startup, IIS may serve sensitive files publicly.

The *web.config* file must be present in the deployment at all times, correctly named, and able to configure the site for normal start up. Never remove the *web.config* file from a production deployment.

Transform web.config

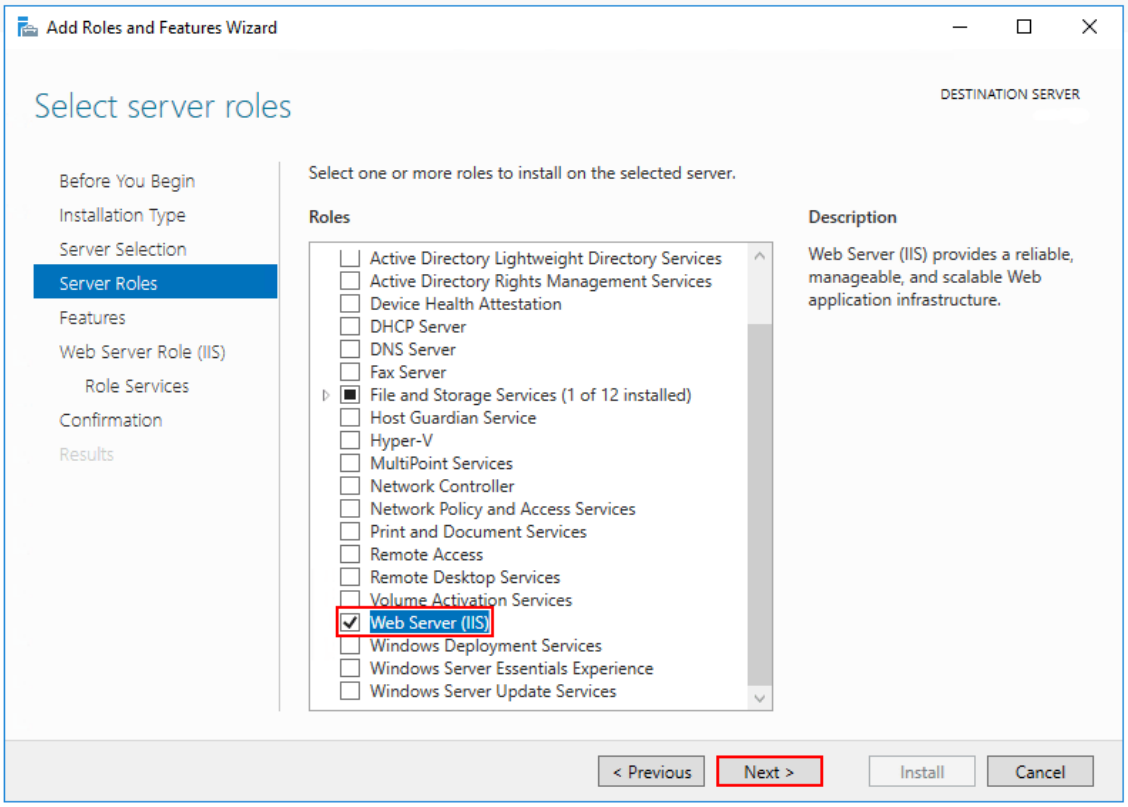
If you need to transform *web.config* on publish (for example, set environment variables based on the configuration, profile, or environment), see Transform web.config.

IIS configuration

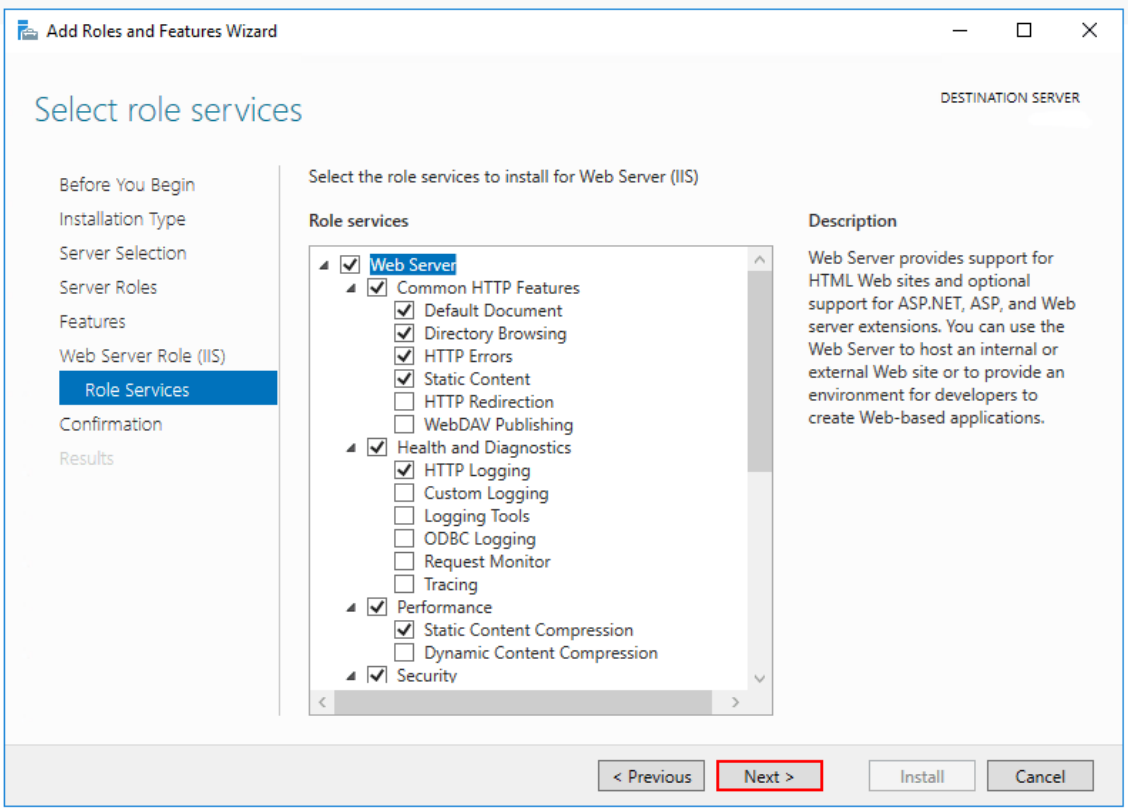
Windows Server operating systems

Enable the **Web Server (IIS)** server role and establish role services.

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



2. After the **Features** step, the **Role services** step loads for Web Server (IIS). Select the IIS role services desired or accept the default role services provided.



Windows Authentication (Optional)

To enable Windows Authentication, expand the following nodes: **Web Server** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

WebSockets (Optional)

WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **Web Server** > **Application Development**. Select the **WebSocket Protocol** feature. For

more information, see WebSockets.

3. Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart isn't required after installing the **Web Server (IIS)** role.

Windows desktop operating systems

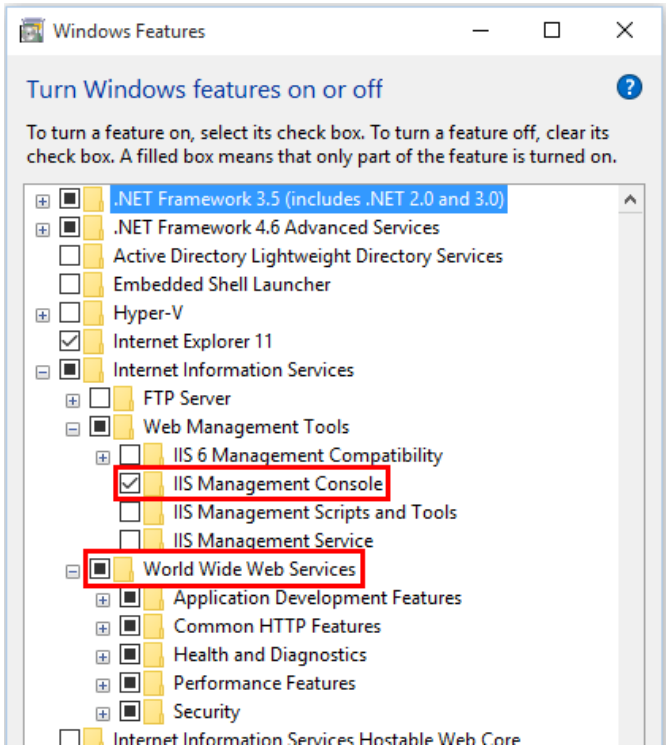
Enable the **IIS Management Console** and **World Wide Web Services**.

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the **Internet Information Services** node. Open the **Web Management Tools** node.
3. Check the box for **IIS Management Console**.
4. Check the box for **World Wide Web Services**.
5. Accept the default features for **World Wide Web Services** or customize the IIS features.

   Windows Authentication (Optional)
   To enable Windows Authentication, expand the following nodes: **World Wide Web Services** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

   WebSockets (Optional)
   WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **World Wide Web Services** > **Application Development Features**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

6. If the IIS installation requires a restart, restart the system.



Install the .NET Core Hosting Bundle

Install the *.NET Core Hosting Bundle* on the hosting system. The bundle installs the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. The module allows ASP.NET Core apps to run behind IIS.

> **IMPORTANT**
>
> If the Hosting Bundle is installed before IIS, the bundle installation must be repaired. Run the Hosting Bundle installer again after installing IIS.
>
> If the Hosting Bundle is installed after installing the 64-bit (x64) version of .NET Core, SDKs might appear to be missing (No .NET Core SDKs were detected). To resolve the problem, see Troubleshoot and debug ASP.NET Core projects.

Download

1. Navigate to the Download .NET Core page.
2. Select the desired .NET Core version.
3. In the **Run apps - Runtime** column, find the row of the .NET Core runtime version desired.
4. Download the installer using the **Hosting Bundle** link.

> **WARNING**
>
> Some installers contain release versions that have reached their end of life (EOL) and are no longer supported by Microsoft. For more information, see the support policy.

Install the Hosting Bundle

1. Run the installer on the server. The following parameters are available when running the installer from an administrator command shell:

   - `OPT_NO_ANCM=1`: Skip installing the ASP.NET Core Module.
   - `OPT_NO_RUNTIME=1`: Skip installing the .NET Core runtime. Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_SHAREDFX=1`: Skip installing the ASP.NET Shared Framework (ASP.NET runtime). Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_X86=1`: Skip installing x86 runtimes. Use this parameter when you know that you won't be hosting 32-bit apps. If there's any chance that you will host both 32-bit and 64-bit apps in the future, don't use this parameter and install both runtimes.
   - `OPT_NO_SHARED_CONFIG_CHECK=1`: Disable the check for using an IIS Shared Configuration when the shared configuration (*applicationHost.config*) is on the same machine as the IIS installation. *Only available for ASP.NET Core 2.2 or later Hosting Bundler installers.* For more information, see ASP.NET Core Module.

2. Restart the system or execute the following commands in a command shell:

   ```
   net stop was /y
   net start w3svc
   ```

   Restarting IIS picks up a change to the system PATH, which is an environment variable, made by the installer.

It isn't necessary to manually stop individual sites in IIS when installing the Hosting Bundle. Hosted apps (IIS sites) restart when IIS restarts. Apps start up again when they receive their first request, including from the Application Initialization Module.

ASP.NET Core adopts roll-forward behavior for patch releases of shared framework packages. When apps hosted by IIS restart with IIS, the apps load with the latest patch releases of their referenced packages when they receive their first request. If IIS isn't restarted, apps restart and exhibit roll-forward behavior when their worker processes are recycled and they receive their first request.

> **NOTE**
>
> For information on IIS Shared Configuration, see ASP.NET Core Module with IIS Shared Configuration.

Install Web Deploy when publishing with Visual Studio

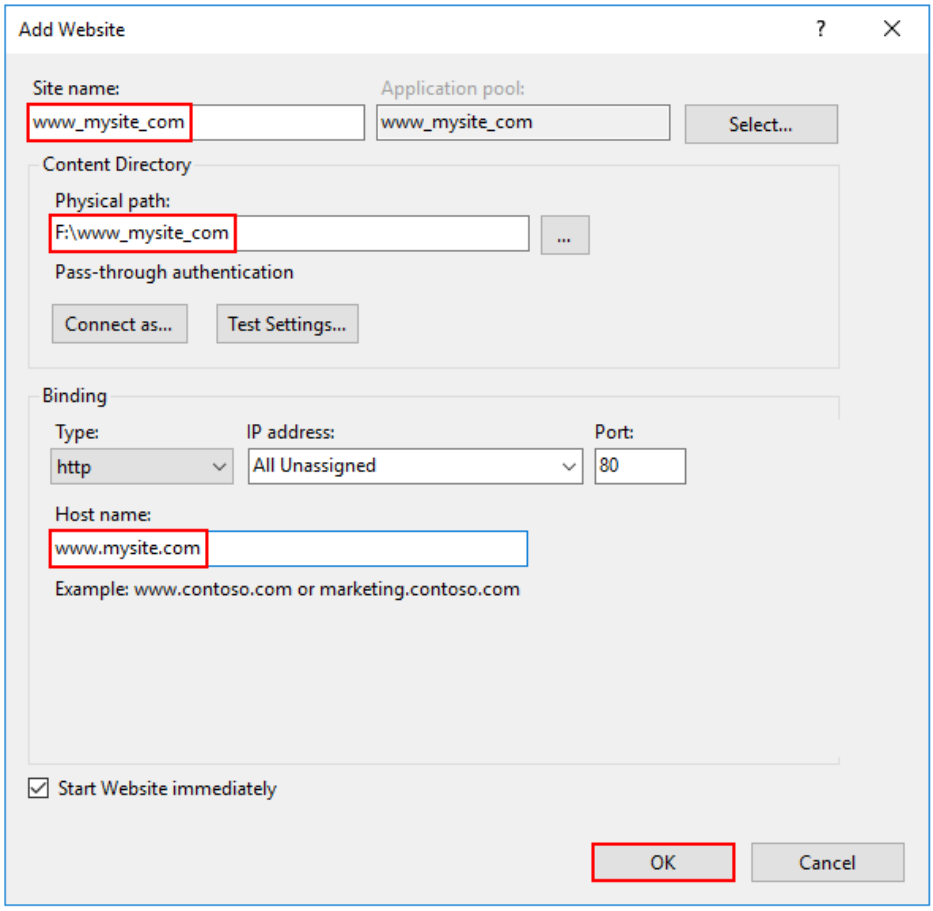
When deploying apps to servers with Web Deploy, install the latest version of Web Deploy on the server. To install Web Deploy, use the Web Platform Installer (WebPI) or obtain an installer directly from the Microsoft Download Center. The preferred method is to use WebPI. WebPI offers a standalone setup and a configuration for hosting providers.

Create the IIS site

1. On the hosting system, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see ASP.NET Core directory structure.

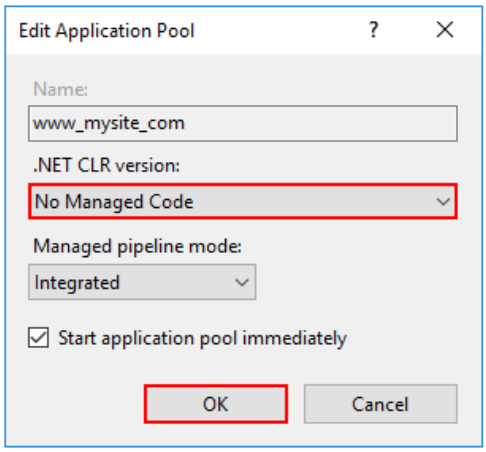
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.

3. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website by selecting **OK**:



> **WARNING**
>
> Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have this security risk if you control the entire parent domain (as opposed to `*.com` , which is vulnerable). See rfc7230 section-5.4 for more information.

4. Under the server's node, select **Application Pools**.
5. Right-click the site's app pool and select **Basic Settings** from the contextual menu.
6. In the **Edit Application Pool** window, set the **.NET CLR version** to **No Managed Code**:



   ASP.NET Core runs in a separate process and manages the runtime. ASP.NET Core doesn't rely on loading the desktop CLR (.NET CLR)—the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process. Setting the **.NET CLR version** to **No Managed Code** is optional but recommended.

7. *ASP.NET Core 2.2 or later*: For a 64-bit (x64) self-contained deployment that uses the in-process hosting model, disable the app pool for 32-bit (x86) processes.

   In the **Actions** sidebar of IIS Manager > **Application Pools**, select **Set Application Pool Defaults** or **Advanced Settings**. Locate **Enable 32-Bit Applications** and set the value to `False` . This setting doesn't affect apps deployed for out-of-process hosting.

8. Confirm the process model identity has the proper permissions.

   If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

Windows Authentication configuration (Optional)

For more information, see Configure Windows authentication.

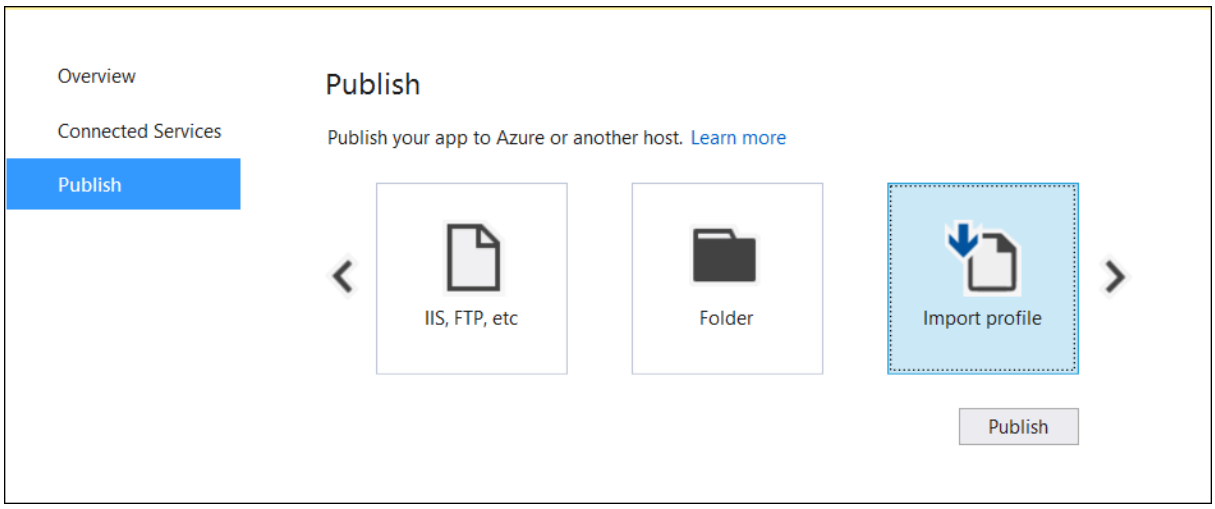
Deploy the app

Deploy the app to the IIS **Physical path** folder that was established in the Create the IIS site section. Web Deploy is the recommended mechanism for deployment, but several options exist for moving the app from the project's *publish* folder to the hosting system's deployment folder.

Web Deploy with Visual Studio

See the Visual Studio publish profiles for ASP.NET Core app deployment topic to learn how to create a publish

profile for use with Web Deploy. If the hosting provider provides a Publish Profile or support for creating one, download their profile and import it using the Visual Studio **Publish** dialog:



Web Deploy outside of Visual Studio

Web Deploy can also be used outside of Visual Studio from the command line. For more information, see Web Deployment Tool.

Alternatives to Web Deploy

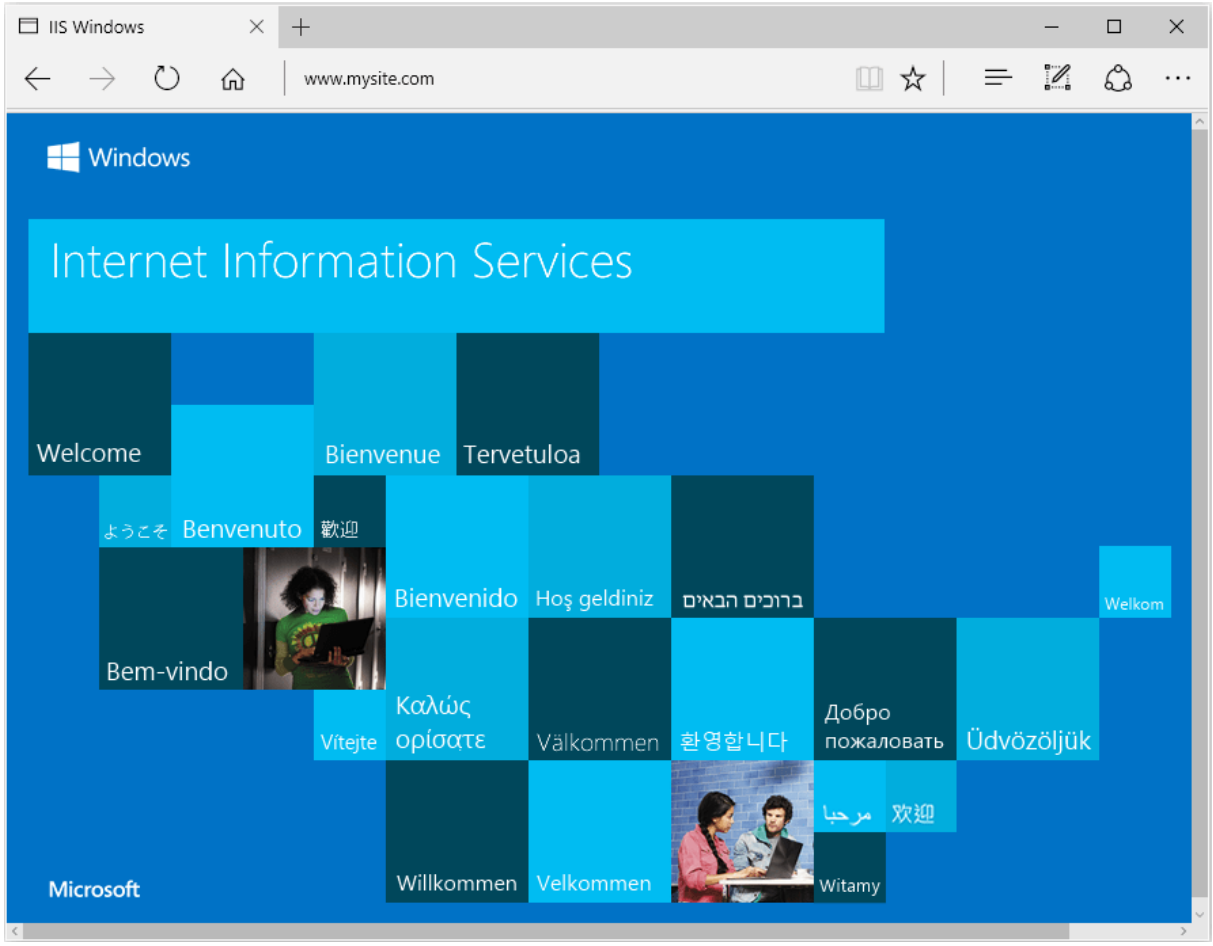
Use any of several methods to move the app to the hosting system, such as manual copy, Xcopy, Robocopy, or PowerShell.

For more information on ASP.NET Core deployment to IIS, see the Deployment resources for IIS administrators section.

Browse the website

After the app is deployed to the hosting system, make a request to one of the app's public endpoints.

In the following example, the site is bound to an IIS **Host name** of `www.mysite.com` on **Port** `80`. A request is made to `http://www.mysite.com`:



Locked deployment files

Files in the deployment folder are locked when the app is running. Locked files can't be overwritten during deployment. To release locked files in a deployment, stop the app pool using **one** of the following approaches:

- Use Web Deploy and reference `Microsoft.NET.Sdk.Web` in the project file. An *app_offline.htm* file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the *app_offline.htm* file during the deployment. For more information, see the ASP.NET Core Module configuration reference.
- Manually stop the app pool in the IIS Manager on the server.
- Use PowerShell to drop *app_offline.htm* (requires PowerShell 5 or later):

```
$pathToApp = 'PATH_TO_APP'

# Stop the AppPool
New-Item -Path $pathToApp app_offline.htm

# Provide script commands here to deploy the app

# Restart the AppPool
Remove-Item -Path $pathToApp app_offline.htm
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including middleware used in authentication. Even if Data Protection APIs aren't called by user code, data protection should be configured with a deployment script or in user code to create a persistent cryptographic key store. If data

protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection under IIS to persist the key ring, use **one** of the following approaches:

- **Create Data Protection Registry Keys**

  Data protection keys used by ASP.NET Core apps are stored in the registry external to the apps. To persist the keys for a given app, create registry keys for the app pool.

  For standalone, non-webfarm IIS installations, the Data Protection Provision-AutoGenKeys.ps1 PowerShell script can be used for each app pool used with an ASP.NET Core app. This script creates a registry key in the HKLM registry that's accessible only to the worker process account of the app's app pool. Keys are encrypted at rest using DPAPI with a machine-wide key.

  In web farm scenarios, an app can be configured to use a UNC path to store its data protection key ring. By default, the data protection keys aren't encrypted. Ensure that the file permissions for the network share are limited to the Windows account the app runs under. An X509 certificate can be used to protect keys at rest. Consider a mechanism to allow users to upload certificates: Place certificates into the user's trusted certificate store and ensure they're available on all machines where the user's app runs. See Configure ASP.NET Core Data Protection for details.

- **Configure the IIS Application Pool to load the user profile**

  This setting is in the **Process Model** section under the **Advanced Settings** for the app pool. Set **Load User Profile** to `True`. When set to `True`, keys are stored in the user profile directory and protected using DPAPI with a key specific to the user account. Keys are persisted to the *%LOCALAPPDATA%/ASP.NET/DataProtection-Keys* folder.

  The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:

  1. Navigate to the *%windir%/system32/inetsrv/config* folder.
  2. Open the *applicationHost.config* file.
  3. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.
  4. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

- **Use the file system as a key ring store**

  Adjust the app code to use the file system as a key ring store. Use an X509 certificate to protect the key ring and ensure the certificate is a trusted certificate. If the certificate is self-signed, place the certificate in the Trusted Root store.

  When using IIS in a web farm:

  - Use a file share that all machines can access.
  - Deploy an X509 certificate to each machine. Configure data protection in code.

- **Set a machine-wide policy for data protection**

  The data protection system has limited support for setting a default machine-wide policy for all apps

that consume the Data Protection APIs. For more information, see ASP.NET Core Data Protection.

Virtual Directories

IIS Virtual Directories aren't supported with ASP.NET Core apps. An app can be hosted as a sub-application.

Sub-applications

An ASP.NET Core app can be hosted as an IIS sub-application (sub-app). The sub-app's path becomes part of the root app's URL.

A sub-app shouldn't include the ASP.NET Core Module as a handler. If the module is added as a handler in a sub-app's *web.config* file, a *500.19 Internal Server Error* referencing the faulty config file is received when attempting to browse the sub-app.

The following example shows a published *web.config* file for an ASP.NET Core sub-app:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

When hosting a non-ASP.NET Core sub-app underneath an ASP.NET Core app, explicitly remove the inherited handler in the sub-app's *web.config* file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore" />
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

Static asset links within the sub-app should use tilde-slash (`~/`) notation. Tilde-slash notation triggers a Tag Helper to prepend the sub-app's pathbase to the rendered relative link. For a sub-app at `/subapp_path` , an image linked with `src="~/image.png"` is rendered as `src="/subapp_path/image.png"` . The root app's Static File Middleware doesn't process the static file request. The request is processed by the sub-app's Static File Middleware.

If a static asset's `src` attribute is set to an absolute path (for example, `src="/image.png"`), the link is rendered without the sub-app's pathbase. The root app's Static File Middleware attempts to serve the asset from the root app's web root, which results in a *404 - Not Found* response unless the static asset is available from the root app.

To host an ASP.NET Core app as a sub-app under another ASP.NET Core app:

1. Establish an app pool for the sub-app. Set the **.NET CLR Version** to **No Managed Code** because the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process, not the desktop CLR (.NET CLR).

2. Add the root site in IIS Manager with the sub-app in a folder under the root site.
3. Right-click the sub-app folder in IIS Manager and select **Convert to Application**.
4. In the **Add Application** dialog, use the **Select** button for the **Application Pool** to assign the app pool that you created for the sub-app. Select **OK**.

The assignment of a separate app pool to the sub-app is a requirement when using the in-process hosting model.

For more information on the in-process hosting model and configuring the ASP.NET Core Module, see ASP.NET Core Module.

Configuration of IIS with web.config

IIS configuration is influenced by the `<system.webServer>` section of *web.config* for IIS scenarios that are functional for ASP.NET Core apps with the ASP.NET Core Module. For example, IIS configuration is functional for dynamic compression. If IIS is configured at the server level to use dynamic compression, the `<urlCompression>` element in the app's *web.config* file can disable it for an ASP.NET Core app.

For more information, see the following topics:

- Configuration reference for <system.webServer>
- ASP.NET Core Module
- IIS modules with ASP.NET Core

To set environment variables for individual apps running in isolated app pools (supported for IIS 10.0 or later), see the *AppCmd.exe command* section of the Environment Variables <environmentVariables> topic in the IIS reference documentation.

Configuration sections of web.config

Configuration sections of ASP.NET 4.x apps in *web.config* aren't used by ASP.NET Core apps for configuration:

- `<system.web>`
- `<appSettings>`
- `<connectionStrings>`
- `<location>`

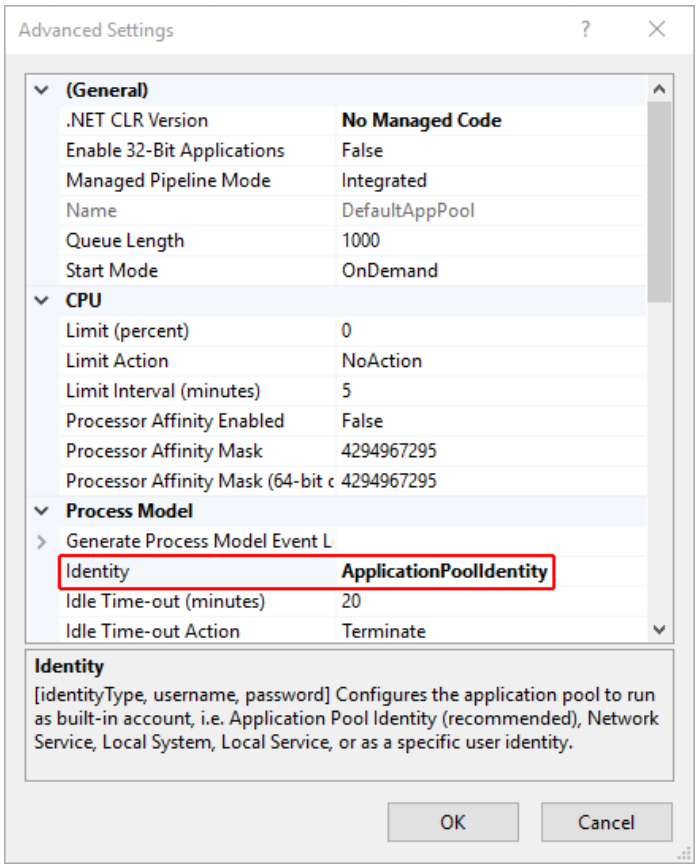
ASP.NET Core apps are configured using other configuration providers. For more information, see Configuration.

Application Pools

When hosting multiple websites on a server, we recommend isolating the apps from each other by running each app in its own app pool. The IIS **Add Website** dialog defaults to this configuration. When a **Site name** is provided, the text is automatically transferred to the **Application pool** textbox. A new app pool is created using the site name when the site is added.

Application Pool Identity

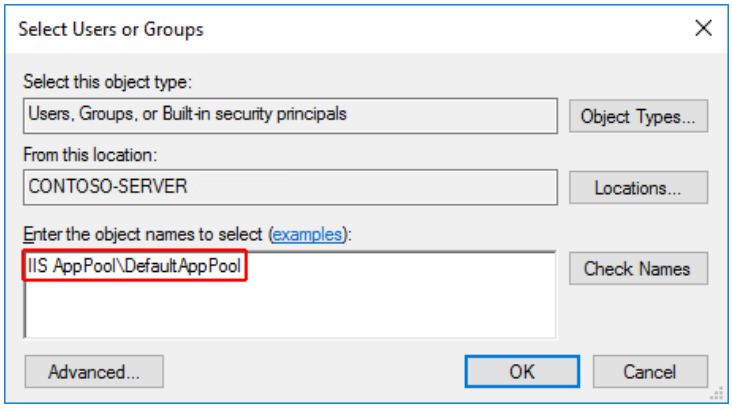
An app pool identity account allows an app to run under a unique account without having to create and manage domains or local accounts. On IIS 8.0 or later, the IIS Admin Worker Process (WAS) creates a virtual account with the name of the new app pool and runs the app pool's worker processes under this account by default. In the IIS Management Console under **Advanced Settings** for the app pool, ensure that the **Identity** is set to use **ApplicationPoolIdentity**:



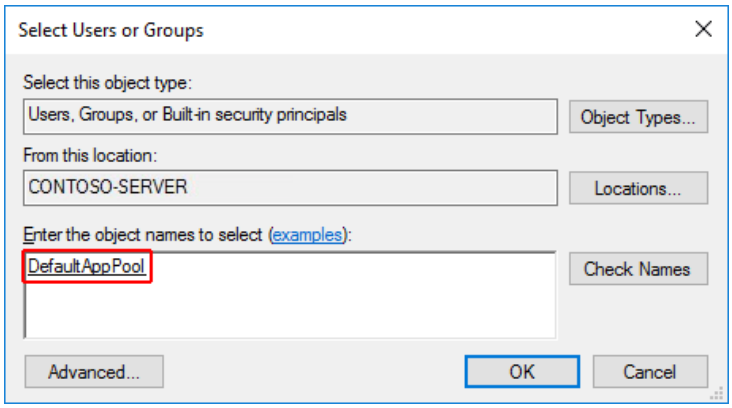
The IIS management process creates a secure identifier with the name of the app pool in the Windows Security System. Resources can be secured using this identity. However, this identity isn't a real user account and doesn't show up in the Windows User Management Console.

If the IIS worker process requires elevated access to the app, modify the Access Control List (ACL) for the directory containing the app:

1. Open Windows Explorer and navigate to the directory.
2. Right-click on the directory and select **Properties**.
3. Under the **Security** tab, select the **Edit** button and then the **Add** button.
4. Select the **Locations** button and make sure the system is selected.
5. Enter **IIS AppPool\<app_pool_name>** in **Enter the object names to select** area. Select the **Check Names** button. For the *DefaultAppPool* check the names using **IIS AppPool\DefaultAppPool**. When the **Check Names** button is selected, a value of **DefaultAppPool** is indicated in the object names area. It isn't possible to enter the app pool name directly into the object names area. Use the **IIS AppPool\<app_pool_name>** format when checking for the object name.



6. Select **OK**.



7. Read & execute permissions should be granted by default. Provide additional permissions as needed.

Access can also be granted at a command prompt using the **ICACLS** tool. Using the *DefaultAppPool* as an example, the following command is used:

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool":F
```

For more information, see the icacls topic.

HTTP/2 support

HTTP/2 is supported for out-of-process deployments that meet the following base requirements:

- Windows Server 2016/Windows 10 or later; IIS 10 or later
- Public-facing edge server connections use HTTP/2, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.
- Target framework: Not applicable to out-of-process deployments, since the HTTP/2 connection is handled entirely by IIS.
- TLS 1.2 or later connection

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/1.1`.

HTTP/2 is enabled by default. Connections fall back to HTTP/1.1 if an HTTP/2 connection isn't established. For more information on HTTP/2 configuration with IIS deployments, see HTTP/2 on IIS.

CORS preflight requests

This section only applies to ASP.NET Core apps that target the .NET Framework.

For an ASP.NET Core app that targets the .NET Framework, OPTIONS requests aren't passed to the app by default in IIS. To learn how to configure the app's IIS handlers in *web.config* to pass OPTIONS requests, see Enable cross-origin requests in ASP.NET Web API 2: How CORS Works.

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common errors reference for Azure App Service and IIS with ASP.NET Core

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Introduction to ASP.NET Core
- The Official Microsoft IIS Site
- Windows Server technical content library
- HTTP/2 on IIS
- Transform web.config

Publish an ASP.NET Core app to IIS

9/22/2020 • 5 minutes to read • Edit Online

This tutorial shows how to host an ASP.NET Core app on an IIS server.

This tutorial covers the following subjects:

- Install the .NET Core Hosting Bundle on Windows Server.
- Create an IIS site in IIS Manager.
- Deploy an ASP.NET Core app.

Prerequisites

- .NET Core SDK installed on the development machine.
- Windows Server configured with the **Web Server (IIS)** server role. If your server isn't configured to host websites with IIS, follow the guidance in the *IIS configuration* section of the Host ASP.NET Core on Windows with IIS article and then return to this tutorial.

> **WARNING**
>
> **IIS configuration and website security involve concepts that aren't covered by this tutorial.** Consult the IIS guidance in the Microsoft IIS documentation and the ASP.NET Core article on hosting with IIS before hosting production apps on IIS.
>
> Important scenarios for IIS hosting not covered by this tutorial include:
>
> - Creation of a registry hive for ASP.NET Core Data Protection
> - Configuration of the app pool's Access Control List (ACL)
> - To focus on IIS deployment concepts, this tutorial deploys an app without HTTPS security configured in IIS. For more information on hosting an app enabled for HTTPS protocol, see the security topics in the Additional resources section of this article. Further guidance for hosting ASP.NET Core apps is provided in the Host ASP.NET Core on Windows with IIS article.

Install the .NET Core Hosting Bundle

Install the *.NET Core Hosting Bundle* on the IIS server. The bundle installs the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. The module allows ASP.NET Core apps to run behind IIS.

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

1. Run the installer on the IIS server.
2. Restart the server or execute **net stop was /y** followed by **net start w3svc** in a command shell.

Create the IIS site

1. On the IIS server, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app.
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.

3. Provide a **Site name** and set the **Physical path** to the app's deployment folder that you created. Provide the **Binding** configuration and create the website by selecting **OK**.

Create an ASP.NET Core Razor Pages app

Follow the Get started with ASP.NET Core tutorial to create a Razor Pages app.

Publish and deploy the app

Publish an app means to produce a compiled app that can be hosted by a server. *Deploy an app* means to move the published app to a hosting system. The publish step is handled by the .NET Core SDK, while the deployment step can be handled by a variety of approaches. This tutorial adopts the *folder* deployment approach, where:

- The app is published to a folder.
- The folder's contents are moved to the IIS site's folder (the **Physical path** to the site in IIS Manager).

- Visual Studio
- .NET Core CLI
- Visual Studio for Mac

1. Right-click on the project in **Solution Explorer** and select **Publish**.
2. In the **Pick a publish target** dialog, select the **Folder** publish option.
3. Set the **Folder or File Share** path.
   - If you created a folder for the IIS site that's available on the development machine as a network share, provide the path to the share. The current user must have write access to publish to the share.
   - If you're unable to deploy directly to the IIS site folder on the IIS server, publish to a folder on removeable media and physically move the published app to the IIS site folder on the server, which is the site's **Physical path** in IIS Manager. Move the contents of the *bin/Release/{TARGET FRAMEWORK}/publish* folder to the IIS site folder on the server, which is the site's **Physical path** in IIS Manager.

Browse the website

The app is accessible in a browser after it receives the first request. Make a request to the app at the endpoint binding that you established in IIS Manager for the site.

Next steps

In this tutorial, you learned how to:

- Install the .NET Core Hosting Bundle on Windows Server.
- Create an IIS site in IIS Manager.
- Deploy an ASP.NET Core app.

To learn more about hosting ASP.NET Core apps on IIS, see the IIS Overview article:

Host ASP.NET Core on Windows with IIS

Additional resources

Articles in the ASP.NET Core documentation set

- ASP.NET Core Module
- ASP.NET Core directory structure
- Troubleshoot ASP.NET Core on Azure App Service and IIS

- Enforce HTTPS in ASP.NET Core

Articles pertaining to ASP.NET Core app deployment

- Publish an ASP.NET Core app to Azure with Visual Studio
- Publish an ASP.NET Core app to Azure with Visual Studio Code
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment
- Publish a Web app to a folder using Visual Studio for Mac

Articles on IIS HTTPS configuration

- Configuring SSL in IIS Manager
- How to Set Up SSL on IIS

Articles on IIS and Windows Server

- The Official Microsoft IIS Site
- Windows Server technical content library

ASP.NET Core Module

9/22/2020 • 40 minutes to read • Edit Online

By Tom Dykstra, Rick Strahl, Chris Ross, Rick Anderson, Sourabh Shirhatti, and Justin Kotalik

The ASP.NET Core Module is a native IIS module that plugs into the IIS pipeline to either:

- Host an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forward web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

Supported Windows versions:

- Windows 7 or later
- Windows Server 2012 R2 or later

When hosting in-process, the module uses an in-process server implementation for IIS, called IIS HTTP Server (`IISHttpServer`).

When hosting out-of-process, the module only works with Kestrel. The module doesn't function with HTTP.sys.

Hosting models

In-process hosting model

ASP.NET Core apps default to the in-process hosting model.

The following characteristics apply when hosting in-process:

- IIS HTTP Server (`IISHttpServer`) is used instead of Kestrel server. For in-process, CreateDefaultBuilder calls UseIIS to:
  - Register the `IISHttpServer`.
  - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
  - Configure the host to capture startup errors.
- The requestTimeout attribute doesn't apply to in-process hosting.
- Sharing an app pool among apps isn't supported. Use one app pool per app.
- When using Web Deploy or manually placing an app_offline.htm file in the deployment, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.
- The architecture (bitness) of the app and installed runtime (x64 or x86) must match the architecture of the app pool.
- Client disconnects are detected. The HttpContext.RequestAborted cancellation token is cancelled when the client disconnects.
- In ASP.NET Core 2.2.1 or earlier, GetCurrentDirectory returns the worker directory of the process started by IIS rather than the app's directory (for example,

C:\Windows\System32\inetsrv for *w3wp.exe*).

For sample code that sets the app's current directory, see the CurrentDirectoryHelpers class. Call the `SetCurrentDirectory` method. Subsequent calls to GetCurrentDirectory provide the app's directory.

- When hosting in-process, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. When transforming claims with an IClaimsTransformation implementation, call AddAuthentication to add authentication services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IClaimsTransformation, ClaimsTransformer>();
    services.AddAuthentication(IISServerDefaults.AuthenticationScheme);
}

public void Configure(IApplicationBuilder app)
{
    app.UseAuthentication();
}
```

  - Web Package (single-file) deployments aren't supported.

Out-of-process hosting model

To configure an app for out-of-process hosting, set the value of the `<AspNetCoreHostingModel>` property to `OutOfProcess` in the project file (*.csproj*):

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

In-process hosting is set with `InProcess`, which is the default value.

The value of `<AspNetCoreHostingModel>` is case insensitive, so `inprocess` and `outofprocess` are valid values.

Kestrel server is used instead of IIS HTTP Server (`IISHttpServer`).

For out-of-process, CreateDefaultBuilder calls UseIISIntegration to:

- Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
- Configure the host to capture startup errors.

Hosting model changes

If the `hostingModel` setting is changed in the *web.config* file (explained in the Configuration with web.config section), the module recycles the worker process for IIS.

For IIS Express, the module doesn't recycle the worker process but instead triggers a graceful shutdown of the current IIS Express process. The next request to the app spawns a new IIS Express process.

Process name

`Process.GetCurrentProcess().ProcessName` reports `w3wp` / `iisexpress` (in-process) or `dotnet` (out-of-process).

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

How to install and use the ASP.NET Core Module

For instructions on how to install the ASP.NET Core Module, see Install the .NET Core Hosting Bundle.

Configuration with web.config

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's *web.config* file.

The following *web.config* file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\MyApp.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The following *web.config* is published for a self-contained deployment:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The InheritInChildApplications property is set to `false` to indicate that the settings

specified within the <location> element aren't inherited by apps that reside in a subdirectory of the app.

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the *LogFiles* folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Host ASP.NET Core on Windows with IIS.

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION	DEFAULT
`arguments`	Optional string attribute. Arguments to the executable specified in **processPath**.	
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code page configured in the *web.config* takes precedence.	`false`
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`hostingModel`	Optional string attribute. Specifies the hosting model as in-process (`InProcess` / `inprocess`) or out-of-process (`OutOfProcess` / `outofprocess`).	`InProcess` `inprocess`

ATTRIBUTE	DESCRIPTION	DEFAULT
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the **processPath** setting that can be spun up per app. †For in-process hosting, the value is limited to `1`. Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100` †
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.`, the path is considered to be relative to the site root.	
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the module stops launching the process for the remainder of the minute. Not supported with in-process hosting.	Default: `10` Min: `0` Max: `100`

ATTRIBUTE	DESCRIPTION	DEFAULT
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds. Doesn't apply to in-process hosting. For in-process hosting, the module waits for the app to process the request. Valid values for minutes and seconds segments of the string are in the range 0-59. Use of **60** in the value for minutes or seconds results in a *500 - Internal Server Error*.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the *app_offline.htm* file is detected.	Default: `10` Min: `0` Max: `600`

ATTRIBUTE	DESCRIPTION	DEFAULT
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this time limit is exceeded, the module kills the process. When hosting *in-process*: The process is **not** restarted and does **not** use the **rapidFailsPerMinute** setting. When hosting *out-of-process*: The module attempts to relaunch the process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start **rapidFailsPerMinute** number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	Default: `120` Min: `0` Max: `3600`
`stdoutLogEnabled`	Optional Boolean attribute. If true, **stdout** and **stderr** for the process specified in **processPath** are redirected to the file specified in **stdoutLogFile**.	`false`

ATTRIBUTE	DESCRIPTION	DEFAULT
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which **stdout** and **stderr** from the process specified in **processPath** are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path are created by the module when the log file is created. Using underscore delimiters, a timestamp, process ID, and file extension (*.log*) are added to the last segment of the **stdoutLogFile** path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as *stdout_20180205194132_1934.log* in the *logs* folder when saved on 2/5/2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Set environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element. Environment variables set in this section take precedence over system environment variables.

The following example sets two environment variables in *web.config*. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

NOTE

An alternative to setting the environment directly in *web.config* is to include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

WARNING

Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

app_offline.htm

If a file with the name *app_offline.htm* is detected in the root directory of an app, the ASP.NET Core Module attempts to gracefully shutdown the app and stop processing incoming requests. If the app is still running after the number of seconds defined in `shutdownTimeLimit`, the ASP.NET Core Module kills the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module responds to requests by sending back the contents of the *app_offline.htm* file. When the *app_offline.htm* file is removed, the next request starts the app.

When using the out-of-process hosting model, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.

Start-up error page

Both in-process and out-of-process hosting produce custom error pages when they fail to start the app.

If the ASP.NET Core Module fails to find either the in-process or out-of-process request handler, a *500.0 - In-Process/Out-Of-Process Handler Load Failure* status code page appears.

For in-process hosting if the ASP.NET Core Module fails to start the app, a *500.30 - Start Failure* status code page appears.

For out-of-process hosting if the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, a *502.5 - Process Failure* status code page appears.

To suppress this page and revert to the default IIS 5xx status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see HTTP Errors <httpErrors>.

Log creation and redirection

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any

folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (*.log*) to the last segment of the `stdoutLogFile` path (typically *stdout*) delimited by underscores. If the `stdoutLogFile` path ends with *stdout*, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name *stdout_20180205194132_1934.log*.

If `stdoutLogEnabled` is false, errors that occur on app startup are captured and emitted to the event log up to 30 KB. After startup, all additional logs are discarded.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout"
    hostingModel="inprocess">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is predefined for apps hosted by Azure App Service.

To create logging filter rules, see the Configuration and Log filtering sections of the ASP.NET Core logging documentation.

For more information on path formats, see File path formats on Windows systems.

Enhanced diagnostic logs

The ASP.NET Core Module is configurable to provide enhanced diagnostics logs. Add the `<handlerSettings>` element to the `<aspNetCore>` element in *web.config*. Setting the `debugLevel` to `TRACE` exposes a higher fidelity of diagnostic information:

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="debugFile" value=".\logs\aspnetcore-debug.log" />
    <handlerSetting name="debugLevel" value="FILE,TRACE" />
  </handlerSettings>
</aspNetCore>
```

Any folders in the path (*logs* in the preceding example) are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Debug level (`debugLevel`) values can include both the level and the location.

Levels (in order from least to most verbose):

- ERROR
- WARNING
- INFO
- TRACE

Locations (multiple locations are permitted):

- CONSOLE
- EVENTLOG
- FILE

The handler settings can also be provided via environment variables:

- `ASPNETCORE_MODULE_DEBUG_FILE`: Path to the debug log file. (Default: *aspnetcore-debug.log*)
- `ASPNETCORE_MODULE_DEBUG`: Debug level setting.

> **WARNING**
> Do **not** leave debug logging enabled in the deployment for longer than required to troubleshoot an issue. The size of the log isn't limited. Leaving the debug log enabled can exhaust the available disk space and crash the server or app service.

See Configuration with web.config for an example of the `aspNetCore` element in the *web.config* file.

Modify the stack size

Only applies when using the in-process hosting model.

Configure the managed stack size using the `stackSize` setting in bytes in *web.config*. The default size is `1048576` bytes (1 MB).

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="stackSize" value="2097152" />
  </handlerSettings>
</aspNetCore>
```

Proxy configuration uses HTTP protocol and a pairing token

Only applies to out-of-process hosting.

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, an attacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **TrustedInstaller** account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the *applicationHost.config* file on the share.

When using an IIS Shared Configuration on the same machine as the IIS installation, run the ASP.NET Core Hosting Bundle installer with the `OPT_NO_SHARED_CONFIG_CHECK` parameter set to `1`:

```
dotnet-hosting-{VERSION}.exe OPT_NO_SHARED_CONFIG_CHECK=1
```

When the path to the shared configuration isn't on the same machine as the IIS installation, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated *applicationHost.config* file to the share.
4. Re-enable the IIS Shared Configuration.

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to *%windir%\System32\inetsrv*.
2. Locate the *aspnetcore.dll* file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at *C:\Users\%UserName%\AppData\Local\Temp*. The file is named *dd_DotNetCoreWinSvrHosting__ <timestamp>_000_AspNetCoreModule_x64.log*.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll
- %ProgramFiles%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll

IIS Express (x86/amd64):

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll
- %ProgramFiles%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll

Schema

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml
- %windir%\System32\inetsrv\config\schema\aspnetcore_schema_v2.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml
- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema_v2.xml

Configuration

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- Visual Studio: {APPLICATION ROOT}\.vs\config\applicationHost.config
- *iisexpress.exe* CLI: %USERPROFILE%\Documents\IISExpress\config\applicationhost.config

The files can be found by searching for *aspnetcore* in the *applicationHost.config* file.

The ASP.NET Core Module is a native IIS module that plugs into the IIS pipeline to either:

- Host an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forward web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

Supported Windows versions:

- Windows 7 or later
- Windows Server 2008 R2 or later

When hosting in-process, the module uses an in-process server implementation for IIS, called IIS HTTP Server (`IISHttpServer`).

When hosting out-of-process, the module only works with Kestrel. The module doesn't function with HTTP.sys.

Hosting models

In-process hosting model

To configure an app for in-process hosting, add the `<AspNetCoreHostingModel>` property to the app's project file with a value of `InProcess` (out-of-process hosting is set with `OutOfProcess`):

```
<PropertyGroup>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

The in-process hosting model isn't supported for ASP.NET Core apps that target the .NET Framework.

The value of `<AspNetCoreHostingModel>` is case insensitive, so `inprocess` and `outofprocess` are valid values.

If the `<AspNetCoreHostingModel>` property isn't present in the file, the default value is `OutOfProcess`.

The following characteristics apply when hosting in-process:

- IIS HTTP Server (`IISHttpServer`) is used instead of Kestrel server. For in-process, CreateDefaultBuilder calls UseIIS to:
  - Register the `IISHttpServer`.
  - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
  - Configure the host to capture startup errors.
- The requestTimeout attribute doesn't apply to in-process hosting.
- Sharing an app pool among apps isn't supported. Use one app pool per app.
- When using Web Deploy or manually placing an app_offline.htm file in the deployment, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.
- The architecture (bitness) of the app and installed runtime (x64 or x86) must match the architecture of the app pool.

- Client disconnects are detected. The HttpContext.RequestAborted cancellation token is cancelled when the client disconnects.
- In ASP.NET Core 2.2.1 or earlier, GetCurrentDirectory returns the worker directory of the process started by IIS rather than the app's directory (for example, *C:\Windows\System32\inetsrv* for *w3wp.exe*).

  For sample code that sets the app's current directory, see the CurrentDirectoryHelpers class. Call the `SetCurrentDirectory` method. Subsequent calls to GetCurrentDirectory provide the app's directory.
- When hosting in-process, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. When transforming claims with an IClaimsTransformation implementation, call AddAuthentication to add authentication services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IClaimsTransformation, ClaimsTransformer>();
    services.AddAuthentication(IISServerDefaults.AuthenticationScheme);
}

public void Configure(IApplicationBuilder app)
{
    app.UseAuthentication();
}
```

Out-of-process hosting model

To configure an app for out-of-process hosting, use either of the following approaches in the project file:

- Don't specify the `<AspNetCoreHostingModel>` property. If the `<AspNetCoreHostingModel>` property isn't present in the file, the default value is `OutOfProcess`.
- Set the value of the `<AspNetCoreHostingModel>` property to `OutOfProcess` (in-process hosting is set with `InProcess`):

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

The value is case insensitive, so `inprocess` and `outofprocess` are valid values.

Kestrel server is used instead of IIS HTTP Server (`IISHttpServer`).

For out-of-process, CreateDefaultBuilder calls UseIISIntegration to:

- Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
- Configure the host to capture startup errors.

Hosting model changes

If the `hostingModel` setting is changed in the *web.config* file (explained in the Configuration with web.config section), the module recycles the worker process for IIS.

For IIS Express, the module doesn't recycle the worker process but instead triggers a graceful shutdown of the current IIS Express process. The next request to the app spawns

a new IIS Express process.

Process name

`Process.GetCurrentProcess().ProcessName` reports `w3wp` / `iisexpress` (in-process) or `dotnet` (out-of-process).

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

How to install and use the ASP.NET Core Module

For instructions on how to install the ASP.NET Core Module, see Install the .NET Core Hosting Bundle.

Configuration with web.config

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's *web.config* file.

The following *web.config* file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\MyApp.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The following *web.config* is published for a self-contained deployment:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the app.

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the *LogFiles* folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Host ASP.NET Core on Windows with IIS.

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION	DEFAULT
`arguments`	Optional string attribute. Arguments to the executable specified in **processPath**.	
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code page configured in the *web.config* takes precedence.	`false`

ATTRIBUTE	DESCRIPTION	DEFAULT
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`hostingModel`	Optional string attribute. Specifies the hosting model as in-process (`InProcess` / `inprocess`) or out-of-process (`OutOfProcess` / `outofprocess`).	`OutOfProcess` `outofprocess`
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the **processPath** setting that can be spun up per app. †For in-process hosting, the value is limited to `1` . Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100` †
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.` , the path is considered to be relative to the site root.	

ATTRIBUTE	DESCRIPTION	DEFAULT
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the module stops launching the process for the remainder of the minute. Not supported with in-process hosting.	Default: `10` Min: `0` Max: `100`
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds. Doesn't apply to in-process hosting. For in-process hosting, the module waits for the app to process the request. Valid values for minutes and seconds segments of the string are in the range 0-59. Use of **60** in the value for minutes or seconds results in a *500 - Internal Server Error*.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the *app_offline.htm* file is detected.	Default: `10` Min: `0` Max: `600`

ATTRIBUTE	DESCRIPTION	DEFAULT
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this time limit is exceeded, the module kills the process. When hosting *in-process*: The process is **not** restarted and does **not** use the **rapidFailsPerMinute** setting. When hosting *out-of-process*: The module attempts to relaunch the process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start **rapidFailsPerMinute** number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	Default: `120` Min: `0` Max: `3600`
`stdoutLogEnabled`	Optional Boolean attribute. If true, **stdout** and **stderr** for the process specified in **processPath** are redirected to the file specified in **stdoutLogFile**.	`false`

ATTRIBUTE	DESCRIPTION	DEFAULT
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which **stdout** and **stderr** from the process specified in **processPath** are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path are created by the module when the log file is created. Using underscore delimiters, a timestamp, process ID, and file extension (*.log*) are added to the last segment of the **stdoutLogFile** path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as *stdout_20180205194132_1934.log* in the *logs* folder when saved on 2/5/2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Setting environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element. Environment variables set in this section take precedence over system environment variables.

The following example sets two environment variables. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

NOTE

An alternative to setting the environment directly in *web.config* is to include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

WARNING

Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

app_offline.htm

If a file with the name *app_offline.htm* is detected in the root directory of an app, the ASP.NET Core Module attempts to gracefully shutdown the app and stop processing incoming requests. If the app is still running after the number of seconds defined in `shutdownTimeLimit`, the ASP.NET Core Module kills the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module responds to requests by sending back the contents of the *app_offline.htm* file. When the *app_offline.htm* file is removed, the next request starts the app.

When using the out-of-process hosting model, the app might not shut down immediately if there's an open connection. For example, a websocket connection may delay app shut down.

Start-up error page

Both in-process and out-of-process hosting produce custom error pages when they fail to start the app.

If the ASP.NET Core Module fails to find either the in-process or out-of-process request handler, a *500.0 - In-Process/Out-Of-Process Handler Load Failure* status code page appears.

For in-process hosting if the ASP.NET Core Module fails to start the app, a *500.30 - Start Failure* status code page appears.

For out-of-process hosting if the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, a *502.5 - Process Failure* status code page appears.

To suppress this page and revert to the default IIS 5xx status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see HTTP Errors <httpErrors>.

Log creation and redirection

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any

folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (*.log*) to the last segment of the `stdoutLogFile` path (typically *stdout*) delimited by underscores. If the `stdoutLogFile` path ends with *stdout*, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name *stdout_20180205194132_1934.log*.

If `stdoutLogEnabled` is false, errors that occur on app startup are captured and emitted to the event log up to 30 KB. After startup, all additional logs are discarded.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout"
    hostingModel="inprocess">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is predefined for apps hosted by Azure App Service.

For more information on path formats, see File path formats on Windows systems.

Enhanced diagnostic logs

The ASP.NET Core Module is configurable to provide enhanced diagnostics logs. Add the `<handlerSettings>` element to the `<aspNetCore>` element in *web.config*. Setting the `debugLevel` to `TRACE` exposes a higher fidelity of diagnostic information:

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="debugFile" value=".\logs\aspnetcore-debug.log" />
    <handlerSetting name="debugLevel" value="FILE,TRACE" />
  </handlerSettings>
</aspNetCore>
```

Folders in the path provided to the `<handlerSetting>` value (*logs* in the preceding example) aren't created by the module automatically and should pre-exist in the deployment. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Debug level (`debugLevel`) values can include both the level and the location.

Levels (in order from least to most verbose):

- ERROR
- WARNING
- INFO
- TRACE

Locations (multiple locations are permitted):

- CONSOLE
- EVENTLOG
- FILE

The handler settings can also be provided via environment variables:

- `ASPNETCORE_MODULE_DEBUG_FILE`: Path to the debug log file. (Default: *aspnetcore-debug.log*)
- `ASPNETCORE_MODULE_DEBUG`: Debug level setting.

> **WARNING**
> Do **not** leave debug logging enabled in the deployment for longer than required to troubleshoot an issue. The size of the log isn't limited. Leaving the debug log enabled can exhaust the available disk space and crash the server or app service.

See Configuration with web.config for an example of the `aspNetCore` element in the *web.config* file.

Proxy configuration uses HTTP protocol and a pairing token

Only applies to out-of-process hosting.

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by

IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, an attacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **TrustedInstaller** account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the *applicationHost.config* file on the share.

When using an IIS Shared Configuration on the same machine as the IIS installation, run the ASP.NET Core Hosting Bundle installer with the `OPT_NO_SHARED_CONFIG_CHECK` parameter set to `1`:

```
dotnet-hosting-{VERSION}.exe OPT_NO_SHARED_CONFIG_CHECK=1
```

When the path to the shared configuration isn't on the same machine as the IIS installation, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated *applicationHost.config* file to the share.
4. Re-enable the IIS Shared Configuration.

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to *%windir%\System32\inetsrv*.
2. Locate the *aspnetcore.dll* file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at *C:\Users\%UserName%\AppData\Local\Temp*. The file is named *dd_DotNetCoreWinSvrHosting__<timestamp>_000_AspNetCoreModule_x64.log*.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll

- %ProgramFiles%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll

IIS Express (x86/amd64):

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll
- %ProgramFiles%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll
- %ProgramFiles(x86)%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll

Schema

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml
- %windir%\System32\inetsrv\config\schema\aspnetcore_schema_v2.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml
- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema_v2.xml

Configuration

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- Visual Studio: {APPLICATION ROOT}\.vs\config\applicationHost.config
- *iisexpress.exe* CLI: %USERPROFILE%\Documents\IISExpress\config\applicationhost.config

The files can be found by searching for *aspnetcore* in the *applicationHost.config* file.

The ASP.NET Core Module is a native IIS module that plugs into the IIS pipeline to forward web requests to backend ASP.NET Core apps.

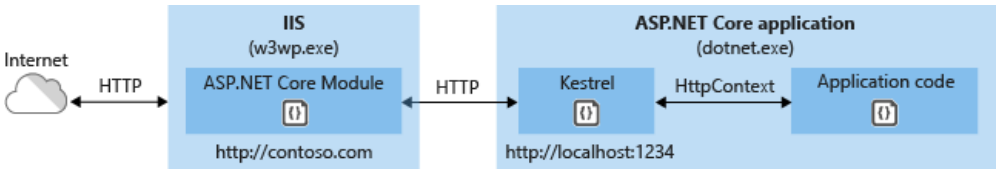
Supported Windows versions:

- Windows 7 or later
- Windows Server 2008 R2 or later

The module only works with Kestrel. The module is incompatible with HTTP.sys.

Because ASP.NET Core apps run in a process separate from the IIS worker process, the module also handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it crashes. This is essentially the same behavior as seen with ASP.NET 4.x apps that run in-process in IIS that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app:



Requests arrive from the web to the kernel-mode HTTP.sys driver. The driver routes the requests to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS). The module forwards the requests to Kestrel on a random port for the app, which isn't port 80 or 443.

The module specifies the port via an environment variable at startup, and the IIS Integration Middleware configures the server to listen on `http://localhost:{port}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding, so requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is pushed into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which pushes it back out to the HTTP client that initiated the request.

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

How to install and use the ASP.NET Core Module

For instructions on how to install the ASP.NET Core Module, see Install the .NET Core Hosting Bundle.

Configuration with web.config

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's *web.config* file.

The following *web.config* file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet"
                arguments=".\MyApp.dll"
                stdoutLogEnabled="false"
                stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

The following *web.config* is published for a self-contained deployment:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath=".\MyApp.exe"
                stdoutLogEnabled="false"
                stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the *LogFiles* folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Host ASP.NET Core on Windows with IIS.

Attributes of the aspNetCore element

ATTRIBUTE	DESCRIPTION	DEFAULT
`arguments`	Optional string attribute. Arguments to the executable specified in **processPath**.	
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code page configured in the *web.config* takes precedence.	`false`

ATTRIBUTE	DESCRIPTION	DEFAULT
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on %ASPNETCORE_PORT% as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the **processPath** setting that can be spun up per app. Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100`
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.`, the path is considered to be relative to the site root.	
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in **processPath** is allowed to crash per minute. If this limit is exceeded, the module stops launching the process for the remainder of the minute.	Default: `10` Min: `0` Max: `100`

ATTRIBUTE	DESCRIPTION	DEFAULT
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the *app_offline.htm* file is detected.	Default: `10` Min: `0` Max: `600`
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this time limit is exceeded, the module kills the process. The module attempts to relaunch the process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start **rapidFailsPerMinute** number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	Default: `120` Min: `0` Max: `3600`

ATTRIBUTE	DESCRIPTION	DEFAULT
`stdoutLogEnabled`	Optional Boolean attribute. If true, **stdout** and **stderr** for the process specified in **processPath** are redirected to the file specified in **stdoutLogFile**.	`false`
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which **stdout** and **stderr** from the process specified in **processPath** are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path must exist in order for the module to create the log file. Using underscore delimiters, a timestamp, process ID, and file extension (*.log*) are added to the last segment of the **stdoutLogFile** path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as *stdout_20180205194132_1934.log* in the *logs* folder when saved on 2/5/2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Setting environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element.

> **WARNING**
>
> Environment variables set in this section conflict with system environment variables set with the same name. If an environment variable is set in both the *web.config* file and at the system level in Windows, the value from the *web.config* file becomes appended to the system environment variable value (for example, `ASPNETCORE_ENVIRONMENT: Development;Development`), which prevents the app from starting.

The following example sets two environment variables. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the *web.config* file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile="\\?\%home%\LogFiles\stdout">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

WARNING

Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

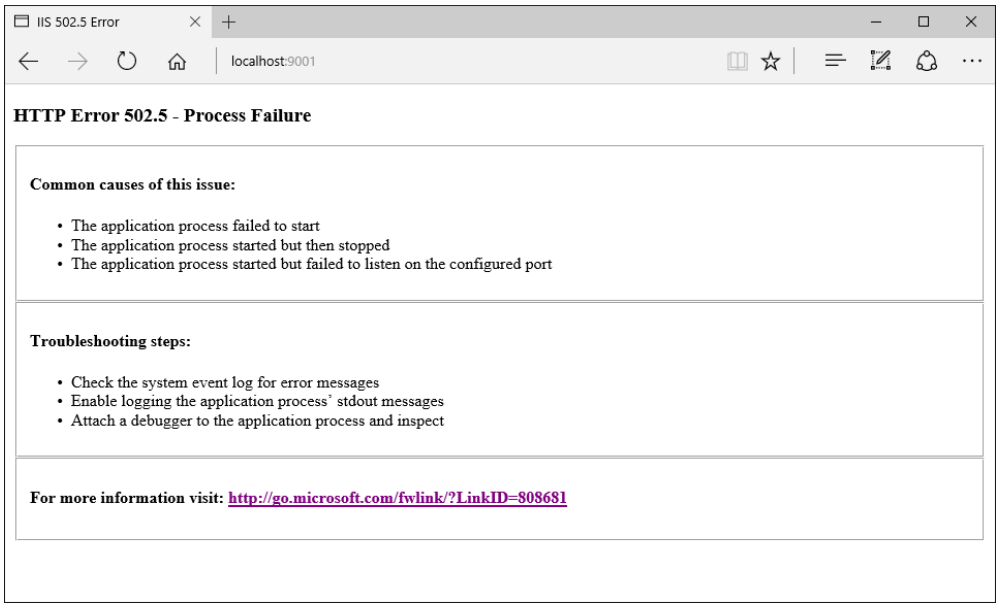
app_offline.htm

If a file with the name *app_offline.htm* is detected in the root directory of an app, the ASP.NET Core Module attempts to gracefully shutdown the app and stop processing incoming requests. If the app is still running after the number of seconds defined in `shutdownTimeLimit`, the ASP.NET Core Module kills the running process.

While the *app_offline.htm* file is present, the ASP.NET Core Module responds to requests by sending back the contents of the *app_offline.htm* file. When the *app_offline.htm* file is removed, the next request starts the app.

Start-up error page

If the ASP.NET Core Module fails to launch the backend process or the backend process starts but fails to listen on the configured port, a *502.5 - Process Failure* status code page appears. To suppress this page and revert to the default IIS 502 status code page, use the `disableStartUpErrorPage` attribute. For more information on configuring custom error messages, see HTTP Errors <httpErrors>.



Log creation and redirection

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\<app_pool_name>` to provide write permission).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (*.log*) to the last segment of the `stdoutLogFile` path (typically *stdout*) delimited by underscores. If the `stdoutLogFile` path ends with *stdout*, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name *stdout_20180205194132_1934.log*.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path provided.

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is

predefined for apps hosted by Azure App Service.

To create logging filter rules, see the Configuration and Log filtering sections of the ASP.NET Core logging documentation.

For more information on path formats, see File path formats on Windows systems.

Proxy configuration uses HTTP protocol and a pairing token

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, an attacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the **TrustedInstaller** account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the *applicationHost.config* file on the share.

When using an IIS Shared Configuration, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated *applicationHost.config* file to the share.
4. Re-enable the IIS Shared Configuration.

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to *%windir%\System32\inetsrv*.
2. Locate the *aspnetcore.dll* file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at *C:\Users\%UserName%\AppData\Local\Temp*. The file is named *dd_DotNetCoreWinSvrHosting__<timestamp>_000_AspNetCoreModule_x64.log*.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- %windir%\System32\inetsrv\aspnetcore.dll
- %windir%\SysWOW64\inetsrv\aspnetcore.dll

IIS Express (x86/amd64):

- %ProgramFiles%\IIS Express\aspnetcore.dll
- %ProgramFiles(x86)%\IIS Express\aspnetcore.dll

Schema

IIS

- %windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml

IIS Express

- %ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml

Configuration

IIS

- %windir%\System32\inetsrv\config\applicationHost.config

IIS Express

- Visual Studio: {APPLICATION ROOT}\.vs\config\applicationHost.config
- *iisexpress.exe* CLI: %USERPROFILE%\Documents\IISExpress\config\applicationhost.config

The files can be found by searching for *aspnetcore* in the *applicationHost.config* file.

Additional resources

- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service
- ASP.NET Core Module reference source (master branch): Use the **Branch** drop down list to select a specific release (for example, `release/3.1`).
- IIS modules with ASP.NET Core

Development-time IIS support in Visual Studio for ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

By Sourabh Shirhatti

This article describes Visual Studio support for debugging ASP.NET Core apps running with IIS on Windows Server. This topic walks through enabling this scenario and setting up a project.

Prerequisites

- Visual Studio for Windows
- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload
- X.509 security certificate (for HTTPS support)

Enable IIS

1. In Windows, navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Select the **Internet Information Services** check box. Select **OK**.

The IIS installation may require a system restart.

Configure IIS

IIS must have a website configured with the following:

- **Host name**: Typically, the **Default Web Site** is used with a **Host name** of `localhost`. However, any valid IIS website with a unique host name works.
- **Site Binding**
  - For apps that require HTTPS, create a binding to port 443 with a certificate. Typically, the **IIS Express Development Certificate** is used, but any valid certificate works.
  - For apps that use HTTP, confirm the existence of a binding to post 80 or create a binding to port 80 for a new site.
  - Use a single binding for either HTTP or HTTPS. **Binding to both HTTP and HTTPS ports simultaneously isn't supported.**

Enable development-time IIS support in Visual Studio

1. Launch the Visual Studio installer.
2. Select **Modify** for the Visual Studio installation that you plan to use for IIS development-time support.
3. For the **ASP.NET and web development** workload, locate and install the **Development time IIS support** component.

   The component is listed in the **Optional** section under **Development time IIS support** in the **Installation details** panel to the right of the workloads. The component installs the ASP.NET Core Module, which is a native IIS module required to run ASP.NET Core apps with IIS.

Configure the project

HTTPS redirection

For a new project that requires HTTPS, select the check box to **Configure for HTTPS** in the **Create a new ASP.NET Core Web Application** window. Selecting the check box adds HTTPS Redirection and HSTS Middleware to the app when it's created.

For an existing project that requires HTTPS, use HTTPS Redirection and HSTS Middleware in `Startup.Configure`. For more information, see Enforce HTTPS in ASP.NET Core.

For a project that uses HTTP, HTTPS Redirection and HSTS Middleware aren't added to the app. No app configuration is required.

IIS launch profile

Create a new launch profile to add development-time IIS support:

1. Right-click the project in **Solution Explorer**. Select **Properties**. Open the **Debug** tab.
2. For **Profile**, select the **New** button. Name the profile "IIS" in the popup window. Select **OK** to create the profile.
3. For the **Launch** setting, select **IIS** from the list.
4. Select the check box for **Launch browser** and provide the endpoint URL.

   When the app requires HTTPS, use an HTTPS endpoint (`https://`). For HTTP, use an HTTP (`http://`) endpoint.

   Provide the same host name and port as the IIS configuration specified earlier uses, typically `localhost`.

   Provide the name of the app at the end of the URL.

   For example, `https://localhost/WebApplication1` (HTTPS) or `http://localhost/WebApplication1` (HTTP) are valid endpoint URLs.
5. In the **Environment variables** section, select the **Add** button. Provide an environment variable with a **Name** of `ASPNETCORE_ENVIRONMENT` and a **Value** of `Development`.
6. In the **Web Server Settings** area, set the **App URL** to the same value used for the **Launch browser** endpoint URL.
7. For the **Hosting Model** setting in Visual Studio 2019 or later, select **Default** to use the hosting model used by the project. If the project sets the `<AspNetCoreHostingModel>` property in its project file, the value of the property (`InProcess` or `OutOfProcess`) is used. If the property isn't present, the default hosting model of the app is used, which is in-process. If the app requires an explicit hosting model setting different from the app's normal hosting model, set the **Hosting Model** to either `In Process` or `Out Of Process` as needed.
8. Save the profile.

When not using Visual Studio, manually add a launch profile to the launchSettings.json file in the *Properties* folder. The following example configures the profile to use the HTTPS protocol:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iis": {
      "applicationUrl": "https://localhost/WebApplication1",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS": {
      "commandName": "IIS",
      "launchBrowser": true,
      "launchUrl": "https://localhost/WebApplication1",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Confirm that the `applicationUrl` and `launchUrl` endpoints match and use the same protocol as the IIS binding configuration, either HTTP or HTTPS.

Run the project

Run Visual Studio as an administrator:

- Confirm that the build configuration drop-down list is set to **Debug**.
- Set the Start Debugging button to the **IIS** profile and select the button to start the app.

Visual Studio may prompt a restart if not running as an administrator. If prompted, restart Visual Studio.

If an untrusted development certificate is used, the browser may require you to create an exception for the untrusted certificate.

> **NOTE**
> Debugging a Release build configuration with Just My Code and compiler optimizations results in a degraded experience. For example, break points aren't hit.

Additional resources

- Getting Started with the IIS Manager in IIS
- Enforce HTTPS in ASP.NET Core

This article describes Visual Studio support for debugging ASP.NET Core apps running with IIS on Windows Server. This topic walks through enabling this scenario and setting up a project.

Prerequisites

- Visual Studio for Windows
- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload
- X.509 security certificate (for HTTPS support)

Enable IIS

1. In Windows, navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Select the **Internet Information Services** check box. Select **OK**.

The IIS installation may require a system restart.

Configure IIS

IIS must have a website configured with the following:

- **Host name**: Typically, the **Default Web Site** is used with a **Host name** of `localhost`. However, any valid IIS website with a unique host name works.
- **Site Binding**
  - For apps that require HTTPS, create a binding to port 443 with a certificate. Typically, the **IIS Express Development Certificate** is used, but any valid certificate works.
  - For apps that use HTTP, confirm the existence of a binding to post 80 or create a binding to port 80 for a new site.
  - Use a single binding for either HTTP or HTTPS. **Binding to both HTTP and HTTPS ports simultaneously isn't supported.**

Enable development-time IIS support in Visual Studio

1. Launch the Visual Studio installer.
2. Select **Modify** for the Visual Studio installation that you plan to use for IIS development-time support.
3. For the **ASP.NET and web development** workload, locate and install the **Development time IIS support** component.

   The component is listed in the **Optional** section under **Development time IIS support** in the **Installation details** panel to the right of the workloads. The component installs the ASP.NET Core Module, which is a native IIS module required to run ASP.NET Core apps with IIS.

Configure the project

HTTPS redirection

For a new project that requires HTTPS, select the check box to **Configure for HTTPS** in the **Create a new ASP.NET Core Web Application** window. Selecting the check box adds HTTPS Redirection and HSTS Middleware to the app when it's created.

For an existing project that requires HTTPS, use HTTPS Redirection and HSTS Middleware in `Startup.Configure`. For more information, see Enforce HTTPS in ASP.NET Core.

For a project that uses HTTP, HTTPS Redirection and HSTS Middleware aren't added to the app. No app configuration is required.

IIS launch profile

Create a new launch profile to add development-time IIS support:

1. Right-click the project in **Solution Explorer**. Select **Properties**. Open the **Debug** tab.
2. For **Profile**, select the **New** button. Name the profile "IIS" in the popup window. Select **OK** to create the profile.

3. For the **Launch** setting, select **IIS** from the list.
4. Select the check box for **Launch browser** and provide the endpoint URL.

   When the app requires HTTPS, use an HTTPS endpoint (`https://`). For HTTP, use an HTTP (`http://`) endpoint.

   Provide the same host name and port as the IIS configuration specified earlier uses, typically `localhost`.

   Provide the name of the app at the end of the URL.

   For example, `https://localhost/WebApplication1` (HTTPS) or `http://localhost/WebApplication1` (HTTP) are valid endpoint URLs.
5. In the **Environment variables** section, select the **Add** button. Provide an environment variable with a **Name** of `ASPNETCORE_ENVIRONMENT` and a **Value** of `Development`.
6. In the **Web Server Settings** area, set the **App URL** to the same value used for the **Launch browser** endpoint URL.
7. For the **Hosting Model** setting in Visual Studio 2019 or later, select **Default** to use the hosting model used by the project. If the project sets the `<AspNetCoreHostingModel>` property in its project file, the value of the property (`InProcess` or `OutOfProcess`) is used. If the property isn't present, the default hosting model of the app is used, which is out-of-process. If the app requires an explicit hosting model setting different from the app's normal hosting model, set the **Hosting Model** to either `In Process` or `Out Of Process` as needed.
8. Save the profile.

When not using Visual Studio, manually add a launch profile to the launchSettings.json file in the *Properties* folder. The following example configures the profile to use the HTTPS protocol:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iis": {
      "applicationUrl": "https://localhost/WebApplication1",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS": {
      "commandName": "IIS",
      "launchBrowser": true,
      "launchUrl": "https://localhost/WebApplication1",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Confirm that the `applicationUrl` and `launchUrl` endpoints match and use the same protocol as the IIS binding configuration, either HTTP or HTTPS.

Run the project

Run Visual Studio as an administrator:

- Confirm that the build configuration drop-down list is set to **Debug**.

- Set the Start Debugging button to the **IIS** profile and select the button to start the app.

Visual Studio may prompt a restart if not running as an administrator. If prompted, restart Visual Studio.

If an untrusted development certificate is used, the browser may require you to create an exception for the untrusted certificate.

> **NOTE**
>
> Debugging a Release build configuration with Just My Code and compiler optimizations results in a degraded experience. For example, break points aren't hit.

Additional resources

- Getting Started with the IIS Manager in IIS
- Enforce HTTPS in ASP.NET Core

IIS modules with ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

Some of the native IIS modules and all of the IIS managed modules aren't able to process requests for ASP.NET Core apps. In many cases, ASP.NET Core offers an alternative to the scenarios addressed by IIS native and managed modules.

Native modules

The table indicates native IIS modules that are functional with ASP.NET Core apps and the ASP.NET Core Module.

MODULE	FUNCTIONAL WITH ASP.NET CORE APPS	ASP.NET CORE OPTION
Anonymous Authentication `AnonymousAuthenticationModule`	Yes	
Basic Authentication `BasicAuthenticationModule`	Yes	
Client Certification Mapping Authentication `CertificateMappingAuthenticationModule`	Yes	
CGI `CgiModule`	No	
Configuration Validation `ConfigurationValidationModule`	Yes	
HTTP Errors `CustomErrorModule`	No	Status Code Pages Middleware
Custom Logging `CustomLoggingModule`	Yes	
Default Document `DefaultDocumentModule`	No	Default Files Middleware
Digest Authentication `DigestAuthenticationModule`	Yes	
Directory Browsing `DirectoryListingModule`	No	Directory Browsing Middleware
Dynamic Compression `DynamicCompressionModule`	Yes	Response Compression Middleware
Failed Requests Tracing `FailedRequestsTracingModule`	Yes	ASP.NET Core Logging

MODULE	FUNCTIONAL WITH ASP.NET CORE APPS	ASP.NET CORE OPTION
File Caching `FileCacheModule`	No	Response Caching Middleware
HTTP Caching `HttpCacheModule`	No	Response Caching Middleware
HTTP Logging `HttpLoggingModule`	Yes	ASP.NET Core Logging
HTTP Redirection `HttpRedirectionModule`	Yes	URL Rewriting Middleware
HTTP Tracing `TracingModule`	Yes	
IIS Client Certificate Mapping Authentication `IISCertificateMappingAuthenticationModule`	Yes	
IP and Domain Restrictions `IpRestrictionModule`	Yes	
ISAPI Filters `IsapiFilterModule`	Yes	Middleware
ISAPI `IsapiModule`	Yes	Middleware
Protocol Support `ProtocolSupportModule`	Yes	
Request Filtering `RequestFilteringModule`	Yes	URL Rewriting Middleware `IRule`
Request Monitor `RequestMonitorModule`	Yes	
URL Rewriting† `RewriteModule`	Yes	URL Rewriting Middleware
Server-Side Includes `ServerSideIncludeModule`	No	
Static Compression `StaticCompressionModule`	No	Response Compression Middleware
Static Content `StaticFileModule`	No	Static File Middleware
Token Caching `TokenCacheModule`	Yes	

MODULE	FUNCTIONAL WITH ASP.NET CORE APPS	ASP.NET CORE OPTION
URI Caching `UriCacheModule`	Yes	
URL Authorization `UrlAuthorizationModule`	Yes	ASP.NET Core Identity
Windows Authentication `WindowsAuthenticationModule`	Yes	

†The URL Rewrite Module's `isFile` and `isDirectory` match types don't work with ASP.NET Core apps due to the changes in directory structure.

Managed modules

Managed modules are *not* functional with hosted ASP.NET Core apps when the app pool's .NET CLR version is set to **No Managed Code**. ASP.NET Core offers middleware alternatives in several cases.

MODULE	ASP.NET CORE OPTION
AnonymousIdentification	
DefaultAuthentication	
FileAuthorization	
FormsAuthentication	Cookie Authentication Middleware
OutputCache	Response Caching Middleware
Profile	
RoleManager	
ScriptModule-4.0	
Session	Session Middleware
UrlAuthorization	
UrlMappingsModule	URL Rewriting Middleware
UrlRoutingModule-4.0	ASP.NET Core Identity
WindowsAuthentication	

IIS Manager application changes

When using IIS Manager to configure settings, the *web.config* file of the app is changed. If deploying an app and including *web.config*, any changes made with IIS Manager are overwritten by the deployed *web.config* file. If changes are made to the server's *web.config* file, copy the updated *web.config* file on the server to the local

project immediately.

Disabling IIS modules

If an IIS module is configured at the server level that must be disabled for an app, an addition to the app's *web.config* file can disable the module. Either leave the module in place and deactivate it using a configuration setting (if available) or remove the module from the app.

Module deactivation

Many modules offer a configuration setting that allows them to be disabled without removing the module from the app. This is the simplest and quickest way to deactivate a module. For example, the HTTP Redirection Module can be disabled with the `<httpRedirect>` element in *web.config*:

```
<configuration>
  <system.webServer>
    <httpRedirect enabled="false" />
  </system.webServer>
</configuration>
```

For more information on disabling modules with configuration settings, follow the links in the *Child Elements* section of IIS <system.webServer>.

Module removal

If opting to remove a module with a setting in *web.config*, unlock the module and unlock the `<modules>` section of *web.config* first:

1. Unlock the module at the server level. Select the IIS server in the IIS Manager **Connections** sidebar. Open the **Modules** in the **IIS** area. Select the module in the list. In the **Actions** sidebar on the right, select **Unlock**. If the action entry for the module appears as **Lock**, the module is already unlocked, and no action is required. Unlock as many modules as you plan to remove from *web.config* later.

2. Deploy the app without a `<modules>` section in *web.config*. If an app is deployed with a *web.config* containing the `<modules>` section without having unlocked the section first in the IIS Manager, the Configuration Manager throws an exception when attempting to unlock the section. Therefore, deploy the app without a `<modules>` section.

3. Unlock the `<modules>` section of *web.config*. In the **Connections** sidebar, select the website in **Sites**. In the **Management** area, open the **Configuration Editor**. Use the navigation controls to select the `system.webServer/modules` section. In the **Actions** sidebar on the right, select to **Unlock** the section. If the action entry for the module section appears as **Lock Section**, the module section is already unlocked, and no action is required.

4. Add a `<modules>` section to the app's local *web.config* file with a `<remove>` element to remove the module from the app. Add multiple `<remove>` elements to remove multiple modules. If *web.config* changes are made on the server, immediately make the same changes to the project's *web.config* file locally. Removing a module using this approach doesn't affect the use of the module with other apps on the server.

   ```
   <configuration>
    <system.webServer>
      <modules>
        <remove name="MODULE_NAME" />
      </modules>
    </system.webServer>
   </configuration>
   ```

In order to add or remove modules for IIS Express using *web.config*, modify *applicationHost.config* to unlock the `<modules>` section:

1. Open *{APPLICATION ROOT}\.vs\config\applicationhost.config*.
2. Locate the `<section>` element for IIS modules and change `overrideModeDefault` from `Deny` to `Allow`:

   ```
   <section name="modules"
            allowDefinition="MachineToApplication"
            overrideModeDefault="Allow" />
   ```

3. Locate the `<location path="" overrideMode="Allow"><system.webServer><modules>` section. For any modules that you wish to remove, set `lockItem` from `true` to `false`. In the following example, the CGI Module is unlocked:

   ```
   <add name="CgiModule" lockItem="false" />
   ```

4. After the `<modules>` section and individual modules are unlocked, you're free to add or remove IIS modules using the app's *web.config* file for running the app on IIS Express.

An IIS module can also be removed with *Appcmd.exe*. Provide the `MODULE_NAME` and `APPLICATION_NAME` in the command:

```
Appcmd.exe delete module MODULE_NAME /app.name:APPLICATION_NAME
```

For example, remove the `DynamicCompressionModule` from the Default Web Site:

```
%windir%\system32\inetsrv\appcmd.exe delete module DynamicCompressionModule /app.name:"Default Web Site"
```

Minimum module configuration

The only modules required to run an ASP.NET Core app are the Anonymous Authentication Module and the ASP.NET Core Module.

The URI Caching Module (`UriCacheModule`) allows IIS to cache website configuration at the URL level. Without this module, IIS must read and parse configuration on every request, even when the same URL is repeatedly requested. Parsing the configuration every request results in a significant performance penalty. *Although the URI Caching Module isn't strictly required for a hosted ASP.NET Core app to run, we recommend that the URI Caching Module be enabled for all ASP.NET Core deployments.*

The HTTP Caching Module (`HttpCacheModule`) implements the IIS output cache and also the logic for caching items in the HTTP.sys cache. Without this module, content is no longer cached in kernel mode, and cache profiles are ignored. Removing the HTTP Caching Module usually has adverse effects on performance and resource usage. *Although the HTTP Caching Module isn't strictly required for a hosted ASP.NET Core app to run, we recommend that the HTTP Caching Module be enabled for all ASP.NET Core deployments.*

Additional resources

- Introduction to IIS Architectures: Modules in IIS
- IIS Modules Overview
- Customizing IIS 7.0 Roles and Modules
- IIS <system.webServer>

Troubleshoot ASP.NET Core on Azure App Service and IIS

9/22/2020 • 59 minutes to read • Edit Online

By Justin Kotalik

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:

- Confirm that the *web.config* file is present in the deployment and that its contents are correct.
- When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
- When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error, either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file deployment model. Use **one** of the following approaches in the app's project file to fix this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:

- If the app is a framework-dependent deployment:

```
dotnet .\{ASSEMBLY NAME}.dll
```

- If the app is a self-contained deployment:

```
{ASSEMBLY NAME}.exe
```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`
- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.

4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:
   - Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
   - Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
     a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
     b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
     c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.
2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to

create the *logs* folder in the deployment automatically, see the Directory structure topic.

3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.
2. Run the EnableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017

- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.

- When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module fails to find the .NET Core CLR and find the in-process request handler (*aspnetcorev2_inprocess.dll*). Check that:

- The app targets either the Microsoft.AspNetCore.Server.IIS NuGet package or the Microsoft.AspNetCore.App metapackage.
- The version of the ASP.NET Core shared framework that the app targets is installed on the target machine.

500.0 Out-Of-Process Handler Load Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module fails to find the out-of-process hosting request handler. Make sure the *aspnetcorev2_outofprocess.dll* is present in a subfolder next to *aspnetcorev2.dll*.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.

4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go →** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`
- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Under **SELECT PROBLEM CATEGORY**, select the **Web App Down** button.
3. Under **Suggested Solutions** > **Enable Stdout Log Redirection**, select the button to **Open Kudu Console to edit Web.Config**.
4. In the Kudu **Diagnostic Console**, open the folders to the path **site** > **wwwroot**. Scroll down to reveal the *web.config* file at the bottom of the list.
5. Click the pencil icon next to the *web.config* file.
6. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
7. Select **Save** to save the updated *web.config* file.
8. Make a request to the app.
9. Return to the Azure portal. Select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
10. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
11. Select the **LogFiles** folder.
12. Inspect the **Modified** column and select the pencil icon to edit the stdout log with the latest modification date.
13. When the log file opens, the error is displayed.

Disable stdout logging when troubleshooting is complete:

1. In the Kudu **Diagnostic Console**, return to the path **site** > **wwwroot** to reveal the *web.config* file. Open the **web.config** file again by selecting the pencil icon.
2. Set **stdoutLogEnabled** to `false`.
3. Select **Save** to save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:
   - Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
   - Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
     a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
     b. Using the navigation bar at the top of the page, open **Debug console** and select

CMD.

c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select

the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.
2. Run the EnableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.

3. Restore and rebuild the project.

4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, an ASP.NET Core project defaults to IIS Express hosting during debugging. A *502.5 Process Failure* that occurs when debugging locally can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:
   - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
   - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
   - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.
3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as

`Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.
2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:
   - If deploying a 32-bit (x86) app, set the value to `True`.
   - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.

4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go →** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the *eventlog.xml* file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go →** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:
   - If the app is a framework-dependent deployment:

     ```
     dotnet .\{ASSEMBLY NAME}.dll
     ```

   - If the app is a self-contained deployment:

     ```
     {ASSEMBLY NAME}.exe
     ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:
  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`
- If the app is a self-contained deployment:
  1. `cd D:\home\site\wwwroot`

2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Under **SELECT PROBLEM CATEGORY**, select the **Web App Down** button.
3. Under **Suggested Solutions** > **Enable Stdout Log Redirection**, select the button to **Open Kudu Console to edit Web.Config**.
4. In the Kudu **Diagnostic Console**, open the folders to the path **site** > **wwwroot**. Scroll down to reveal the *web.config* file at the bottom of the list.
5. Click the pencil icon next to the *web.config* file.
6. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
7. Select **Save** to save the updated *web.config* file.
8. Make a request to the app.
9. Return to the Azure portal. Select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
10. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
11. Select the **LogFiles** folder.
12. Inspect the **Modified** column and select the pencil icon to edit the stdout log with the latest modification date.
13. When the log file opens, the error is displayed.

Disable stdout logging when troubleshooting is complete:

1. In the Kudu **Diagnostic Console**, return to the path **site** > **wwwroot** to reveal the *web.config* file. Open the **web.config** file again by selecting the pencil icon.
2. Set **stdoutLogEnabled** to `false`.
3. Select **Save** to save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or hanging app (Azure App Service)

When an app responds slowly or hangs on a request, see the following articles:

- Troubleshoot slow web app performance issues in Azure App Service
- Use Crash Diagnoser Site Extension to Capture Dump for Intermittent Exception issues or performance issues on Azure Web App

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go→** button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.
5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed

request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.

2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`). `stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.
4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module.

> **WARNING**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or hanging app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.
2. Run the EnableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:
   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Common errors reference for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)
- Azure Friday: Azure App Service Diagnostic and Troubleshooting Experience (12-minute video)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common errors reference for Azure App Service and IIS with ASP.NET Core

9/22/2020 • 22 minutes to read • Edit Online

This topic describes common errors and provides troubleshooting advice for specific errors when hosting ASP.NET Core apps on Azure Apps Service and IIS.

For general troubleshooting guidance, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Collect the following information:

- Browser behavior (status code and error message)
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS
    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.
- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions

installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely
- **ASP.NET Core Module stdout Log:** The log file is created but empty.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the

RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED --**OR**-- Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied --**OR**-- 403.14 Forbidden - The Web server is configured to not list the contents of this directory.

- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. --**OR**-- This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:

{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.

- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Download Archives and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK

commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.
- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure --**OR**-- HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

This topic describes common errors and provides troubleshooting advice for specific errors when hosting ASP.NET Core apps on Azure Apps Service and IIS.

For general troubleshooting guidance, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Collect the following information:

- Browser behavior (status code and error message)
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS
    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.
- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **IMPORTANT**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application

'/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect

format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED --**OR**-- Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied --**OR**-- 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. --**OR**-- This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0.
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

Current .NET Core Hosting Bundle installer (direct download)

For more information, see Install the .NET Core Hosting Bundle.

If a specific runtime is required, download the runtime from the .NET Download Archives and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"dotnet" .{ASSEMBLY}.dll', ErrorCode = '0x80004005 : 80008081.
- **ASP.NET Core Module stdout Log:** The application to execute does not exist: 'PATH{ASSEMBLY}.dll'

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section.

stdout log path incorrect

- **Browser:** The app responds normally.

- **Application Log:** Warning: Could not create stdoutLogFile \?{PATH}\path_doesnt_exist\stdout_{PROCESS ID}_{TIMESTAMP}.log, ErrorCode = -2147024893.
- **ASP.NET Core Module stdout Log:** The log file isn't created.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' created process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ' but either crashed or did not respond or did not listen on the given port '{PORT}', ErrorCode = '{ERROR CODE}'
- **ASP.NET Core Module stdout Log:** The log file is created but empty.

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

Transform web.config

9/22/2020 • 2 minutes to read • Edit Online

By Vijay Ramakrishnan

Transformations to the *web.config* file can be applied automatically when an app is published based on:

- Build configuration
- Profile
- Environment
- Custom

These transformations occur for either of the following *web.config* generation scenarios:

- Generated automatically by the `Microsoft.NET.Sdk.Web` SDK.
- Provided by the developer in the content root of the app.

Build configuration

Build configuration transforms are run first.

Include a *web.{CONFIGURATION}.config* file for each build configuration (Debug|Release) requiring a *web.config* transformation.

In the following example, a configuration-specific environment variable is set in *web.Release.config*:

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Configuration_Specific"
                               value="Configuration_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the configuration is set to *Release*:

```
dotnet publish --configuration Release
```

The MSBuild property for the configuration is `$(Configuration)`.

Profile

Profile transformations are run second, after Build configuration transforms.

Include a *web.{PROFILE}.config* file for each profile configuration requiring a *web.config* transformation.

In the following example, a profile-specific environment variable is set in *web.FolderProfile.config* for a folder publish profile:

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Profile_Specific"
                               value="Profile_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the profile is *FolderProfile*:

```
dotnet publish --configuration Release /p:PublishProfile=FolderProfile
```

The MSBuild property for the profile name is `$(PublishProfile)`.

If no profile is passed, the default profile name is **FileSystem** and *web.FileSystem.config* is applied if the file is present in the app's content root.

Environment

Environment transformations are run third, after Build configuration and Profile transforms.

Include a *web.{ENVIRONMENT}.config* file for each environment requiring a *web.config* transformation.

In the following example, a environment-specific environment variable is set in *web.Production.config* for the Production environment:

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Environment_Specific"
                               value="Environment_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the environment is *Production*:

```
dotnet publish --configuration Release /p:EnvironmentName=Production
```

The MSBuild property for the environment is `$(EnvironmentName)`.

When publishing from Visual Studio and using a publish profile, see Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment.

The `ASPNETCORE_ENVIRONMENT` environment variable is automatically added to the *web.config* file when the environment name is specified.

Custom

Custom transformations are run last, after Build configuration, Profile, and Environment transforms.

Include a *{CUSTOM_NAME}.transform* file for each custom configuration requiring a *web.config* transformation.

In the following example, a custom transform environment variable is set in *custom.transform*:

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Custom_Specific"
                               value="Custom_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the `CustomTransformFileName` property is passed to the dotnet publish command:

```
dotnet publish --configuration Release /p:CustomTransformFileName=custom.transform
```

The MSBuild property for the profile name is `$(CustomTransformFileName)`.

Prevent web.config transformation

To prevent transformations of the *web.config* file, set the MSBuild property `$(IsWebConfigTransformDisabled)`:

```
dotnet publish /p:IsWebConfigTransformDisabled=true
```

Additional resources

- Web.config Transformation Syntax for Web Application Project Deployment
- Web.config Transformation Syntax for Web Project Deployment Using Visual Studio

Kestrel web server implementation in ASP.NET Core

9/22/2020 • 69 minutes to read • Edit Online

By Tom Dykstra, Chris Ross, and Stephen Halter

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the web server that's included by default in ASP.NET Core project templates.

Kestrel supports the following scenarios:

- HTTPS
- Opaque upgrade used to enable WebSockets
- Unix sockets for high performance behind Nginx
- HTTP/2 (except on macOS†)

†HTTP/2 will be supported on macOS in a future release.

Kestrel is supported on all platforms and versions that .NET Core supports.

View or download sample code (how to download)

HTTP/2 support

HTTP/2 is available for ASP.NET Core apps if the following base requirements are met:

- Operating system†
  - Windows Server 2016/Windows 10 or later‡
  - Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
- Target framework: .NET Core 2.2 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

†HTTP/2 will be supported on macOS in a future release. ‡Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

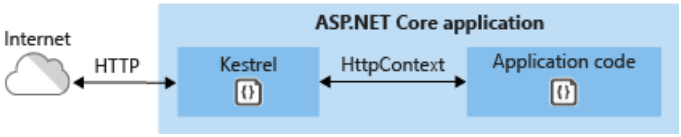
If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

HTTP/2 is disabled by default. For more information on configuration, see the Kestrel options and ListenOptions.Protocols sections.

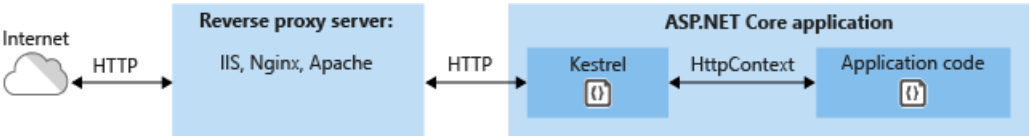
When to use Kestrel with a reverse proxy

Kestrel can be used by itself or with a *reverse proxy server*, such as Internet Information Services (IIS), Nginx, or Apache. A reverse proxy server receives HTTP requests from the network and forwards them to Kestrel.

Kestrel used as an edge (Internet-facing) web server:



Kestrel used in a reverse proxy configuration:



Either configuration, with or without a reverse proxy server, is a supported hosting configuration.

Kestrel used as an edge server without a reverse proxy server doesn't support sharing the same IP and port among multiple processes. When Kestrel is configured to listen on a port, Kestrel handles all of the traffic for that port regardless of requests' `Host` headers. A reverse proxy that can share ports has the ability to forward requests to Kestrel on a unique IP and port.

Even if a reverse proxy server isn't required, using a reverse proxy server might be a good choice.

A reverse proxy:

- Can limit the exposed public surface area of the apps that it hosts.
- Provide an additional layer of configuration and defense.
- Might integrate better with existing infrastructure.
- Simplify load balancing and secure communication (HTTPS) configuration. Only the reverse proxy server requires an X.509 certificate, and that server can communicate with the app's servers on the internal network using plain HTTP.

> **WARNING**
> Hosting in a reverse proxy configuration requires host filtering.

Kestrel in ASP.NET Core apps

ASP.NET Core project templates use Kestrel by default. In *Program.cs*, the ConfigureWebHostDefaults method calls UseKestrel:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

For more information on building the host, see the *Set up a host* and *Default builder settings* sections of .NET Generic Host.

To provide additional configuration after calling `ConfigureWebHostDefaults`, use

`ConfigureKestrel`:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(serverOptions =>
            {
                // Set properties and call methods on options
            })
            .UseStartup<Startup>();
        });
```

Kestrel options

The Kestrel web server has constraint configuration options that are especially useful in Internet-facing deployments.

Set constraints on the Limits property of the KestrelServerOptions class. The `Limits` property holds an instance of the KestrelServerLimits class.

The following examples use the Microsoft.AspNetCore.Server.Kestrel.Core namespace:

```
using Microsoft.AspNetCore.Server.Kestrel.Core;
```

In examples shown later in this article, Kestrel options are configured in C# code. Kestrel options can also be set using a configuration provider. For example, the File Configuration Provider can load Kestrel configuration from an *appsettings.json* or *appsettings.{Environment}.json* file:

```
{
  "Kestrel": {
    "Limits": {
      "MaxConcurrentConnections": 100,
      "MaxConcurrentUpgradedConnections": 100
    },
    "DisableStringReuse": true
  }
}
```

> **NOTE**
>
> KestrelServerOptions and endpoint configuration are configurable from configuration providers. Remaining Kestrel configuration must be configured in C# code.

Use **one** of the following approaches:

- Configure Kestrel in `Startup.ConfigureServices`:

  1. Inject an instance of `IConfiguration` into the `Startup` class. The following example assumes that the injected configuration is assigned to the `Configuration` property.

  2. In `Startup.ConfigureServices`, load the `Kestrel` section of configuration into Kestrel's configuration:

```
using Microsoft.Extensions.Configuration

public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<KestrelServerOptions>(
            Configuration.GetSection("Kestrel"));
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        ...
    }
}
```

- Configure Kestrel when building the host:

  In *Program.cs*, load the `Kestrel` section of configuration into Kestrel's configuration:

  ```
  // using Microsoft.Extensions.DependencyInjection;

  public static IHostBuilder CreateHostBuilder(string[] args) =>
      Host.CreateDefaultBuilder(args)
          .ConfigureServices((context, services) =>
          {
              services.Configure<KestrelServerOptions>(
                  context.Configuration.GetSection("Kestrel"));
          })
          .ConfigureWebHostDefaults(webBuilder =>
          {
              webBuilder.UseStartup<Startup>();
          });
  ```

Both of the preceding approaches work with any configuration provider.

Keep-alive timeout

KeepAliveTimeout

Gets or sets the keep-alive timeout. Defaults to 2 minutes.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001,
        listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx",
                "testPassword");
        });
    serverOptions.Limits.KeepAliveTimeout =
        TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout =
        TimeSpan.FromMinutes(1);
})
```

Maximum client connections

MaxConcurrentConnections MaxConcurrentUpgradedConnections

The maximum number of concurrent open TCP connections can be set for the entire app with the following code:

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001,
        listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx",
                "testPassword");
        });
    serverOptions.Limits.KeepAliveTimeout =
        TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout =
        TimeSpan.FromMinutes(1);
})
```

There's a separate limit for connections that have been upgraded from HTTP or HTTPS to another protocol (for example, on a WebSockets request). After a connection is upgraded, it isn't counted against the `MaxConcurrentConnections` limit.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001,
        listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx",
                "testPassword");
        });
    serverOptions.Limits.KeepAliveTimeout =
        TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout =
        TimeSpan.FromMinutes(1);
})
```

The maximum number of connections is unlimited (null) by default.

Maximum request body size

MaxRequestBodySize

The default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB.

The recommended approach to override the limit in an ASP.NET Core MVC app is to use the RequestSizeLimitAttribute attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

Here's an example that shows how to configure the constraint for the app on every request:

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001,
        listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx",
                "testPassword");
        });
    serverOptions.Limits.KeepAliveTimeout =
        TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout =
        TimeSpan.FromMinutes(1);
})
```

Override the setting on a specific request in middleware:

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;

    var minRequestRateFeature =
        context.Features.Get<IHttpMinRequestBodyDataRateFeature>();
    var minResponseRateFeature =
        context.Features.Get<IHttpMinResponseDataRateFeature>();

    if (minRequestRateFeature != null)
    {
        minRequestRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    if (minResponseRateFeature != null)
    {
        minResponseRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }
```

An exception is thrown if the app configures the limit on a request after the app has started to read the request. There's an `IsReadOnly` property that indicates if the `MaxRequestBodySize` property is in read-only state, meaning it's too late to configure the limit.

When an app is run out-of-process behind the ASP.NET Core Module, Kestrel's request body size limit is disabled because IIS already sets the limit.

Minimum request body data rate

MinRequestBodyDataRate MinResponseDataRate

Kestrel checks every second if data is arriving at the specified rate in bytes/second. If the rate drops below the minimum, the connection is timed out. The grace period is the amount of time that Kestrel gives the client to increase its send rate up to the minimum; the rate isn't checked during that time. The grace period helps avoid dropping connections that are initially sending data at a slow rate due to TCP slow-start.

The default minimum rate is 240 bytes/second with a 5 second grace period.

A minimum rate also applies to the response. The code to set the request limit and the response limit is the same except for having `RequestBody` or `Response` in the property and interface names.

Here's an example that shows how to configure the minimum data rates in *Program.cs*:

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001,
        listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx",
                "testPassword");
        });
    serverOptions.Limits.KeepAliveTimeout =
        TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout =
        TimeSpan.FromMinutes(1);
})
```

Override the minimum rate limits per request in middleware:

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;

    var minRequestRateFeature =
        context.Features.Get<IHttpMinRequestBodyDataRateFeature>();
    var minResponseRateFeature =
        context.Features.Get<IHttpMinResponseDataRateFeature>();

    if (minRequestRateFeature != null)
    {
        minRequestRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    if (minResponseRateFeature != null)
    {
        minResponseRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }
```

The IHttpMinResponseDataRateFeature referenced in the prior sample is not present in `HttpContext.Features` for HTTP/2 requests because modifying rate limits on a per-request basis is generally not supported for HTTP/2 due to the protocol's support for request multiplexing. However, the IHttpMinRequestBodyDataRateFeature is still present `HttpContext.Features` for HTTP/2 requests, because the read rate limit can still be *disabled entirely* on a per-request basis by setting `IHttpMinRequestBodyDataRateFeature.MinDataRate` to `null` even for an HTTP/2 request. Attempting to read `IHttpMinRequestBodyDataRateFeature.MinDataRate` or attempting to set it to a value other than `null` will result in a `NotSupportedException` being thrown given an HTTP/2 request.

Server-wide rate limits configured via `KestrelServerOptions.Limits` still apply to both HTTP/1.x and HTTP/2 connections.

Request headers timeout

RequestHeadersTimeout

Gets or sets the maximum amount of time the server spends receiving request headers. Defaults to 30 seconds.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100,
            gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001,
        listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx",
                "testPassword");
        });
    serverOptions.Limits.KeepAliveTimeout =
        TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout =
        TimeSpan.FromMinutes(1);
})
```

Maximum streams per connection

`Http2.MaxStreamsPerConnection` limits the number of concurrent request streams per HTTP/2 connection. Excess streams are refused.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxStreamsPerConnection = 100;
});
```

The default value is 100.

Header table size

The HPACK decoder decompresses HTTP headers for HTTP/2 connections. `Http2.HeaderTableSize` limits the size of the header compression table that the HPACK decoder uses. The value is provided in octets and must be greater than zero (0).

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.HeaderTableSize = 4096;
});
```

The default value is 4096.

Maximum frame size

`Http2.MaxFrameSize` indicates the maximum allowed size of an HTTP/2 connection frame payload received or sent by the server. The value is provided in octets and must be between 2^14 (16,384) and 2^24-1 (16,777,215).

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxFrameSize = 16384;
});
```

The default value is 2^14 (16,384).

Maximum request header size

`Http2.MaxRequestHeaderFieldSize` indicates the maximum allowed size in octets of request header values. This limit applies to both name and value in their compressed and uncompressed representations. The value must be greater than zero (0).

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxRequestHeaderFieldSize = 8192;
});
```

The default value is 8,192.

Initial connection window size

`Http2.InitialConnectionWindowSize` indicates the maximum request body data in bytes the server buffers at one time aggregated across all requests (streams) per connection. Requests are also limited by `Http2.InitialStreamWindowSize`. The value must be greater than or equal to 65,535 and less than 2^31 (2,147,483,648).

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.InitialConnectionWindowSize = 131072;
});
```

The default value is 128 KB (131,072).

Initial stream window size

`Http2.InitialStreamWindowSize` indicates the maximum request body data in bytes the server buffers at one time per request (stream). Requests are also limited by `Http2.InitialConnectionWindowSize`. The value must be greater than or equal to 65,535 and less than 2^31 (2,147,483,648).

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.InitialStreamWindowSize = 98304;
});
```

The default value is 96 KB (98,304).

Synchronous I/O

AllowSynchronousIO controls whether synchronous I/O is allowed for the request and response. The default value is `false`.

> **WARNING**
> A large number of blocking synchronous I/O operations can lead to thread pool starvation, which makes the app unresponsive. Only enable `AllowSynchronousIO` when using a library that doesn't support asynchronous I/O.

The following example enables synchronous I/O:

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.AllowSynchronousIO = true;
})
```

For information about other Kestrel options and limits, see:

- KestrelServerOptions
- KestrelServerLimits
- ListenOptions

Endpoint configuration

By default, ASP.NET Core binds to:

- `http://localhost:5000`
- `https://localhost:5001` (when a local development certificate is present)

Specify URLs using the:

- `ASPNETCORE_URLS` environment variable.
- `--urls` command-line argument.
- `urls` host configuration key.
- `UseUrls` extension method.

The value provided using these approaches can be one or more HTTP and HTTPS endpoints (HTTPS if a default cert is available). Configure the value as a semicolon-separated list (for example, `"Urls": "http://localhost:8000;http://localhost:8001"`).

For more information on these approaches, see Server URLs and Override configuration.

A development certificate is created:

- When the .NET Core SDK is installed.
- The dev-certs tool is used to create a certificate.

Some browsers require granting explicit permission to trust the local development certificate.

Project templates configure apps to run on HTTPS by default and include HTTPS redirection and HSTS support.

Call Listen or ListenUnixSocket methods on KestrelServerOptions to configure URL prefixes and ports for Kestrel.

`UseUrls`, the `--urls` command-line argument, `urls` host configuration key, and the `ASPNETCORE_URLS` environment variable also work but have the limitations noted later in this section (a default certificate must be available for HTTPS endpoint configuration).

`KestrelServerOptions` configuration:

ConfigureEndpointDefaults(Action<ListenOptions>)

Specifies a configuration `Action` to run for each specified endpoint. Calling `ConfigureEndpointDefaults` multiple times replaces prior `Action`s with the last `Action` specified.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions =>
    {
        // Configure endpoint defaults
    });
});
```

> **NOTE**
> Endpoints created by calling Listen **before** calling ConfigureEndpointDefaults won't have the defaults applied.

ConfigureHttpsDefaults(Action<HttpsConnectionAdapterOptions>)

Specifies a configuration `Action` to run for each HTTPS endpoint. Calling `ConfigureHttpsDefaults` multiple times replaces prior `Action`s with the last `Action` specified.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        // certificate is an X509Certificate2
        listenOptions.ServerCertificate = certificate;
    });
});
```

> **NOTE**
> Endpoints created by calling Listen **before** calling ConfigureHttpsDefaults won't have the defaults applied.

Configure(IConfiguration)

Creates a configuration loader for setting up Kestrel that takes an IConfiguration as input. The configuration must be scoped to the configuration section for Kestrel.

ListenOptions.UseHttps

Configure Kestrel to use HTTPS.

`ListenOptions.UseHttps` extensions:

- `UseHttps`: Configure Kestrel to use HTTPS with the default certificate. Throws an exception if no default certificate is configured.
- `UseHttps(string fileName)`
- `UseHttps(string fileName, string password)`
- `UseHttps(string fileName, string password, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(StoreName storeName, string subject)`

- `UseHttps(StoreName storeName, string subject, bool allowInvalid)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(X509Certificate2 serverCertificate)`
- `UseHttps(X509Certificate2 serverCertificate, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(Action<HttpsConnectionAdapterOptions> configureOptions)`

`ListenOptions.UseHttps` parameters:

- `filename` is the path and file name of a certificate file, relative to the directory that contains the app's content files.
- `password` is the password required to access the X.509 certificate data.
- `configureOptions` is an `Action` to configure the `HttpsConnectionAdapterOptions`. Returns the `ListenOptions`.
- `storeName` is the certificate store from which to load the certificate.
- `subject` is the subject name for the certificate.
- `allowInvalid` indicates if invalid certificates should be considered, such as self-signed certificates.
- `location` is the store location to load the certificate from.
- `serverCertificate` is the X.509 certificate.

In production, HTTPS must be explicitly configured. At a minimum, a default certificate must be provided.

Supported configurations described next:

- No configuration
- Replace the default certificate from configuration
- Change the defaults in code

No configuration

Kestrel listens on `http://localhost:5000` and `https://localhost:5001` (if a default cert is available).

Replace the default certificate from configuration

`CreateDefaultBuilder` calls `Configure(context.Configuration.GetSection("Kestrel"))` by default to load Kestrel configuration. A default HTTPS app settings configuration schema is available for Kestrel. Configure multiple endpoints, including the URLs and the certificates to use, either from a file on disk or from a certificate store.

In the following *appsettings.json* example:

- Set **AllowInvalid** to `true` to permit the use of invalid certificates (for example, self-signed certificates).
- Any HTTPS endpoint that doesn't specify a certificate (**HttpsDefaultCert** in the example that follows) falls back to the cert defined under **Certificates** > **Default** or the development certificate.

```
{
  "Kestrel": {
    "Endpoints": {
      "Http": {
        "Url": "http://localhost:5000"
      },
      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      },
      "HttpsInlineCertStore": {
        "Url": "https://localhost:5002",
        "Certificate": {
          "Subject": "<subject; required>",
          "Store": "<certificate store; required>",
          "Location": "<location; defaults to CurrentUser>",
          "AllowInvalid": "<true or false; defaults to false>"
        }
      },
      "HttpsDefaultCert": {
        "Url": "https://localhost:5003"
      },
      "Https": {
        "Url": "https://*:5004",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "<certificate password>"
      }
    }
  }
}
```

An alternative to using **Path** and **Password** for any certificate node is to specify the certificate using certificate store fields. For example, the **Certificates** > **Default** certificate can be specified as:

```
"Default": {
  "Subject": "<subject; required>",
  "Store": "<cert store; required>",
  "Location": "<location; defaults to CurrentUser>",
  "AllowInvalid": "<true or false; defaults to false>"
}
```

Schema notes:

- Endpoints names are case-insensitive. For example, `HTTPS` and `Https` are valid.
- The `Url` parameter is required for each endpoint. The format for this parameter is the same as the top-level `Urls` configuration parameter except that it's limited to a single value.
- These endpoints replace those defined in the top-level `Urls` configuration rather than adding to them. Endpoints defined in code via `Listen` are cumulative with the endpoints defined in the configuration section.

- The `Certificate` section is optional. If the `Certificate` section isn't specified, the defaults defined in earlier scenarios are used. If no defaults are available, the server throws an exception and fails to start.
- The `Certificate` section supports both **Path–Password** and **Subject–Store** certificates.
- Any number of endpoints may be defined in this way so long as they don't cause port conflicts.
- `options.Configure(context.Configuration.GetSection("{SECTION}"))` returns a `KestrelConfigurationLoader` with an `.Endpoint(string name, listenOptions => { })` method that can be used to supplement a configured endpoint's settings:

```
webBuilder.UseKestrel((context, serverOptions) =>
{
    serverOptions.Configure(context.Configuration.GetSection("Kestrel"))
        .Endpoint("HTTPS", listenOptions =>
        {
            listenOptions.HttpsOptions.SslProtocols = SslProtocols.Tls12;
        });
});
```

`KestrelServerOptions.ConfigurationLoader` can be directly accessed to continue iterating on the existing loader, such as the one provided by CreateDefaultBuilder.

- The configuration section for each endpoint is available on the options in the `Endpoint` method so that custom settings may be read.
- Multiple configurations may be loaded by calling `options.Configure(context.Configuration.GetSection("{SECTION}"))` again with another section. Only the last configuration is used, unless `Load` is explicitly called on prior instances. The metapackage doesn't call `Load` so that its default configuration section may be replaced.
- `KestrelConfigurationLoader` mirrors the `Listen` family of APIs from `KestrelServerOptions` as `Endpoint` overloads, so code and config endpoints may be configured in the same place. These overloads don't use names and only consume default settings from configuration.

Change the defaults in code

`ConfigureEndpointDefaults` and `ConfigureHttpsDefaults` can be used to change default settings for `ListenOptions` and `HttpsConnectionAdapterOptions`, including overriding the default certificate specified in the prior scenario. `ConfigureEndpointDefaults` and `ConfigureHttpsDefaults` should be called before any endpoints are configured.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions =>
    {
        // Configure endpoint defaults
    });

    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.SslProtocols = SslProtocols.Tls12;
    });
});
```

Kestrel support for SNI

Server Name Indication (SNI) can be used to host multiple domains on the same IP address and port. For SNI to function, the client sends the host name for the secure session to the server

during the TLS handshake so that the server can provide the correct certificate. The client uses the furnished certificate for encrypted communication with the server during the secure session that follows the TLS handshake.

Kestrel supports SNI via the `ServerCertificateSelector` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate.

SNI support requires:

- Running on target framework `netcoreapp2.1` or later. On `net461` or later, the callback is invoked but the `name` is always `null`. The `name` is also `null` if the client doesn't provide the host name parameter in the TLS handshake.
- All websites run on the same Kestrel instance. Kestrel doesn't support sharing an IP address and port across multiple instances without a reverse proxy.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenAnyIP(5005, listenOptions =>
    {
        listenOptions.UseHttps(httpsOptions =>
        {
            var localhostCert = CertificateLoader.LoadFromStoreCert(
                "localhost", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var exampleCert = CertificateLoader.LoadFromStoreCert(
                "example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var subExampleCert = CertificateLoader.LoadFromStoreCert(
                "sub.example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var certs = new Dictionary<string, X509Certificate2>(
                StringComparer.OrdinalIgnoreCase);
            certs["localhost"] = localhostCert;
            certs["example.com"] = exampleCert;
            certs["sub.example.com"] = subExampleCert;

            httpsOptions.ServerCertificateSelector = (connectionContext, name) =>
            {
                if (name != null && certs.TryGetValue(name, out var cert))
                {
                    return cert;
                }

                return exampleCert;
            };
        });
    });
});
```

Connection logging

Call UseConnectionLogging to emit Debug level logs for byte-level communication on a connection. Connection logging is helpful for troubleshooting problems in low-level communication, such as during TLS encryption and behind proxies. If `UseConnectionLogging` is placed before `UseHttps`, encrypted traffic is logged. If `UseConnectionLogging` is placed after `UseHttps`, decrypted traffic is logged.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseConnectionLogging();
    });
});
```

Bind to a TCP socket

The Listen method binds to a TCP socket, and an options lambda permits X.509 certificate configuration:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(serverOptions =>
            {
                serverOptions.Listen(IPAddress.Loopback, 5000);
                serverOptions.Listen(IPAddress.Loopback, 5001,
                    listenOptions =>
                    {
                        listenOptions.UseHttps("testCert.pfx",
                            "testPassword");
                    });
            })
            .UseStartup<Startup>();
        });
```

The example configures HTTPS for an endpoint with ListenOptions. Use the same API to configure other Kestrel settings for specific endpoints.

On Windows, self-signed certificates can be created using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

On macOS, Linux, and Windows, certificates can be created using OpenSSL.

Bind to a Unix socket

Listen on a Unix socket with ListenUnixSocket for improved performance with Nginx, as shown in this example:

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock");
    serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock",
        listenOptions =>
        {
            listenOptions.UseHttps("testCert.pfx",
                "testpassword");
        });
})
```

- In the Nginx configuration file, set the `server` > `location` > `proxy_pass` entry to `http://unix:/tmp/{KESTREL SOCKET}:/;`. `{KESTREL SOCKET}` is the name of the socket provided

to ListenUnixSocket (for example, `kestrel-test.sock` in the preceding example).

- Ensure that the socket is writeable by Nginx (for example, `chmod go+w /tmp/kestrel-test.sock`).

Port 0

When the port number `0` is specified, Kestrel dynamically binds to an available port. The following example shows how to determine which port Kestrel actually bound at runtime:

```
public void Configure(IApplicationBuilder app)
{
    var serverAddressesFeature =
        app.ServerFeatures.Get<IServerAddressesFeature>();

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/html";
        await context.Response
            .WriteAsync("<!DOCTYPE html><html lang=\"en\"><head>" +
                "<title></title></head><body><p>Hosted by Kestrel</p>");

        if (serverAddressesFeature != null)
        {
            await context.Response
                .WriteAsync("<p>Listening on the following addresses: " +
                    string.Join(", ", serverAddressesFeature.Addresses) +
                    "</p>");
        }

        await context.Response.WriteAsync("<p>Request URL: " +
            $"{context.Request.GetDisplayUrl()}<p>");
    });
}
```

When the app is run, the console window output indicates the dynamic port where the app can be reached:

```
Listening on the following addresses: http://127.0.0.1:48508
```

Limitations

Configure endpoints with the following approaches:

- UseUrls
- `--urls` command-line argument
- `urls` host configuration key
- `ASPNETCORE_URLS` environment variable

These methods are useful for making code work with servers other than Kestrel. However, be aware of the following limitations:

- HTTPS can't be used with these approaches unless a default certificate is provided in the HTTPS endpoint configuration (for example, using `KestrelServerOptions` configuration or a configuration file as shown earlier in this topic).
- When both the `Listen` and `UseUrls` approaches are used simultaneously, the `Listen` endpoints override the `UseUrls` endpoints.

IIS endpoint configuration

When using IIS, the URL bindings for IIS override bindings are set by either `Listen` or `UseUrls`. For more information, see the ASP.NET Core Module topic.

ListenOptions.Protocols

The `Protocols` property establishes the HTTP protocols (`HttpProtocols`) enabled on a connection endpoint or for the server. Assign a value to the `Protocols` property from the `HttpProtocols` enum.

`HTTPPROTOCOLS` ENUM VALUE	CONNECTION PROTOCOL PERMITTED
`Http1`	HTTP/1.1 only. Can be used with or without TLS.
`Http2`	HTTP/2 only. May be used without TLS only if the client supports a Prior Knowledge mode.
`Http1AndHttp2`	HTTP/1.1 and HTTP/2. HTTP/2 requires the client to select HTTP/2 in the TLS Application-Layer Protocol Negotiation (ALPN) handshake; otherwise, the connection defaults to HTTP/1.1.

The default `ListenOptions.Protocols` value for any endpoint is `HttpProtocols.Http1AndHttp2`.

TLS restrictions for HTTP/2:

- TLS version 1.2 or later
- Renegotiation disabled
- Compression disabled
- Minimum ephemeral key exchange sizes:
  - Elliptic curve Diffie-Hellman (ECDHE) [RFC4492]: 224 bits minimum
  - Finite field Diffie-Hellman (DHE) [`TLS12`]: 2048 bits minimum
- Cipher suite not prohibited.

`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` [`TLS-ECDHE`] with the P-256 elliptic curve [`FIPS186`] is supported by default.

The following example permits HTTP/1.1 and HTTP/2 connections on port 8000. Connections are secured by TLS with a supplied certificate:

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

Use Connection Middleware to filter TLS handshakes on a per-connection basis for specific ciphers if required.

The following example throws NotSupportedException for any cipher algorithm that the app doesn't support. Alternatively, define and compare ITlsHandshakeFeature.CipherAlgorithm to a list of acceptable cipher suites.

No encryption is used with a CipherAlgorithmType.Null cipher algorithm.

```
// using System.Net;
// using Microsoft.AspNetCore.Connections;

webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
        listenOptions.UseTlsFilter();
    });
});
```

```
using System;
using System.Security.Authentication;
using Microsoft.AspNetCore.Connections.Features;

namespace Microsoft.AspNetCore.Connections
{
    public static class TlsFilterConnectionMiddlewareExtensions
    {
        public static IConnectionBuilder UseTlsFilter(
            this IConnectionBuilder builder)
        {
            return builder.Use((connection, next) =>
            {
                var tlsFeature = connection.Features.Get<ITlsHandshakeFeature>();

                if (tlsFeature.CipherAlgorithm == CipherAlgorithmType.Null)
                {
                    throw new NotSupportedException("Prohibited cipher: " +
                        tlsFeature.CipherAlgorithm);
                }

                return next();
            });
        }
    }
}
```

Connection filtering can also be configured via an IConnectionBuilder lambda:

```
// using System;
// using System.Net;
// using System.Security.Authentication;
// using Microsoft.AspNetCore.Connections;
// using Microsoft.AspNetCore.Connections.Features;

webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
        listenOptions.Use((context, next) =>
        {
            var tlsFeature = context.Features.Get<ITlsHandshakeFeature>();

            if (tlsFeature.CipherAlgorithm == CipherAlgorithmType.Null)
            {
                throw new NotSupportedException(
                    $"Prohibited cipher: {tlsFeature.CipherAlgorithm}");
            }

            return next();
        });
    });
});
```

On Linux, CipherSuitesPolicy can be used to filter TLS handshakes on a per-connection basis:

```
// using System.Net.Security;
// using Microsoft.AspNetCore.Hosting;
// using Microsoft.AspNetCore.Server.Kestrel.Core;
// using Microsoft.Extensions.DependencyInjection;
// using Microsoft.Extensions.Hosting;

webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.OnAuthenticate = (context, sslOptions) =>
        {
            sslOptions.CipherSuitesPolicy = new CipherSuitesPolicy(
                new[]
                {
                    TlsCipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
                    TlsCipherSuite.TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
                    // ...
                });
        };
    });
});
```

Set the protocol from configuration

`CreateDefaultBuilder` calls `serverOptions.Configure(context.Configuration.GetSection("Kestrel"))` by default to load Kestrel configuration.

The following *appsettings.json* example establishes HTTP/1.1 as the default connection protocol for all endpoints:

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1"
    }
  }
}
```

The following *appsettings.json* example establishes the HTTP/1.1 connection protocol for a specific endpoint:

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsDefaultCert": {
        "Url": "https://localhost:5001",
        "Protocols": "Http1"
      }
    }
  }
}
```

Protocols specified in code override values set by configuration.

Transport configuration

For projects that require the use of Libuv (UseLibuv):

- Add a dependency for the Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv package to the app's project file:

  ```
  <PackageReference Include="Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv"
                    Version="{VERSION}" />
  ```

- Call UseLibuv on the `IWebHostBuilder`:

  ```
  public class Program
  {
       public static void Main(string[] args)
       {
           CreateHostBuilder(args).Build().Run();
       }

       public static IHostBuilder CreateHostBuilder(string[] args) =>
           Host.CreateDefaultBuilder(args)
               .ConfigureWebHostDefaults(webBuilder =>
               {
                   webBuilder.UseLibuv();
                   webBuilder.UseStartup<Startup>();
               });
  }
  ```

URL prefixes

When using `UseUrls`, `--urls` command-line argument, `urls` host configuration key, or `ASPNETCORE_URLS` environment variable, the URL prefixes can be in any of the following formats.

Only HTTP URL prefixes are valid. Kestrel doesn't support HTTPS when configuring URL bindings using `UseUrls`.

- IPv4 address with port number

  ```
  http://65.55.39.10:80/
  ```

  `0.0.0.0` is a special case that binds to all IPv4 addresses.

- IPv6 address with port number

  ```
  http://[0:0:0:0:0:ffff:4137:270a]:80/
  ```

  `[::]` is the IPv6 equivalent of IPv4 `0.0.0.0`.

- Host name with port number

  ```
  http://contoso.com:80/
  http://*:80/
  ```

  Host names, `*`, and `+`, aren't special. Anything not recognized as a valid IP address or `localhost` binds to all IPv4 and IPv6 IPs. To bind different host names to different ASP.NET Core apps on the same port, use HTTP.sys or a reverse proxy server, such as IIS, Nginx, or Apache.

  > **WARNING**
  > Hosting in a reverse proxy configuration requires host filtering.

- Host `localhost` name with port number or loopback IP with port number

  ```
  http://localhost:5000/
  http://127.0.0.1:5000/
  http://[::1]:5000/
  ```

  When `localhost` is specified, Kestrel attempts to bind to both IPv4 and IPv6 loopback interfaces. If the requested port is in use by another service on either loopback interface, Kestrel fails to start. If either loopback interface is unavailable for any other reason (most commonly because IPv6 isn't supported), Kestrel logs a warning.

Host filtering

While Kestrel supports configuration based on prefixes such as `http://example.com:5000`, Kestrel largely ignores the host name. Host `localhost` is a special case used for binding to loopback addresses. Any host other than an explicit IP address binds to all public IP addresses. `Host` headers aren't validated.

As a workaround, use Host Filtering Middleware. Host Filtering Middleware is provided by the Microsoft.AspNetCore.HostFiltering package, which is implicitly provided for ASP.NET Core apps. The middleware is added by CreateDefaultBuilder, which calls AddHostFiltering:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Host Filtering Middleware is disabled by default. To enable the middleware, define an `AllowedHosts` key in *appsettings.json*/*appsettings.<EnvironmentName>.json*. The value is a semicolon-delimited list of host names without port numbers:

appsettings.json:

```
{
  "AllowedHosts": "example.com;localhost"
}
```

> **NOTE**
>
> Forwarded Headers Middleware also has an AllowedHosts option. Forwarded Headers Middleware and Host Filtering Middleware have similar functionality for different scenarios. Setting `AllowedHosts` with Forwarded Headers Middleware is appropriate when the `Host` header isn't preserved while forwarding requests with a reverse proxy server or load balancer. Setting `AllowedHosts` with Host Filtering Middleware is appropriate when Kestrel is used as a public-facing edge server or when the `Host` header is directly forwarded.
>
> For more information on Forwarded Headers Middleware, see Configure ASP.NET Core to work with proxy servers and load balancers.

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the web server that's included by default in ASP.NET Core project templates.

Kestrel supports the following scenarios:

- HTTPS
- Opaque upgrade used to enable WebSockets
- Unix sockets for high performance behind Nginx
- HTTP/2 (except on macOS†)

†HTTP/2 will be supported on macOS in a future release.

Kestrel is supported on all platforms and versions that .NET Core supports.

View or download sample code (how to download)

HTTP/2 support

HTTP/2 is available for ASP.NET Core apps if the following base requirements are met:

- Operating system†
  - Windows Server 2016/Windows 10 or later‡

    - Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
- Target framework: .NET Core 2.2 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

†HTTP/2 will be supported on macOS in a future release. ‡Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

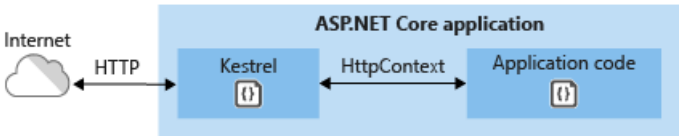
If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

HTTP/2 is disabled by default. For more information on configuration, see the Kestrel options and ListenOptions.Protocols sections.

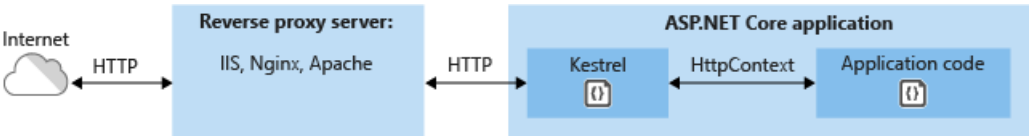
When to use Kestrel with a reverse proxy

Kestrel can be used by itself or with a *reverse proxy server*, such as Internet Information Services (IIS), Nginx, or Apache. A reverse proxy server receives HTTP requests from the network and forwards them to Kestrel.

Kestrel used as an edge (Internet-facing) web server:



Kestrel used in a reverse proxy configuration:



Either configuration, with or without a reverse proxy server, is a supported hosting configuration.

Kestrel used as an edge server without a reverse proxy server doesn't support sharing the same IP and port among multiple processes. When Kestrel is configured to listen on a port, Kestrel handles all of the traffic for that port regardless of requests' `Host` headers. A reverse proxy that can share ports has the ability to forward requests to Kestrel on a unique IP and port.

Even if a reverse proxy server isn't required, using a reverse proxy server might be a good choice.

A reverse proxy:

- Can limit the exposed public surface area of the apps that it hosts.
- Provide an additional layer of configuration and defense.
- Might integrate better with existing infrastructure.
- Simplify load balancing and secure communication (HTTPS) configuration. Only the reverse proxy server requires an X.509 certificate, and that server can communicate with the app's servers on the internal network using plain HTTP.

> **WARNING**
>
> Hosting in a reverse proxy configuration requires host filtering.

How to use Kestrel in ASP.NET Core apps

The Microsoft.AspNetCore.Server.Kestrel package is included in the Microsoft.AspNetCore.App metapackage.

ASP.NET Core project templates use Kestrel by default. In *Program.cs*, the template code calls CreateDefaultBuilder, which calls UseKestrel behind the scenes.

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

For more information on `CreateDefaultBuilder` and building the host, see the *Set up a host* section of ASP.NET Core Web Host.

To provide additional configuration after calling `CreateDefaultBuilder`, use `ConfigureKestrel`:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            // Set properties and call methods on serverOptions
        });
```

If the app doesn't call `CreateDefaultBuilder` to set up the host, call UseKestrel **before** calling `ConfigureKestrel`:

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseKestrel()
        .UseIISIntegration()
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            // Set properties and call methods on serverOptions
        })
        .Build();

    host.Run();
}
```

Kestrel options

The Kestrel web server has constraint configuration options that are especially useful in Internet-facing deployments.

Set constraints on the Limits property of the KestrelServerOptions class. The `Limits` property holds an instance of the KestrelServerLimits class.

The following examples use the Microsoft.AspNetCore.Server.Kestrel.Core namespace:

```
using Microsoft.AspNetCore.Server.Kestrel.Core;
```

Kestrel options, which are configured in C# code in the following examples, can also be set using a configuration provider. For example, the File Configuration Provider can load Kestrel configuration from an *appsettings.json* or *appsettings.{Environment}.json* file:

```
{
  "Kestrel": {
    "Limits": {
      "MaxConcurrentConnections": 100,
      "MaxConcurrentUpgradedConnections": 100
    }
  }
}
```

Use **one** of the following approaches:

- Configure Kestrel in `Startup.ConfigureServices`:

  1. Inject an instance of `IConfiguration` into the `Startup` class. The following example assumes that the injected configuration is assigned to the `Configuration` property.

  2. In `Startup.ConfigureServices`, load the `Kestrel` section of configuration into Kestrel's configuration:

     ```
     using Microsoft.Extensions.Configuration

     public class Startup
     {
         public Startup(IConfiguration configuration)
         {
             Configuration = configuration;
         }

         public IConfiguration Configuration { get; }

         public void ConfigureServices(IServiceCollection services)
         {
             services.Configure<KestrelServerOptions>(
                 Configuration.GetSection("Kestrel"));
         }

         public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
         {
             ...
         }
     }
     ```

- Configure Kestrel when building the host:

  In *Program.cs*, load the `Kestrel` section of configuration into Kestrel's configuration:

```
// using Microsoft.Extensions.DependencyInjection;

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureServices((context, services) =>
        {
            services.Configure<KestrelServerOptions>(
                context.Configuration.GetSection("Kestrel"));
        })
        .UseStartup<Startup>();
```

Both of the preceding approaches work with any configuration provider.

Keep-alive timeout

KeepAliveTimeout

Gets or sets the keep-alive timeout. Defaults to 2 minutes.

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

Maximum client connections

MaxConcurrentConnections MaxConcurrentUpgradedConnections

The maximum number of concurrent open TCP connections can be set for the entire app with the following code:

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

There's a separate limit for connections that have been upgraded from HTTP or HTTPS to

another protocol (for example, on a WebSockets request). After a connection is upgraded, it isn't counted against the `MaxConcurrentConnections` limit.

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

The maximum number of connections is unlimited (null) by default.

Maximum request body size

MaxRequestBodySize

The default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB.

The recommended approach to override the limit in an ASP.NET Core MVC app is to use the RequestSizeLimitAttribute attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

Here's an example that shows how to configure the constraint for the app on every request:

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

Override the setting on a specific request in middleware:

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;

    var minRequestRateFeature =
        context.Features.Get<IHttpMinRequestBodyDataRateFeature>();
    var minResponseRateFeature =
        context.Features.Get<IHttpMinResponseDataRateFeature>();

    if (minRequestRateFeature != null)
    {
        minRequestRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    if (minResponseRateFeature != null)
    {
        minResponseRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }
```

An exception is thrown if the app configures the limit on a request after the app has started to read the request. There's an `IsReadOnly` property that indicates if the `MaxRequestBodySize` property is in read-only state, meaning it's too late to configure the limit.

When an app is run out-of-process behind the ASP.NET Core Module, Kestrel's request body size limit is disabled because IIS already sets the limit.

Minimum request body data rate

MinRequestBodyDataRate MinResponseDataRate

Kestrel checks every second if data is arriving at the specified rate in bytes/second. If the rate drops below the minimum, the connection is timed out. The grace period is the amount of time that Kestrel gives the client to increase its send rate up to the minimum; the rate isn't checked during that time. The grace period helps avoid dropping connections that are initially sending data at a slow rate due to TCP slow-start.

The default minimum rate is 240 bytes/second with a 5 second grace period.

A minimum rate also applies to the response. The code to set the request limit and the response limit is the same except for having `RequestBody` or `Response` in the property and interface names.

Here's an example that shows how to configure the minimum data rates in *Program.cs*:

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

Override the minimum rate limits per request in middleware:

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;

    var minRequestRateFeature =
        context.Features.Get<IHttpMinRequestBodyDataRateFeature>();
    var minResponseRateFeature =
        context.Features.Get<IHttpMinResponseDataRateFeature>();

    if (minRequestRateFeature != null)
    {
        minRequestRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    if (minResponseRateFeature != null)
    {
        minResponseRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }
```

Neither rate feature referenced in the prior sample are present in `HttpContext.Features` for HTTP/2 requests because modifying rate limits on a per-request basis isn't supported for HTTP/2 due to the protocol's support for request multiplexing. Server-wide rate limits configured via `KestrelServerOptions.Limits` still apply to both HTTP/1.x and HTTP/2 connections.

Request headers timeout

RequestHeadersTimeout

Gets or sets the maximum amount of time the server spends receiving request headers. Defaults to 30 seconds.

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
    serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
    serverOptions.Limits.MinRequestBodyDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate =
        new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

Maximum streams per connection

`Http2.MaxStreamsPerConnection` limits the number of concurrent request streams per HTTP/2 connection. Excess streams are refused.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.Limits.Http2.MaxStreamsPerConnection = 100;
        });
```

The default value is 100.

Header table size

The HPACK decoder decompresses HTTP headers for HTTP/2 connections. `Http2.HeaderTableSize` limits the size of the header compression table that the HPACK decoder uses. The value is provided in octets and must be greater than zero (0).

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.Limits.Http2.HeaderTableSize = 4096;
        });
```

The default value is 4096.

Maximum frame size

`Http2.MaxFrameSize` indicates the maximum size of the HTTP/2 connection frame payload to receive. The value is provided in octets and must be between 2^14 (16,384) and 2^24-1 (16,777,215).

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.Limits.Http2.MaxFrameSize = 16384;
        });
```

The default value is 2^14 (16,384).

Maximum request header size

`Http2.MaxRequestHeaderFieldSize` indicates the maximum allowed size in octets of request header values. This limit applies to both name and value together in their compressed and uncompressed representations. The value must be greater than zero (0).

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.Limits.Http2.MaxRequestHeaderFieldSize = 8192;
        });
```

The default value is 8,192.

Initial connection window size

`Http2.InitialConnectionWindowSize` indicates the maximum request body data in bytes the server buffers at one time aggregated across all requests (streams) per connection. Requests are also limited by `Http2.InitialStreamWindowSize`. The value must be greater than or equal to 65,535 and less than 2^31 (2,147,483,648).

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.Limits.Http2.InitialConnectionWindowSize = 131072;
        });
```

The default value is 128 KB (131,072).

Initial stream window size

`Http2.InitialStreamWindowSize` indicates the maximum request body data in bytes the server buffers at one time per request (stream). Requests are also limited by `Http2.InitialStreamWindowSize`. The value must be greater than or equal to 65,535 and less than 2^31 (2,147,483,648).

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.Limits.Http2.InitialStreamWindowSize = 98304;
        });
```

The default value is 96 KB (98,304).

Synchronous I/O

AllowSynchronousIO controls whether synchronous I/O is allowed for the request and response. The default value is `true`.

> **WARNING**
> A large number of blocking synchronous I/O operations can lead to thread pool starvation, which makes the app unresponsive. Only enable `AllowSynchronousIO` when using a library that doesn't support asynchronous I/O.

The following example enables synchronous I/O:

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.AllowSynchronousIO = true;
});
```

For information about other Kestrel options and limits, see:

- KestrelServerOptions
- KestrelServerLimits
- ListenOptions

Endpoint configuration

By default, ASP.NET Core binds to:

- `http://localhost:5000`
- `https://localhost:5001` (when a local development certificate is present)

Specify URLs using the:

- `ASPNETCORE_URLS` environment variable.
- `--urls` command-line argument.
- `urls` host configuration key.
- `UseUrls` extension method.

The value provided using these approaches can be one or more HTTP and HTTPS endpoints (HTTPS if a default cert is available). Configure the value as a semicolon-separated list (for example, `"Urls": "http://localhost:8000;http://localhost:8001"`).

For more information on these approaches, see Server URLs and Override configuration.

A development certificate is created:

- When the .NET Core SDK is installed.
- The dev-certs tool is used to create a certificate.

Some browsers require granting explicit permission to trust the local development certificate.

Project templates configure apps to run on HTTPS by default and include HTTPS redirection and HSTS support.

Call Listen or ListenUnixSocket methods on KestrelServerOptions to configure URL prefixes and ports for Kestrel.

`UseUrls`, the `--urls` command-line argument, `urls` host configuration key, and the `ASPNETCORE_URLS` environment variable also work but have the limitations noted later in this section (a default certificate must be available for HTTPS endpoint configuration).

`KestrelServerOptions` configuration:

ConfigureEndpointDefaults(Action<ListenOptions>)

Specifies a configuration `Action` to run for each specified endpoint. Calling `ConfigureEndpointDefaults` multiple times replaces prior `Action`s with the last `Action` specified.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.ConfigureEndpointDefaults(listenOptions =>
            {
                // Configure endpoint defaults
            });
        });
```

> **NOTE**
> Endpoints created by calling Listen **before** calling ConfigureEndpointDefaults won't have the defaults applied.

ConfigureHttpsDefaults(Action<HttpsConnectionAdapterOptions>)

Specifies a configuration `Action` to run for each HTTPS endpoint. Calling `ConfigureHttpsDefaults` multiple times replaces prior `Action`s with the last `Action` specified.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.ConfigureHttpsDefaults(listenOptions =>
            {
                // certificate is an X509Certificate2
                listenOptions.ServerCertificate = certificate;
            });
        });
```

> **NOTE**
> Endpoints created by calling Listen **before** calling ConfigureHttpsDefaults won't have the defaults applied.

Configure(IConfiguration)

Creates a configuration loader for setting up Kestrel that takes an IConfiguration as input. The configuration must be scoped to the configuration section for Kestrel.

ListenOptions.UseHttps

Configure Kestrel to use HTTPS.

`ListenOptions.UseHttps` extensions:

- `UseHttps`: Configure Kestrel to use HTTPS with the default certificate. Throws an exception if no default certificate is configured.
- `UseHttps(string fileName)`
- `UseHttps(string fileName, string password)`
- `UseHttps(string fileName, string password, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(StoreName storeName, string subject)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(X509Certificate2 serverCertificate)`
- `UseHttps(X509Certificate2 serverCertificate, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(Action<HttpsConnectionAdapterOptions> configureOptions)`

`ListenOptions.UseHttps` parameters:

- `filename` is the path and file name of a certificate file, relative to the directory that contains the app's content files.
- `password` is the password required to access the X.509 certificate data.
- `configureOptions` is an `Action` to configure the `HttpsConnectionAdapterOptions`. Returns the `ListenOptions`.
- `storeName` is the certificate store from which to load the certificate.
- `subject` is the subject name for the certificate.
- `allowInvalid` indicates if invalid certificates should be considered, such as self-signed certificates.
- `location` is the store location to load the certificate from.
- `serverCertificate` is the X.509 certificate.

In production, HTTPS must be explicitly configured. At a minimum, a default certificate must be provided.

Supported configurations described next:

- No configuration
- Replace the default certificate from configuration
- Change the defaults in code

No configuration

Kestrel listens on `http://localhost:5000` and `https://localhost:5001` (if a default cert is available).

Replace the default certificate from configuration

`CreateDefaultBuilder` calls `Configure(context.Configuration.GetSection("Kestrel"))` by default to load Kestrel configuration. A default HTTPS app settings configuration schema is available for Kestrel. Configure multiple endpoints, including the URLs and the certificates to use, either from a file on disk or from a certificate store.

In the following *appsettings.json* example:

- Set **AllowInvalid** to `true` to permit the use of invalid certificates (for example, self-signed certificates).
- Any HTTPS endpoint that doesn't specify a certificate (**HttpsDefaultCert** in the example that follows) falls back to the cert defined under **Certificates** > **Default** or the development certificate.

```
{
  "Kestrel": {
    "Endpoints": {
      "Http": {
        "Url": "http://localhost:5000"
      },

      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      },

      "HttpsInlineCertStore": {
        "Url": "https://localhost:5002",
        "Certificate": {
          "Subject": "<subject; required>",
          "Store": "<certificate store; required>",
          "Location": "<location; defaults to CurrentUser>",
          "AllowInvalid": "<true or false; defaults to false>"
        }
      },

      "HttpsDefaultCert": {
        "Url": "https://localhost:5003"
      },

      "Https": {
        "Url": "https://*:5004",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "<certificate password>"
      }
    }
  }
}
```

An alternative to using **Path** and **Password** for any certificate node is to specify the certificate using certificate store fields. For example, the **Certificates** > **Default** certificate can be specified as:

```
"Default": {
  "Subject": "<subject; required>",
  "Store": "<cert store; required>",
  "Location": "<location; defaults to CurrentUser>",
  "AllowInvalid": "<true or false; defaults to false>"
}
```

Schema notes:

- Endpoints names are case-insensitive. For example, `HTTPS` and `Https` are valid.
- The `Url` parameter is required for each endpoint. The format for this parameter is the same as the top-level `Urls` configuration parameter except that it's limited to a single value.
- These endpoints replace those defined in the top-level `Urls` configuration rather than adding to them. Endpoints defined in code via `Listen` are cumulative with the endpoints defined in the configuration section.
- The `Certificate` section is optional. If the `Certificate` section isn't specified, the defaults defined in earlier scenarios are used. If no defaults are available, the server throws an exception and fails to start.
- The `Certificate` section supports both **Path–Password** and **Subject–Store** certificates.
- Any number of endpoints may be defined in this way so long as they don't cause port conflicts.
- `options.Configure(context.Configuration.GetSection("{SECTION}"))` returns a `KestrelConfigurationLoader` with an `.Endpoint(string name, listenOptions => { })` method that can be used to supplement a configured endpoint's settings:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel((context, serverOptions) =>
        {
            serverOptions.Configure(context.Configuration.GetSection("Kestrel"))
                .Endpoint("HTTPS", listenOptions =>
                {
                    listenOptions.HttpsOptions.SslProtocols = SslProtocols.Tls12;
                });
        });
```

`KestrelServerOptions.ConfigurationLoader` can be directly accessed to continue iterating on the existing loader, such as the one provided by CreateDefaultBuilder.

- The configuration section for each endpoint is available on the options in the `Endpoint` method so that custom settings may be read.
- Multiple configurations may be loaded by calling `options.Configure(context.Configuration.GetSection("{SECTION}"))` again with another section. Only the last configuration is used, unless `Load` is explicitly called on prior instances. The metapackage doesn't call `Load` so that its default configuration section may be replaced.
- `KestrelConfigurationLoader` mirrors the `Listen` family of APIs from `KestrelServerOptions` as `Endpoint` overloads, so code and config endpoints may be configured in the same place. These overloads don't use names and only consume default settings from configuration.

Change the defaults in code

`ConfigureEndpointDefaults` and `ConfigureHttpsDefaults` can be used to change default settings for `ListenOptions` and `HttpsConnectionAdapterOptions`, including overriding the default certificate specified in the prior scenario. `ConfigureEndpointDefaults` and `ConfigureHttpsDefaults` should be called before any endpoints are configured.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel((context, serverOptions) =>
        {
            serverOptions.ConfigureEndpointDefaults(listenOptions =>
            {
                // Configure endpoint defaults
            });

            serverOptions.ConfigureHttpsDefaults(listenOptions =>
            {
                listenOptions.SslProtocols = SslProtocols.Tls12;
            });
        });
```

Kestrel support for SNI

Server Name Indication (SNI) can be used to host multiple domains on the same IP address and port. For SNI to function, the client sends the host name for the secure session to the server during the TLS handshake so that the server can provide the correct certificate. The client uses the furnished certificate for encrypted communication with the server during the secure session that follows the TLS handshake.

Kestrel supports SNI via the `ServerCertificateSelector` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate.

SNI support requires:

- Running on target framework `netcoreapp2.1` or later. On `net461` or later, the callback is invoked but the `name` is always `null`. The `name` is also `null` if the client doesn't provide the host name parameter in the TLS handshake.
- All websites run on the same Kestrel instance. Kestrel doesn't support sharing an IP address and port across multiple instances without a reverse proxy.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.ListenAnyIP(5005, listenOptions =>
            {
                listenOptions.UseHttps(httpsOptions =>
                {
                    var localhostCert = CertificateLoader.LoadFromStoreCert(
                        "localhost", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var exampleCert = CertificateLoader.LoadFromStoreCert(
                        "example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var subExampleCert = CertificateLoader.LoadFromStoreCert(
                        "sub.example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var certs = new Dictionary<string, X509Certificate2>(
                        StringComparer.OrdinalIgnoreCase);
                    certs["localhost"] = localhostCert;
                    certs["example.com"] = exampleCert;
                    certs["sub.example.com"] = subExampleCert;

                    httpsOptions.ServerCertificateSelector = (connectionContext, name) =>
                    {
                        if (name != null && certs.TryGetValue(name, out var cert))
                        {
                            return cert;
                        }

                        return exampleCert;
                    };
                });
            });
        });
```

Connection logging

Call UseConnectionLogging to emit Debug level logs for byte-level communication on a connection. Connection logging is helpful for troubleshooting problems in low-level communication, such as during TLS encryption and behind proxies. If `UseConnectionLogging` is placed before `UseHttps`, encrypted traffic is logged. If `UseConnectionLogging` is placed after `UseHttps`, decrypted traffic is logged.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseConnectionLogging();
    });
});
```

Bind to a TCP socket

The Listen method binds to a TCP socket, and an options lambda permits X.509 certificate configuration:

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.Listen(IPAddress.Loopback, 5000);
            serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testPassword");
            });
        });
```

The example configures HTTPS for an endpoint with ListenOptions. Use the same API to configure other Kestrel settings for specific endpoints.

On Windows, self-signed certificates can be created using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

On macOS, Linux, and Windows, certificates can be created using OpenSSL.

Bind to a Unix socket

Listen on a Unix socket with ListenUnixSocket for improved performance with Nginx, as shown in this example:

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock");
    serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock", listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testpassword");
    });
});
```

- In the Nginx confiuguration file, set the `server` > `location` > `proxy_pass` entry to `http://unix:/tmp/{KESTREL SOCKET}:/;`. `{KESTREL SOCKET}` is the name of the socket provided to ListenUnixSocket (for example, `kestrel-test.sock` in the preceding example).
- Ensure that the socket is writeable by Nginx (for example, `chmod go+w /tmp/kestrel-test.sock`).

Port 0

When the port number `0` is specified, Kestrel dynamically binds to an available port. The following example shows how to determine which port Kestrel actually bound at runtime:

```
public void Configure(IApplicationBuilder app)
{
    var serverAddressesFeature =
        app.ServerFeatures.Get<IServerAddressesFeature>();

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/html";
        await context.Response
            .WriteAsync("<!DOCTYPE html><html lang=\"en\"><head>" +
                "<title></title></head><body><p>Hosted by Kestrel</p>");

        if (serverAddressesFeature != null)
        {
            await context.Response
                .WriteAsync("<p>Listening on the following addresses: " +
                    string.Join(", ", serverAddressesFeature.Addresses) +
                    "</p>");
        }

        await context.Response.WriteAsync("<p>Request URL: " +
            $"{context.Request.GetDisplayUrl()}<p>");
    });
}
```

When the app is run, the console window output indicates the dynamic port where the app can be reached:

```
Listening on the following addresses: http://127.0.0.1:48508
```

Limitations

Configure endpoints with the following approaches:

- UseUrls
- `--urls` command-line argument
- `urls` host configuration key
- `ASPNETCORE_URLS` environment variable

These methods are useful for making code work with servers other than Kestrel. However, be aware of the following limitations:

- HTTPS can't be used with these approaches unless a default certificate is provided in the HTTPS endpoint configuration (for example, using `KestrelServerOptions` configuration or a configuration file as shown earlier in this topic).
- When both the `Listen` and `UseUrls` approaches are used simultaneously, the `Listen` endpoints override the `UseUrls` endpoints.

IIS endpoint configuration

When using IIS, the URL bindings for IIS override bindings are set by either `Listen` or `UseUrls`. For more information, see the ASP.NET Core Module topic.

ListenOptions.Protocols

The `Protocols` property establishes the HTTP protocols (`HttpProtocols`) enabled on a connection endpoint or for the server. Assign a value to the `Protocols` property from the `HttpProtocols` enum.

`HttpProtocols` enum value	Connection protocol permitted
`Http1`	HTTP/1.1 only. Can be used with or without TLS.
`Http2`	HTTP/2 only. May be used without TLS only if the client supports a Prior Knowledge mode.
`Http1AndHttp2`	HTTP/1.1 and HTTP/2. HTTP/2 requires a TLS and Application-Layer Protocol Negotiation (ALPN) connection; otherwise, the connection defaults to HTTP/1.1.

The default protocol is HTTP/1.1.

TLS restrictions for HTTP/2:

- TLS version 1.2 or later
- Renegotiation disabled
- Compression disabled
- Minimum ephemeral key exchange sizes:
  - Elliptic curve Diffie-Hellman (ECDHE) [RFC4492]: 224 bits minimum
  - Finite field Diffie-Hellman (DHE) [`TLS12`]: 2048 bits minimum
- Cipher suite not blocked

`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` [`TLS-ECDHE`] with the P-256 elliptic curve [`FIPS186`] is supported by default.

The following example permits HTTP/1.1 and HTTP/2 connections on port 8000. Connections are secured by TLS with a supplied certificate:

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2;
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

Optionally create an `IConnectionAdapter` implementation to filter TLS handshakes on a per-connection basis for specific ciphers:

```
.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2;
        listenOptions.UseHttps("testCert.pfx", "testPassword");
        listenOptions.ConnectionAdapters.Add(new TlsFilterAdapter());
    });
});
```

```
private class TlsFilterAdapter : IConnectionAdapter
{
    public bool IsHttps => false;

    public Task<IAdaptedConnection> OnConnectionAsync(ConnectionAdapterContext context)
    {
        var tlsFeature = context.Features.Get<ITlsHandshakeFeature>();

        // Throw NotSupportedException for any cipher algorithm that the app doesn't
        // wish to support. Alternatively, define and compare
        // ITlsHandshakeFeature.CipherAlgorithm to a list of acceptable cipher
        // suites.
        //
        // No encryption is used with a CipherAlgorithmType.Null cipher algorithm.
        if (tlsFeature.CipherAlgorithm == CipherAlgorithmType.Null)
        {
            throw new NotSupportedException("Prohibited cipher: " +
tlsFeature.CipherAlgorithm);
        }

        return Task.FromResult<IAdaptedConnection>(new
AdaptedConnection(context.ConnectionStream));
    }

    private class AdaptedConnection : IAdaptedConnection
    {
        public AdaptedConnection(Stream adaptedStream)
        {
            ConnectionStream = adaptedStream;
        }

        public Stream ConnectionStream { get; }

        public void Dispose()
        {
        }
    }
}
```

Set the protocol from configuration

CreateDefaultBuilder calls `serverOptions.Configure(context.Configuration.GetSection("Kestrel"))` by default to load Kestrel configuration.

In the following *appsettings.json* example, a default connection protocol (HTTP/1.1 and HTTP/2) is established for all of Kestrel's endpoints:

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
  }
}
```

The following configuration file example establishes a connection protocol for a specific endpoint:

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsDefaultCert": {
        "Url": "https://localhost:5001",
        "Protocols": "Http1AndHttp2"
      }
    }
  }
}
```

Protocols specified in code override values set by configuration.

Transport configuration

With the release of ASP.NET Core 2.1, Kestrel's default transport is no longer based on Libuv but instead based on managed sockets. This is a breaking change for ASP.NET Core 2.0 apps upgrading to 2.1 that call UseLibuv and depend on either of the following packages:

- Microsoft.AspNetCore.Server.Kestrel (direct package reference)
- Microsoft.AspNetCore.App

For projects that require the use of Libuv:

- Add a dependency for the Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv package to the app's project file:

  ```
  <PackageReference Include="Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv"
                    Version="{VERSION}" />
  ```

- Call UseLibuv:

  ```
  public class Program
  {
      public static void Main(string[] args)
      {
          CreateWebHostBuilder(args).Build().Run();
      }

      public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
          WebHost.CreateDefaultBuilder(args)
              .UseLibuv()
              .UseStartup<Startup>();
  }
  ```

URL prefixes

When using `UseUrls`, `--urls` command-line argument, `urls` host configuration key, or `ASPNETCORE_URLS` environment variable, the URL prefixes can be in any of the following formats.

Only HTTP URL prefixes are valid. Kestrel doesn't support HTTPS when configuring URL bindings using `UseUrls`.

- IPv4 address with port number

  ```
  http://65.55.39.10:80/
  ```

  `0.0.0.0` is a special case that binds to all IPv4 addresses.

- IPv6 address with port number

  ```
  http://[0:0:0:0:0:ffff:4137:270a]:80/
  ```

  `[::]` is the IPv6 equivalent of IPv4 `0.0.0.0`.

- Host name with port number

  ```
  http://contoso.com:80/
  http://*:80/
  ```

  Host names, `*`, and `+`, aren't special. Anything not recognized as a valid IP address or `localhost` binds to all IPv4 and IPv6 IPs. To bind different host names to different ASP.NET Core apps on the same port, use HTTP.sys or a reverse proxy server, such as IIS, Nginx, or Apache.

  > **WARNING**
  > Hosting in a reverse proxy configuration requires host filtering.

- Host `localhost` name with port number or loopback IP with port number

  ```
  http://localhost:5000/
  http://127.0.0.1:5000/
  http://[::1]:5000/
  ```

  When `localhost` is specified, Kestrel attempts to bind to both IPv4 and IPv6 loopback interfaces. If the requested port is in use by another service on either loopback interface, Kestrel fails to start. If either loopback interface is unavailable for any other reason (most commonly because IPv6 isn't supported), Kestrel logs a warning.

Host filtering

While Kestrel supports configuration based on prefixes such as `http://example.com:5000`, Kestrel largely ignores the host name. Host `localhost` is a special case used for binding to loopback addresses. Any host other than an explicit IP address binds to all public IP addresses. `Host` headers aren't validated.

As a workaround, use Host Filtering Middleware. Host Filtering Middleware is provided by the Microsoft.AspNetCore.HostFiltering package, which is included in the Microsoft.AspNetCore.App metapackage (ASP.NET Core 2.1 or 2.2). The middleware is added by CreateDefaultBuilder, which calls AddHostFiltering:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Host Filtering Middleware is disabled by default. To enable the middleware, define an `AllowedHosts` key in *appsettings.json*/*appsettings.<EnvironmentName>.json*. The value is a semicolon-delimited list of host names without port numbers:

appsettings.json:

```
{
  "AllowedHosts": "example.com;localhost"
}
```

NOTE

Forwarded Headers Middleware also has an AllowedHosts option. Forwarded Headers Middleware and Host Filtering Middleware have similar functionality for different scenarios. Setting `AllowedHosts` with Forwarded Headers Middleware is appropriate when the `Host` header isn't preserved while forwarding requests with a reverse proxy server or load balancer. Setting `AllowedHosts` with Host Filtering Middleware is appropriate when Kestrel is used as a public-facing edge server or when the `Host` header is directly forwarded.

For more information on Forwarded Headers Middleware, see Configure ASP.NET Core to work with proxy servers and load balancers.

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the web server that's included by default in ASP.NET Core project templates.

Kestrel supports the following scenarios:

- HTTPS
- Opaque upgrade used to enable WebSockets
- Unix sockets for high performance behind Nginx

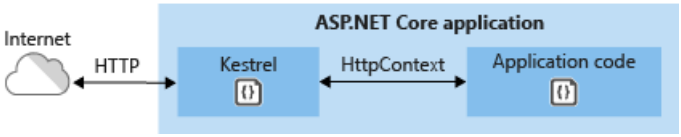
Kestrel is supported on all platforms and versions that .NET Core supports.

View or download sample code (how to download)

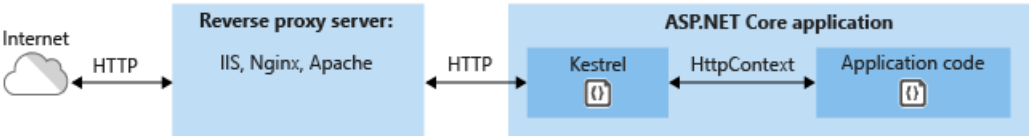
When to use Kestrel with a reverse proxy

Kestrel can be used by itself or with a *reverse proxy server*, such as Internet Information Services (IIS), Nginx, or Apache. A reverse proxy server receives HTTP requests from the network and forwards them to Kestrel.

Kestrel used as an edge (Internet-facing) web server:



Kestrel used in a reverse proxy configuration:



Either configuration, with or without a reverse proxy server, is a supported hosting configuration.

Kestrel used as an edge server without a reverse proxy server doesn't support sharing the same IP and port among multiple processes. When Kestrel is configured to listen on a port, Kestrel handles all of the traffic for that port regardless of requests' `Host` headers. A reverse proxy that can share ports has the ability to forward requests to Kestrel on a unique IP and port.

Even if a reverse proxy server isn't required, using a reverse proxy server might be a good choice.

A reverse proxy:

- Can limit the exposed public surface area of the apps that it hosts.
- Provide an additional layer of configuration and defense.
- Might integrate better with existing infrastructure.
- Simplify load balancing and secure communication (HTTPS) configuration. Only the reverse proxy server requires an X.509 certificate, and that server can communicate with the app's servers on the internal network using plain HTTP.

> **WARNING**
>
> Hosting in a reverse proxy configuration requires host filtering.

How to use Kestrel in ASP.NET Core apps

The Microsoft.AspNetCore.Server.Kestrel package is included in the Microsoft.AspNetCore.App metapackage.

ASP.NET Core project templates use Kestrel by default. In *Program.cs*, the template code calls CreateDefaultBuilder, which calls UseKestrel behind the scenes.

To provide additional configuration after calling `CreateDefaultBuilder`, call UseKestrel:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            // Set properties and call methods on serverOptions
        });
```

For more information on `CreateDefaultBuilder` and building the host, see the *Set up a host* section of ASP.NET Core Web Host.

Kestrel options

The Kestrel web server has constraint configuration options that are especially useful in Internet-facing deployments.

Set constraints on the Limits property of the KestrelServerOptions class. The `Limits` property holds an instance of the KestrelServerLimits class.

The following examples use the Microsoft.AspNetCore.Server.Kestrel.Core namespace:

```
using Microsoft.AspNetCore.Server.Kestrel.Core;
```

Kestrel options, which are configured in C# code in the following examples, can also be set using a configuration provider. For example, the File Configuration Provider can load Kestrel configuration from an *appsettings.json* or *appsettings.{Environment}.json* file:

```
{
  "Kestrel": {
    "Limits": {
      "MaxConcurrentConnections": 100,
      "MaxConcurrentUpgradedConnections": 100
    }
  }
}
```

Use **one** of the following approaches:

- Configure Kestrel in `Startup.ConfigureServices`:

  1. Inject an instance of `IConfiguration` into the `Startup` class. The following example assumes that the injected configuration is assigned to the `Configuration` property.

  2. In `Startup.ConfigureServices`, load the `Kestrel` section of configuration into Kestrel's configuration:

     ```
     using Microsoft.Extensions.Configuration

     public class Startup
     {
         public Startup(IConfiguration configuration)
         {
             Configuration = configuration;
         }

         public IConfiguration Configuration { get; }

         public void ConfigureServices(IServiceCollection services)
         {
             services.Configure<KestrelServerOptions>(
                 Configuration.GetSection("Kestrel"));
         }

         public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
         {
             ...
         }
     }
     ```

- Configure Kestrel when building the host:

In *Program.cs*, load the `Kestrel` section of configuration into Kestrel's configuration:

```
// using Microsoft.Extensions.DependencyInjection;

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureServices((context, services) =>
        {
            services.Configure<KestrelServerOptions>(
                context.Configuration.GetSection("Kestrel"));
        })
        .UseStartup<Startup>();
```

Both of the preceding approaches work with any configuration provider.

Keep-alive timeout

KeepAliveTimeout

Gets or sets the keep-alive timeout. Defaults to 2 minutes.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
        });
```

Maximum client connections

MaxConcurrentConnections MaxConcurrentUpgradedConnections

The maximum number of concurrent open TCP connections can be set for the entire app with the following code:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Limits.MaxConcurrentConnections = 100;
        });
```

There's a separate limit for connections that have been upgraded from HTTP or HTTPS to another protocol (for example, on a WebSockets request). After a connection is upgraded, it isn't counted against the `MaxConcurrentConnections` limit.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
        });
```

The maximum number of connections is unlimited (null) by default.

Maximum request body size

MaxRequestBodySize

The default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB.

The recommended approach to override the limit in an ASP.NET Core MVC app is to use the RequestSizeLimitAttribute attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

Here's an example that shows how to configure the constraint for the app on every request:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Limits.MaxRequestBodySize = 10 * 1024;
        });
```

Override the setting on a specific request in middleware:

```
app.Run(async (context) =>
{
    context.Features.Get<IHttpMaxRequestBodySizeFeature>()
        .MaxRequestBodySize = 10 * 1024;

    var minRequestRateFeature =
        context.Features.Get<IHttpMinRequestBodyDataRateFeature>();
    var minResponseRateFeature =
        context.Features.Get<IHttpMinResponseDataRateFeature>();

    if (minRequestRateFeature != null)
    {
        minRequestRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    if (minResponseRateFeature != null)
    {
        minResponseRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }
```

An exception is thrown if the app configures the limit on a request after the app has started to read the request. There's an `IsReadOnly` property that indicates if the `MaxRequestBodySize` property is in read-only state, meaning it's too late to configure the limit.

When an app is run out-of-process behind the ASP.NET Core Module, Kestrel's request body size limit is disabled because IIS already sets the limit.

Minimum request body data rate

MinRequestBodyDataRate MinResponseDataRate

Kestrel checks every second if data is arriving at the specified rate in bytes/second. If the rate drops below the minimum, the connection is timed out. The grace period is the amount of time that Kestrel gives the client to increase its send rate up to the minimum; the rate isn't checked during that time. The grace period helps avoid dropping connections that are initially sending data at a slow rate due to TCP slow-start.

The default minimum rate is 240 bytes/second with a 5 second grace period.

A minimum rate also applies to the response. The code to set the request limit and the response limit is the same except for having `RequestBody` or `Response` in the property and interface names.

Here's an example that shows how to configure the minimum data rates in *Program.cs*:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Limits.MinRequestBodyDataRate =
                new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
            serverOptions.Limits.MinResponseDataRate =
                new MinDataRate(bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
        });
```

Request headers timeout

RequestHeadersTimeout

Gets or sets the maximum amount of time the server spends receiving request headers. Defaults to 30 seconds.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
        });
```

Synchronous I/O

AllowSynchronousIO controls whether synchronous I/O is allowed for the request and response. The default value is `true`.

> **WARNING**
> A large number of blocking synchronous I/O operations can lead to thread pool starvation, which makes the app unresponsive. Only enable `AllowSynchronousIO` when using a library that doesn't support asynchronous I/O.

The following example disables synchronous I/O:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.AllowSynchronousIO = false;
        });
```

For information about other Kestrel options and limits, see:

- KestrelServerOptions
- KestrelServerLimits
- ListenOptions

Endpoint configuration

By default, ASP.NET Core binds to:

- `http://localhost:5000`
- `https://localhost:5001` (when a local development certificate is present)

Specify URLs using the:

- `ASPNETCORE_URLS` environment variable.
- `--urls` command-line argument.
- `urls` host configuration key.
- `UseUrls` extension method.

The value provided using these approaches can be one or more HTTP and HTTPS endpoints (HTTPS if a default cert is available). Configure the value as a semicolon-separated list (for example, `"Urls": "http://localhost:8000;http://localhost:8001"`).

For more information on these approaches, see Server URLs and Override configuration.

A development certificate is created:

- When the .NET Core SDK is installed.
- The dev-certs tool is used to create a certificate.

Some browsers require granting explicit permission to trust the local development certificate.

Project templates configure apps to run on HTTPS by default and include HTTPS redirection and HSTS support.

Call Listen or ListenUnixSocket methods on KestrelServerOptions to configure URL prefixes and ports for Kestrel.

`UseUrls`, the `--urls` command-line argument, `urls` host configuration key, and the `ASPNETCORE_URLS` environment variable also work but have the limitations noted later in this section (a default certificate must be available for HTTPS endpoint configuration).

`KestrelServerOptions` configuration:

ConfigureEndpointDefaults(Action<ListenOptions>)

Specifies a configuration `Action` to run for each specified endpoint. Calling `ConfigureEndpointDefaults` multiple times replaces prior `Action`s with the last `Action` specified.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.ConfigureEndpointDefaults(listenOptions =>
            {
                // Configure endpoint defaults
            });
        });
```

> **NOTE**
> Endpoints created by calling Listen **before** calling ConfigureEndpointDefaults won't have the defaults applied.

ConfigureHttpsDefaults(Action<HttpsConnectionAdapterOptions>)

Specifies a configuration `Action` to run for each HTTPS endpoint. Calling `ConfigureHttpsDefaults` multiple times replaces prior `Action`s with the last `Action` specified.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, serverOptions) =>
        {
            serverOptions.ConfigureHttpsDefaults(listenOptions =>
            {
                // certificate is an X509Certificate2
                listenOptions.ServerCertificate = certificate;
            });
        });
```

> **NOTE**
> Endpoints created by calling Listen **before** calling ConfigureHttpsDefaults won't have the defaults applied.

Configure(IConfiguration)

Creates a configuration loader for setting up Kestrel that takes an IConfiguration as input. The configuration must be scoped to the configuration section for Kestrel.

ListenOptions.UseHttps

Configure Kestrel to use HTTPS.

`ListenOptions.UseHttps` extensions:

- `UseHttps`: Configure Kestrel to use HTTPS with the default certificate. Throws an exception if no default certificate is configured.
- `UseHttps(string fileName)`
- `UseHttps(string fileName, string password)`
- `UseHttps(string fileName, string password, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(StoreName storeName, string subject)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location)`
- `UseHttps(StoreName storeName, string subject, bool allowInvalid, StoreLocation location, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(X509Certificate2 serverCertificate)`
- `UseHttps(X509Certificate2 serverCertificate, Action<HttpsConnectionAdapterOptions> configureOptions)`
- `UseHttps(Action<HttpsConnectionAdapterOptions> configureOptions)`

`ListenOptions.UseHttps` parameters:

- `filename` is the path and file name of a certificate file, relative to the directory that contains

the app's content files.

- `password` is the password required to access the X.509 certificate data.
- `configureOptions` is an `Action` to configure the `HttpsConnectionAdapterOptions`. Returns the `ListenOptions`.
- `storeName` is the certificate store from which to load the certificate.
- `subject` is the subject name for the certificate.
- `allowInvalid` indicates if invalid certificates should be considered, such as self-signed certificates.
- `location` is the store location to load the certificate from.
- `serverCertificate` is the X.509 certificate.

In production, HTTPS must be explicitly configured. At a minimum, a default certificate must be provided.

Supported configurations described next:

- No configuration
- Replace the default certificate from configuration
- Change the defaults in code

No configuration

Kestrel listens on `http://localhost:5000` and `https://localhost:5001` (if a default cert is available).

Replace the default certificate from configuration

`CreateDefaultBuilder` calls `Configure(context.Configuration.GetSection("Kestrel"))` by default to load Kestrel configuration. A default HTTPS app settings configuration schema is available for Kestrel. Configure multiple endpoints, including the URLs and the certificates to use, either from a file on disk or from a certificate store.

In the following *appsettings.json* example:

- Set **AllowInvalid** to `true` to permit the use of invalid certificates (for example, self-signed certificates).
- Any HTTPS endpoint that doesn't specify a certificate (**HttpsDefaultCert** in the example that follows) falls back to the cert defined under **Certificates** > **Default** or the development certificate.

```
{
  "Kestrel": {
    "Endpoints": {
      "Http": {
        "Url": "http://localhost:5000"
      },

      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      },

      "HttpsInlineCertStore": {
        "Url": "https://localhost:5002",
        "Certificate": {
          "Subject": "<subject; required>",
          "Store": "<certificate store; required>",
          "Location": "<location; defaults to CurrentUser>",
          "AllowInvalid": "<true or false; defaults to false>"
        }
      },

      "HttpsDefaultCert": {
        "Url": "https://localhost:5003"
      },

      "Https": {
        "Url": "https://*:5004",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "<certificate password>"
      }
    }
  }
}
```

An alternative to using **Path** and **Password** for any certificate node is to specify the certificate using certificate store fields. For example, the **Certificates** > **Default** certificate can be specified as:

```
"Default": {
  "Subject": "<subject; required>",
  "Store": "<cert store; required>",
  "Location": "<location; defaults to CurrentUser>",
  "AllowInvalid": "<true or false; defaults to false>"
}
```

Schema notes:

- Endpoints names are case-insensitive. For example, `HTTPS` and `Https` are valid.
- The `Url` parameter is required for each endpoint. The format for this parameter is the same as the top-level `Urls` configuration parameter except that it's limited to a single value.

- These endpoints replace those defined in the top-level `Urls` configuration rather than adding to them. Endpoints defined in code via `Listen` are cumulative with the endpoints defined in the configuration section.
- The `Certificate` section is optional. If the `Certificate` section isn't specified, the defaults defined in earlier scenarios are used. If no defaults are available, the server throws an exception and fails to start.
- The `Certificate` section supports both **Path–Password** and **Subject–Store** certificates.
- Any number of endpoints may be defined in this way so long as they don't cause port conflicts.
- `options.Configure(context.Configuration.GetSection("{SECTION}"))` returns a `KestrelConfigurationLoader` with an `.Endpoint(string name, listenOptions => { })` method that can be used to supplement a configured endpoint's settings:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel((context, serverOptions) =>
        {
            serverOptions.Configure(context.Configuration.GetSection("Kestrel"))
                .Endpoint("HTTPS", listenOptions =>
                {
                    listenOptions.HttpsOptions.SslProtocols = SslProtocols.Tls12;
                });
        });
```

`KestrelServerOptions.ConfigurationLoader` can be directly accessed to continue iterating on the existing loader, such as the one provided by CreateDefaultBuilder.

- The configuration section for each endpoint is available on the options in the `Endpoint` method so that custom settings may be read.
- Multiple configurations may be loaded by calling `options.Configure(context.Configuration.GetSection("{SECTION}"))` again with another section. Only the last configuration is used, unless `Load` is explicitly called on prior instances. The metapackage doesn't call `Load` so that its default configuration section may be replaced.
- `KestrelConfigurationLoader` mirrors the `Listen` family of APIs from `KestrelServerOptions` as `Endpoint` overloads, so code and config endpoints may be configured in the same place. These overloads don't use names and only consume default settings from configuration.

Change the defaults in code

`ConfigureEndpointDefaults` and `ConfigureHttpsDefaults` can be used to change default settings for `ListenOptions` and `HttpsConnectionAdapterOptions`, including overriding the default certificate specified in the prior scenario. `ConfigureEndpointDefaults` and `ConfigureHttpsDefaults` should be called before any endpoints are configured.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel((context, serverOptions) =>
        {
            serverOptions.ConfigureEndpointDefaults(listenOptions =>
            {
                // Configure endpoint defaults
            });

            serverOptions.ConfigureHttpsDefaults(listenOptions =>
            {
                listenOptions.SslProtocols = SslProtocols.Tls12;
            });
        });
```

Kestrel support for SNI

Server Name Indication (SNI) can be used to host multiple domains on the same IP address and port. For SNI to function, the client sends the host name for the secure session to the server during the TLS handshake so that the server can provide the correct certificate. The client uses the furnished certificate for encrypted communication with the server during the secure session that follows the TLS handshake.

Kestrel supports SNI via the `ServerCertificateSelector` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate.

SNI support requires:

- Running on target framework `netcoreapp2.1` or later. On `net461` or later, the callback is invoked but the `name` is always `null`. The `name` is also `null` if the client doesn't provide the host name parameter in the TLS handshake.
- All websites run on the same Kestrel instance. Kestrel doesn't support sharing an IP address and port across multiple instances without a reverse proxy.

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel((context, serverOptions) =>
        {
            serverOptions.ListenAnyIP(5005, listenOptions =>
            {
                listenOptions.UseHttps(httpsOptions =>
                {
                    var localhostCert = CertificateLoader.LoadFromStoreCert(
                        "localhost", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var exampleCert = CertificateLoader.LoadFromStoreCert(
                        "example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var subExampleCert = CertificateLoader.LoadFromStoreCert(
                        "sub.example.com", "My", StoreLocation.CurrentUser,
                        allowInvalid: true);
                    var certs = new Dictionary<string, X509Certificate2>(
                        StringComparer.OrdinalIgnoreCase);
                    certs["localhost"] = localhostCert;
                    certs["example.com"] = exampleCert;
                    certs["sub.example.com"] = subExampleCert;

                    httpsOptions.ServerCertificateSelector = (connectionContext, name) =>
                    {
                        if (name != null && certs.TryGetValue(name, out var cert))
                        {
                            return cert;
                        }

                        return exampleCert;
                    };
                });
            });
        })
        .Build();
```

Connection logging

Call UseConnectionLogging to emit Debug level logs for byte-level communication on a connection. Connection logging is helpful for troubleshooting problems in low-level communication, such as during TLS encryption and behind proxies. If `UseConnectionLogging` is placed before `UseHttps`, encrypted traffic is logged. If `UseConnectionLogging` is placed after `UseHttps`, decrypted traffic is logged.

```
webBuilder.ConfigureKestrel(serverOptions =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseConnectionLogging();
    });
});
```

Bind to a TCP socket

The Listen method binds to a TCP socket, and an options lambda permits X.509 certificate configuration:

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Listen(IPAddress.Loopback, 5000);
            serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testPassword");
            });
        });
```


```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.Listen(IPAddress.Loopback, 5000);
            serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testPassword");
            });
        });
```


The example configures HTTPS for an endpoint with ListenOptions. Use the same API to configure other Kestrel settings for specific endpoints.

On Windows, self-signed certificates can be created using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

On macOS, Linux, and Windows, certificates can be created using OpenSSL.

Bind to a Unix socket

Listen on a Unix socket with ListenUnixSocket for improved performance with Nginx, as shown in this example:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseKestrel(serverOptions =>
        {
            serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock");
            serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock", listenOptions =>
            {
                listenOptions.UseHttps("testCert.pfx", "testpassword");
            });
        });
```

- In the Nginx confiuguration file, set the `server` > `location` > `proxy_pass` entry to `http://unix:/tmp/{KESTREL SOCKET}:/;`. `{KESTREL SOCKET}` is the name of the socket provided

to ListenUnixSocket (for example, `kestrel-test.sock` in the preceding example).

- Ensure that the socket is writeable by Nginx (for example, `chmod go+w /tmp/kestrel-test.sock`).

Port 0

When the port number `0` is specified, Kestrel dynamically binds to an available port. The following example shows how to determine which port Kestrel actually bound at runtime:

```
public void Configure(IApplicationBuilder app)
{
    var serverAddressesFeature =
        app.ServerFeatures.Get<IServerAddressesFeature>();

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        context.Response.ContentType = "text/html";
        await context.Response
            .WriteAsync("<!DOCTYPE html><html lang=\"en\"><head>" +
                "<title></title></head><body><p>Hosted by Kestrel</p>");

        if (serverAddressesFeature != null)
        {
            await context.Response
                .WriteAsync("<p>Listening on the following addresses: " +
                    string.Join(", ", serverAddressesFeature.Addresses) +
                    "</p>");
        }

        await context.Response.WriteAsync("<p>Request URL: " +
            $"{context.Request.GetDisplayUrl()}<p>");
    });
}
```

When the app is run, the console window output indicates the dynamic port where the app can be reached:

```
Listening on the following addresses: http://127.0.0.1:48508
```

Limitations

Configure endpoints with the following approaches:

- UseUrls
- `--urls` command-line argument
- `urls` host configuration key
- `ASPNETCORE_URLS` environment variable

These methods are useful for making code work with servers other than Kestrel. However, be aware of the following limitations:

- HTTPS can't be used with these approaches unless a default certificate is provided in the HTTPS endpoint configuration (for example, using `KestrelServerOptions` configuration or a configuration file as shown earlier in this topic).
- When both the `Listen` and `UseUrls` approaches are used simultaneously, the `Listen` endpoints override the `UseUrls` endpoints.

IIS endpoint configuration

When using IIS, the URL bindings for IIS override bindings are set by either `Listen` or `UseUrls`. For more information, see the ASP.NET Core Module topic.

Transport configuration

With the release of ASP.NET Core 2.1, Kestrel's default transport is no longer based on Libuv but instead based on managed sockets. This is a breaking change for ASP.NET Core 2.0 apps upgrading to 2.1 that call UseLibuv and depend on either of the following packages:

- Microsoft.AspNetCore.Server.Kestrel (direct package reference)
- Microsoft.AspNetCore.App

For projects that require the use of Libuv:

- Add a dependency for the Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv package to the app's project file:

```
<PackageReference Include="Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv"
                  Version="{VERSION}" />
```

- Call UseLibuv:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseLibuv()
            .UseStartup<Startup>();
}
```

URL prefixes

When using `UseUrls`, `--urls` command-line argument, `urls` host configuration key, or `ASPNETCORE_URLS` environment variable, the URL prefixes can be in any of the following formats.

Only HTTP URL prefixes are valid. Kestrel doesn't support HTTPS when configuring URL bindings using `UseUrls`.

- IPv4 address with port number

```
http://65.55.39.10:80/
```

  `0.0.0.0` is a special case that binds to all IPv4 addresses.

- IPv6 address with port number

```
http://[0:0:0:0:0:ffff:4137:270a]:80/
```

  `[::]` is the IPv6 equivalent of IPv4 `0.0.0.0`.

- Host name with port number

```
http://contoso.com:80/
http://*:80/
```

Host names, `*`, and `+`, aren't special. Anything not recognized as a valid IP address or `localhost` binds to all IPv4 and IPv6 IPs. To bind different host names to different ASP.NET Core apps on the same port, use HTTP.sys or a reverse proxy server, such as IIS, Nginx, or Apache.

> **WARNING**
> Hosting in a reverse proxy configuration requires host filtering.

- Host `localhost` name with port number or loopback IP with port number

```
http://localhost:5000/
http://127.0.0.1:5000/
http://[::1]:5000/
```

When `localhost` is specified, Kestrel attempts to bind to both IPv4 and IPv6 loopback interfaces. If the requested port is in use by another service on either loopback interface, Kestrel fails to start. If either loopback interface is unavailable for any other reason (most commonly because IPv6 isn't supported), Kestrel logs a warning.

Host filtering

While Kestrel supports configuration based on prefixes such as `http://example.com:5000`, Kestrel largely ignores the host name. Host `localhost` is a special case used for binding to loopback addresses. Any host other than an explicit IP address binds to all public IP addresses. `Host` headers aren't validated.

As a workaround, use Host Filtering Middleware. Host Filtering Middleware is provided by the Microsoft.AspNetCore.HostFiltering package, which is included in the Microsoft.AspNetCore.App metapackage (ASP.NET Core 2.1 or 2.2). The middleware is added by CreateDefaultBuilder, which calls AddHostFiltering:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Host Filtering Middleware is disabled by default. To enable the middleware, define an `AllowedHosts` key in *appsettings.json*/*appsettings.<EnvironmentName>.json*. The value is a semicolon-delimited list of host names without port numbers:

appsettings.json:

```
{
  "AllowedHosts": "example.com;localhost"
}
```

NOTE

Forwarded Headers Middleware also has an AllowedHosts option. Forwarded Headers Middleware and Host Filtering Middleware have similar functionality for different scenarios. Setting `AllowedHosts` with Forwarded Headers Middleware is appropriate when the `Host` header isn't preserved while forwarding requests with a reverse proxy server or load balancer. Setting `AllowedHosts` with Host Filtering Middleware is appropriate when Kestrel is used as a public-facing edge server or when the `Host` header is directly forwarded.

For more information on Forwarded Headers Middleware, see Configure ASP.NET Core to work with proxy servers and load balancers.

HTTP/1.1 request draining

Opening HTTP connections is time consuming. For HTTPS, it's also resource intensive. Therefore, Kestrel tries to reuse connections per the HTTP/1.1 protocol. A request body must be fully consumed to allow the connection to be reused. The app doesn't always consume the request body, such as a `POST` requests where the server returns a redirect or 404 response. In the `POST` -redirect case:

- The client may already have sent part of the `POST` data.
- The server writes the 301 response.
- The connection can't be used for a new request until the `POST` data from the previous request body has been fully read.
- Kestrel tries to drain the request body. Draining the request body means reading and discarding the data without processing it.

The draining process makes a tradoff between allowing the connection to be reused and the time it takes to drain any remaining data:

- Draining has a timeout of five seconds, which isn't configurable.
- If all of the data specified by the `Content-Length` or `Transfer-Encoding` header hasn't been read before the timeout, the connection is closed.

Sometimes you may want to terminate the request immediately, before or after writing the response. For example, clients may have restrictive data caps, so limiting uploaded data might be a priority. In such cases to terminate a request, call HttpContext.Abort from a controller, Razor Page, or middleware.

There are caveats to calling `Abort`:

- Creating new connections can be slow and expensive.
- There's no guarantee that the client has read the response before the connection closes.
- Calling `Abort` should be rare and reserved for severe error cases, not common errors.
  - Only call `Abort` when a specific problem needs to be solved. For example, call `Abort` if malicious clients are trying to `POST` data or when there's a bug in client code that causes large or numerous requests.
  - Don't call `Abort` for common error situations, such as HTTP 404 (Not Found).

Calling HttpResponse.CompleteAsync before calling `Abort` ensures that the server has completed writing the response. However, client behavior isn't predictable and they may not read the response before the connection is aborted.

This process is different for HTTP/2 because the protocol supports aborting individual request streams without closing the connection. The five second drain timeout doesn't apply. If there's any unread request body data after completing a response, then the server sends an HTTP/2 RST frame. Additional request body data frames are ignored.

If possible, it's better for clients to utilize the Expect: 100-continue request header and wait for the server to respond before starting to send the request body. That gives the client an opportunity to examine the response and abort before sending unneeded data.

Additional resources

- When using UNIX sockets on Linux, the socket is not automatically deleted on app shut down. For more information, see this GitHub issue.
- Troubleshoot and debug ASP.NET Core projects
- Enforce HTTPS in ASP.NET Core
- Configure ASP.NET Core to work with proxy servers and load balancers
- RFC 7230: Message Syntax and Routing (Section 5.4: Host)

HTTP.sys web server implementation in ASP.NET Core

9/22/2020 • 37 minutes to read • Edit Online

By Tom Dykstra and Chris Ross

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. HTTP.sys is an alternative to Kestrel server and offers some features that Kestrel doesn't provide.

> **IMPORTANT**
> HTTP.sys isn't compatible with the ASP.NET Core Module and can't be used with IIS or IIS Express.

HTTP.sys supports the following features:

- Windows Authentication
- Port sharing
- HTTPS with SNI
- HTTP/2 over TLS (Windows 10 or later)
- Direct file transmission
- Response caching
- WebSockets (Windows 8 or later)

Supported Windows versions:

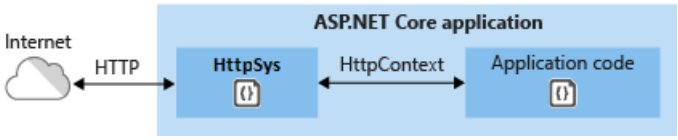
- Windows 7 or later
- Windows Server 2008 R2 or later

View or download sample code (how to download)

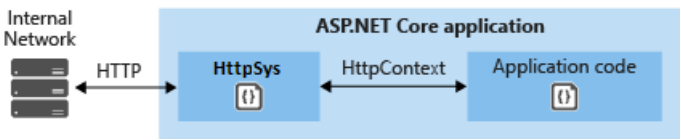
When to use HTTP.sys

HTTP.sys is useful for deployments where:

- There's a need to expose the server directly to the Internet without using IIS.



- An internal deployment requires a feature not available in Kestrel, such as Windows Authentication.



HTTP.sys is mature technology that protects against many types of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of HTTP.sys.

HTTP/2 support

HTTP/2 is enabled for ASP.NET Core apps if the following base requirements are met:

- Windows Server 2016/Windows 10 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

HTTP/2 is enabled by default. If an HTTP/2 connection isn't established, the connection falls back to HTTP/1.1. In a future release of Windows, HTTP/2 configuration flags will be available, including the ability to disable HTTP/2 with HTTP.sys.

Kernel mode authentication with Kerberos

HTTP.sys delegates to kernel mode authentication with the Kerberos authentication protocol. User mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

How to use HTTP.sys

Configure the ASP.NET Core app to use HTTP.sys

Call the UseHttpSys extension method when building the host, specifying any required HttpSysOptions. The following example sets options to their default values:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseHttpSys(options =>
            {
                options.AllowSynchronousIO = false;
                options.Authentication.Schemes = AuthenticationSchemes.None;
                options.Authentication.AllowAnonymous = true;
                options.MaxConnections = null;
                options.MaxRequestBodySize = 30000000;
                options.UrlPrefixes.Add("http://localhost:5005");
            });
            webBuilder.UseStartup<Startup>();
        });
```

Additional HTTP.sys configuration is handled through registry settings.

HTTP.sys options

PROPERTY	DESCRIPTION	DEFAULT
AllowSynchronousIO	Control whether synchronous input/output is allowed for the `HttpContext.Request.Body` and `HttpContext.Response.Body`.	`false`
Authentication.AllowAnonymous	Allow anonymous requests.	`true`

PROPERTY	DESCRIPTION	DEFAULT
Authentication.Schemes	Specify the allowed authentication schemes. May be modified at any time prior to disposing the listener. Values are provided by the AuthenticationSchemes enum: `Basic`, `Kerberos`, `Negotiate`, `None`, and `NTLM`.	`None`
EnableResponseCaching	Attempt kernel-mode caching for responses with eligible headers. The response may not include `Set-Cookie`, `Vary`, or `Pragma` headers. It must include a `Cache-Control` header that's `public` and either a `shared-max-age` or `max-age` value, or an `Expires` header.	`true`
MaxAccepts	The maximum number of concurrent accepts.	5 × Environment.ProcessorCount
MaxConnections	The maximum number of concurrent connections to accept. Use `-1` for infinite. Use `null` to use the registry's machine-wide setting.	`null` (machine-wide setting)
MaxRequestBodySize	See the MaxRequestBodySize section.	30000000 bytes (~28.6 MB)
RequestQueueLimit	The maximum number of requests that can be queued.	1000
`RequestQueueMode`	This indicates whether the server is responsible for creating and configuring the request queue, or if it should attach to an existing queue. Most existing configuration options do not apply when attaching to an existing queue.	`RequestQueueMode.Create`
`RequestQueueName`	The name of the HTTP.sys request queue.	`null` (Anonymous queue)
ThrowWriteExceptions	Indicate if response body writes that fail due to client disconnects should throw exceptions or complete normally.	`false` (complete normally)

PROPERTY	DESCRIPTION	DEFAULT
Timeouts	Expose the HTTP.sys TimeoutManager configuration, which may also be configured in the registry. Follow the API links to learn more about each setting, including default values: • TimeoutManager.DrainEntityBody: Time allowed for the HTTP Server API to drain the entity body on a Keep-Alive connection. • TimeoutManager.EntityBody: Time allowed for the request entity body to arrive. • TimeoutManager.HeaderWait: Time allowed for the HTTP Server API to parse the request header. • TimeoutManager.IdleConnection: Time allowed for an idle connection. • TimeoutManager.MinSendBytesPerSecond: The minimum send rate for the response. • TimeoutManager.RequestQueue: Time allowed for the request to remain in the request queue before the app picks it up.	
UrlPrefixes	Specify the UrlPrefixCollection to register with HTTP.sys. The most useful is UrlPrefixCollection.Add, which is used to add a prefix to the collection. These may be modified at any time prior to disposing the listener.	

MaxRequestBodySize

The maximum allowed size of any request body in bytes. When set to `null`, the maximum request body size is unlimited. This limit has no effect on upgraded connections, which are always unlimited.

The recommended method to override the limit in an ASP.NET Core MVC app for a single `IActionResult` is to use the RequestSizeLimitAttribute attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

An exception is thrown if the app attempts to configure the limit on a request after the app has started reading the request. An `IsReadOnly` property can be used to indicate if the `MaxRequestBodySize` property is in a read-only state, meaning it's too late to configure the limit.

If the app should override MaxRequestBodySize per-request, use the IHttpMaxRequestBodySizeFeature:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    ILogger<Startup> logger, IServer server)
{
    app.Use(async (context, next) =>
    {
        context.Features.Get<IHttpMaxRequestBodySizeFeature>()
            .MaxRequestBodySize = 10 * 1024;

        var serverAddressesFeature =
            app.ServerFeatures.Get<IServerAddressesFeature>();
        var addresses = string.Join(", ", serverAddressesFeature?.Addresses);

        logger.LogInformation("Addresses: {Addresses}", addresses);

        await next.Invoke();
    });

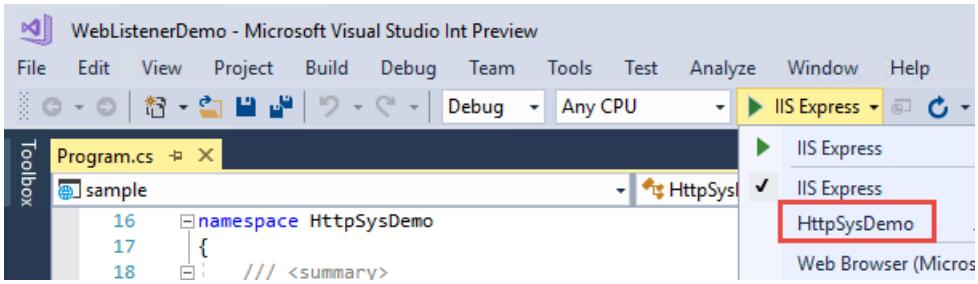
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

If using Visual Studio, make sure the app isn't configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console app, manually change the selected profile, as shown in the following screen shot:



Configure Windows Server

1. Determine the ports to open for the app and use Windows Firewall or the New-NetFirewallRule PowerShell cmdlet to open firewall ports to allow traffic to reach HTTP.sys. In the following commands and app configuration, port 443 is used.

2. When deploying to an Azure VM, open the ports in the Network Security Group. In the following commands and app configuration, port 443 is used.

3. Obtain and install X.509 certificates, if required.

   On Windows, create self-signed certificates using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

   Install either self-signed or CA-signed certificates in the server's **Local Machine** > **Personal** store.

4. If the app is a framework-dependent deployment, install .NET Core, .NET Framework, or both (if the app is a .NET Core app targeting the .NET Framework).

   - **.NET Core**: If the app requires .NET Core, obtain and run the **.NET Core Runtime** installer from .NET Core Downloads. Don't install the full SDK on the server.
   - **.NET Framework**: If the app requires .NET Framework, see the .NET Framework installation guide. Install the required .NET Framework. The installer for the latest .NET Framework is available from the .NET Core Downloads page.

   If the app is a self-contained deployment, the app includes the runtime in its deployment. No framework installation is required on the server.

5. Configure URLs and ports in the app.

   By default, ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, options include:

   - UseUrls
   - `urls` command-line argument
   - `ASPNETCORE_URLS` environment variable
   - UrlPrefixes

   The following code example shows how to use UrlPrefixes with the server's local IP address `10.0.0.4` on port 443:

   ```
   public static IHostBuilder CreateHostBuilder(string[] args) =>
       Host.CreateDefaultBuilder(args)
           .ConfigureWebHostDefaults(webBuilder =>
           {
               webBuilder.UseHttpSys(options =>
               {
                   options.UrlPrefixes.Add("https://10.0.0.4:443");
               });
               webBuilder.UseStartup<Startup>();
           });
   ```

   An advantage of `UrlPrefixes` is that an error message is generated immediately for improperly formatted prefixes.

   The settings in `UrlPrefixes` override `UseUrls` / `urls` / `ASPNETCORE_URLS` settings. Therefore, an advantage of `UseUrls`, `urls`, and the `ASPNETCORE_URLS` environment variable is that it's easier to switch between Kestrel and HTTP.sys.

   HTTP.sys uses the HTTP Server API UrlPrefix string formats.

   > **WARNING**
   > Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings create app security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names or IP addresses rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) isn't a security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 7230: Section 5.4: Host.

6. Preregister URL prefixes on the server.

   The built-in tool for configuring HTTP.sys is *netsh.exe*. *netsh.exe* is used to reserve URL prefixes and assign X.509 certificates. The tool requires administrator privileges.

Use the *netsh.exe* tool to register URLs for the app:

```
netsh http add urlacl url=<URL> user=<USER>
```

- `<URL>`: The fully qualified Uniform Resource Locator (URL). Don't use a wildcard binding. Use a valid hostname or local IP address. *The URL must include a trailing slash.*
- `<USER>`: Specifies the user or user-group name.

In the following example, the local IP address of the server is `10.0.0.4`:

```
netsh http add urlacl url=https://10.0.0.4:443/ user=Users
```

When a URL is registered, the tool responds with `URL reservation successfully added`.

To delete a registered URL, use the `delete urlacl` command:

```
netsh http delete urlacl url=<URL>
```

7. Register X.509 certificates on the server.

   Use the *netsh.exe* tool to register certificates for the app:

   ```
   netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="{<GUID>}"
   ```

   - `<IP>`: Specifies the local IP address for the binding. Don't use a wildcard binding. Use a valid IP address.
   - `<PORT>`: Specifies the port for the binding.
   - `<THUMBPRINT>`: The X.509 certificate thumbprint.
   - `<GUID>`: A developer-generated GUID to represent the app for informational purposes.

   For reference purposes, store the GUID in the app as a package tag:

   - In Visual Studio:
     - Open the app's project properties by right-clicking on the app in **Solution Explorer** and selecting **Properties**.
     - Select the **Package** tab.
     - Enter the GUID that you created in the **Tags** field.
   - When not using Visual Studio:
     - Open the app's project file.
     - Add a `<PackageTags>` property to a new or existing `<PropertyGroup>` with the GUID that you created:

       ```
       <PropertyGroup>
         <PackageTags>9412ee86-c21b-4eb8-bd89-f650fbf44931</PackageTags>
       </PropertyGroup>
       ```

   In the following example:

   - The local IP address of the server is `10.0.0.4`.
   - An online random GUID generator provides the `appid` value.

```
netsh http add sslcert
    ipport=10.0.0.4:443
    certhash=b66ee04419d4ee37464ab8785ff02449980eae10
    appid="{9412ee86-c21b-4eb8-bd89-f650fbf44931}"
```

When a certificate is registered, the tool responds with `SSL Certificate successfully added`.

To delete a certificate registration, use the `delete sslcert` command:

```
netsh http delete sslcert ipport=<IP>:<PORT>
```

Reference documentation for *netsh.exe*:

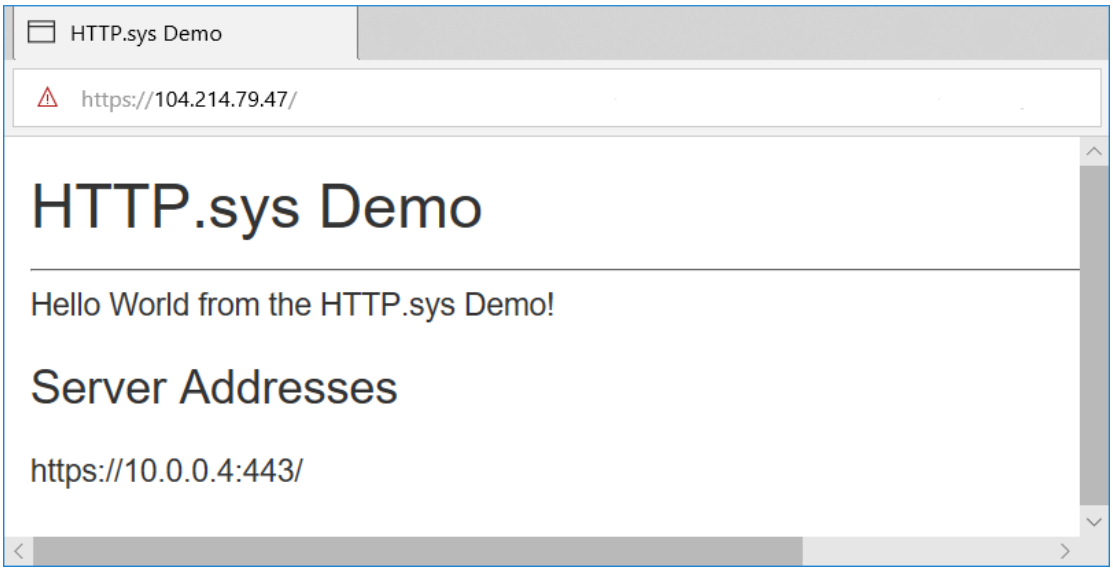
- Netsh Commands for Hypertext Transfer Protocol (HTTP)
- UrlPrefix Strings

8. Run the app.

   Administrator privileges aren't required to run the app when binding to localhost using HTTP (not HTTPS) with a port number greater than 1024. For other configurations (for example, using a local IP address or binding to port 443), run the app with administrator privileges.

   The app responds at the server's public IP address. In this example, the server is reached from the Internet at its public IP address of `104.214.79.47`.

   A development certificate is used in this example. The page loads securely after bypassing the browser's untrusted certificate warning.



Proxy server and load balancer scenarios

For apps hosted by HTTP.sys that interact with requests from the Internet or a corporate network, additional configuration might be required when hosting behind proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Additional resources

- Enable Windows Authentication with HTTP.sys
- HTTP Server API
- aspnet/HttpSysServer GitHub repository (source code)

- The host
- Troubleshoot and debug ASP.NET Core projects

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. HTTP.sys is an alternative to Kestrel server and offers some features that Kestrel doesn't provide.

> **IMPORTANT**
> HTTP.sys isn't compatible with the ASP.NET Core Module and can't be used with IIS or IIS Express.

HTTP.sys supports the following features:

- Windows Authentication
- Port sharing
- HTTPS with SNI
- HTTP/2 over TLS (Windows 10 or later)
- Direct file transmission
- Response caching
- WebSockets (Windows 8 or later)

Supported Windows versions:

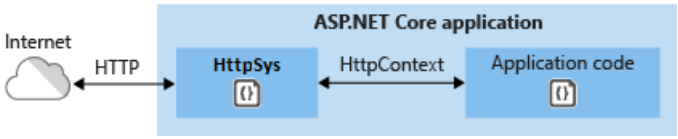
- Windows 7 or later
- Windows Server 2008 R2 or later

View or download sample code (how to download)

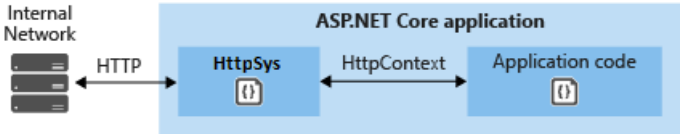
When to use HTTP.sys

HTTP.sys is useful for deployments where:

- There's a need to expose the server directly to the Internet without using IIS.



- An internal deployment requires a feature not available in Kestrel, such as Windows Authentication.



HTTP.sys is mature technology that protects against many types of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of HTTP.sys.

HTTP/2 support

HTTP/2 is enabled for ASP.NET Core apps if the following base requirements are met:

- Windows Server 2016/Windows 10 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

HTTP/2 is enabled by default. If an HTTP/2 connection isn't established, the connection falls back to HTTP/1.1. In a future release of Windows, HTTP/2 configuration flags will be available, including the ability to disable HTTP/2 with HTTP.sys.

Kernel mode authentication with Kerberos

HTTP.sys delegates to kernel mode authentication with the Kerberos authentication protocol. User mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

How to use HTTP.sys

Configure the ASP.NET Core app to use HTTP.sys

Call the UseHttpSys extension method when building the host, specifying any required HttpSysOptions. The following example sets options to their default values:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseHttpSys(options =>
            {
                options.AllowSynchronousIO = false;
                options.Authentication.Schemes = AuthenticationSchemes.None;
                options.Authentication.AllowAnonymous = true;
                options.MaxConnections = null;
                options.MaxRequestBodySize = 30000000;
                options.UrlPrefixes.Add("http://localhost:5005");
            });
            webBuilder.UseStartup<Startup>();
        });
```

Additional HTTP.sys configuration is handled through registry settings.

HTTP.sys options

PROPERTY	DESCRIPTION	DEFAULT
AllowSynchronousIO	Control whether synchronous input/output is allowed for the `HttpContext.Request.Body` and `HttpContext.Response.Body`.	`false`
Authentication.AllowAnonymous	Allow anonymous requests.	`true`
Authentication.Schemes	Specify the allowed authentication schemes. May be modified at any time prior to disposing the listener. Values are provided by the AuthenticationSchemes enum: `Basic`, `Kerberos`, `Negotiate`, `None`, and `NTLM`.	`None`

PROPERTY	DESCRIPTION	DEFAULT
EnableResponseCaching	Attempt kernel-mode caching for responses with eligible headers. The response may not include `Set-Cookie`, `Vary`, or `Pragma` headers. It must include a `Cache-Control` header that's `public` and either a `shared-max-age` or `max-age` value, or an `Expires` header.	`true`
MaxAccepts	The maximum number of concurrent accepts.	5 × Environment.ProcessorCount
MaxConnections	The maximum number of concurrent connections to accept. Use `-1` for infinite. Use `null` to use the registry's machine-wide setting.	`null` (machine-wide setting)
MaxRequestBodySize	See the MaxRequestBodySize section.	30000000 bytes (~28.6 MB)
RequestQueueLimit	The maximum number of requests that can be queued.	1000
ThrowWriteExceptions	Indicate if response body writes that fail due to client disconnects should throw exceptions or complete normally.	`false` (complete normally)

PROPERTY	DESCRIPTION	DEFAULT
Timeouts	Expose the HTTP.sys TimeoutManager configuration, which may also be configured in the registry. Follow the API links to learn more about each setting, including default values: • TimeoutManager.DrainEntityBody: Time allowed for the HTTP Server API to drain the entity body on a Keep-Alive connection. • TimeoutManager.EntityBody: Time allowed for the request entity body to arrive. • TimeoutManager.HeaderWait: Time allowed for the HTTP Server API to parse the request header. • TimeoutManager.IdleConnection: Time allowed for an idle connection. • TimeoutManager.MinSendBytesPerSecond: The minimum send rate for the response. • TimeoutManager.RequestQueue: Time allowed for the request to remain in the request queue before the app picks it up.	
UrlPrefixes	Specify the UrlPrefixCollection to register with HTTP.sys. The most useful is UrlPrefixCollection.Add, which is used to add a prefix to the collection. These may be modified at any time prior to disposing the listener.	

MaxRequestBodySize

The maximum allowed size of any request body in bytes. When set to `null`, the maximum request body size is unlimited. This limit has no effect on upgraded connections, which are always unlimited.

The recommended method to override the limit in an ASP.NET Core MVC app for a single `IActionResult` is to use the RequestSizeLimitAttribute attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

An exception is thrown if the app attempts to configure the limit on a request after the app has started reading the request. An `IsReadOnly` property can be used to indicate if the `MaxRequestBodySize` property is in a read-only state, meaning it's too late to configure the limit.

If the app should override MaxRequestBodySize per-request, use the IHttpMaxRequestBodySizeFeature:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    ILogger<Startup> logger, IServer server)
{
    app.Use(async (context, next) =>
    {
        context.Features.Get<IHttpMaxRequestBodySizeFeature>()
            .MaxRequestBodySize = 10 * 1024;

        var serverAddressesFeature =
            app.ServerFeatures.Get<IServerAddressesFeature>();
        var addresses = string.Join(", ", serverAddressesFeature?.Addresses);

        logger.LogInformation("Addresses: {Addresses}", addresses);

        await next.Invoke();
    });

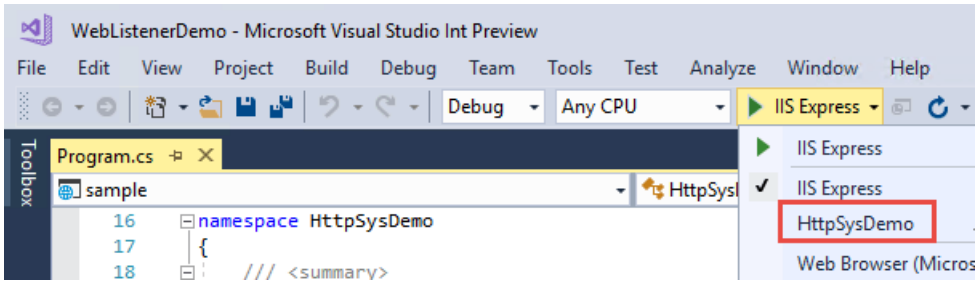
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

If using Visual Studio, make sure the app isn't configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console app, manually change the selected profile, as shown in the following screen shot:



Configure Windows Server

1. Determine the ports to open for the app and use Windows Firewall or the New-NetFirewallRule PowerShell cmdlet to open firewall ports to allow traffic to reach HTTP.sys. In the following commands and app configuration, port 443 is used.

2. When deploying to an Azure VM, open the ports in the Network Security Group. In the following commands and app configuration, port 443 is used.

3. Obtain and install X.509 certificates, if required.

   On Windows, create self-signed certificates using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

   Install either self-signed or CA-signed certificates in the server's **Local Machine** > **Personal** store.

4. If the app is a framework-dependent deployment, install .NET Core, .NET Framework, or both (if the app is a .NET Core app targeting the .NET Framework).

   - **.NET Core**: If the app requires .NET Core, obtain and run the **.NET Core Runtime** installer from .NET Core Downloads. Don't install the full SDK on the server.
   - **.NET Framework**: If the app requires .NET Framework, see the .NET Framework installation guide. Install the required .NET Framework. The installer for the latest .NET Framework is available from the .NET Core Downloads page.

   If the app is a self-contained deployment, the app includes the runtime in its deployment. No framework installation is required on the server.

5. Configure URLs and ports in the app.

   By default, ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, options include:

   - UseUrls
   - `urls` command-line argument
   - `ASPNETCORE_URLS` environment variable
   - UrlPrefixes

   The following code example shows how to use UrlPrefixes with the server's local IP address `10.0.0.4` on port 443:

   ```
   public static IHostBuilder CreateHostBuilder(string[] args) =>
       Host.CreateDefaultBuilder(args)
           .ConfigureWebHostDefaults(webBuilder =>
           {
               webBuilder.UseHttpSys(options =>
               {
                   options.UrlPrefixes.Add("https://10.0.0.4:443");
               });
               webBuilder.UseStartup<Startup>();
           });
   ```

   An advantage of `UrlPrefixes` is that an error message is generated immediately for improperly formatted prefixes.

   The settings in `UrlPrefixes` override `UseUrls` / `urls` / `ASPNETCORE_URLS` settings. Therefore, an advantage of `UseUrls`, `urls`, and the `ASPNETCORE_URLS` environment variable is that it's easier to switch between Kestrel and HTTP.sys.

   HTTP.sys uses the HTTP Server API UrlPrefix string formats.

   > **WARNING**
   > Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings create app security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names or IP addresses rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) isn't a security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 7230: Section 5.4: Host.

6. Preregister URL prefixes on the server.

   The built-in tool for configuring HTTP.sys is *netsh.exe*. *netsh.exe* is used to reserve URL prefixes and assign X.509 certificates. The tool requires administrator privileges.

Use the *netsh.exe* tool to register URLs for the app:

```
netsh http add urlacl url=<URL> user=<USER>
```

- `<URL>`: The fully qualified Uniform Resource Locator (URL). Don't use a wildcard binding. Use a valid hostname or local IP address. *The URL must include a trailing slash.*
- `<USER>`: Specifies the user or user-group name.

In the following example, the local IP address of the server is `10.0.0.4`:

```
netsh http add urlacl url=https://10.0.0.4:443/ user=Users
```

When a URL is registered, the tool responds with `URL reservation successfully added`.

To delete a registered URL, use the `delete urlacl` command:

```
netsh http delete urlacl url=<URL>
```

7. Register X.509 certificates on the server.

   Use the *netsh.exe* tool to register certificates for the app:

   ```
   netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="{<GUID>}"
   ```

   - `<IP>`: Specifies the local IP address for the binding. Don't use a wildcard binding. Use a valid IP address.
   - `<PORT>`: Specifies the port for the binding.
   - `<THUMBPRINT>`: The X.509 certificate thumbprint.
   - `<GUID>`: A developer-generated GUID to represent the app for informational purposes.

   For reference purposes, store the GUID in the app as a package tag:

   - In Visual Studio:
     - Open the app's project properties by right-clicking on the app in **Solution Explorer** and selecting **Properties**.
     - Select the **Package** tab.
     - Enter the GUID that you created in the **Tags** field.
   - When not using Visual Studio:
     - Open the app's project file.
     - Add a `<PackageTags>` property to a new or existing `<PropertyGroup>` with the GUID that you created:

       ```
       <PropertyGroup>
         <PackageTags>9412ee86-c21b-4eb8-bd89-f650fbf44931</PackageTags>
       </PropertyGroup>
       ```

   In the following example:

   - The local IP address of the server is `10.0.0.4`.
   - An online random GUID generator provides the `appid` value.

```
netsh http add sslcert
    ipport=10.0.0.4:443
    certhash=b66ee04419d4ee37464ab8785ff02449980eae10
    appid="{9412ee86-c21b-4eb8-bd89-f650fbf44931}"
```

When a certificate is registered, the tool responds with `SSL Certificate successfully added`.

To delete a certificate registration, use the `delete sslcert` command:

```
netsh http delete sslcert ipport=<IP>:<PORT>
```

Reference documentation for *netsh.exe*:

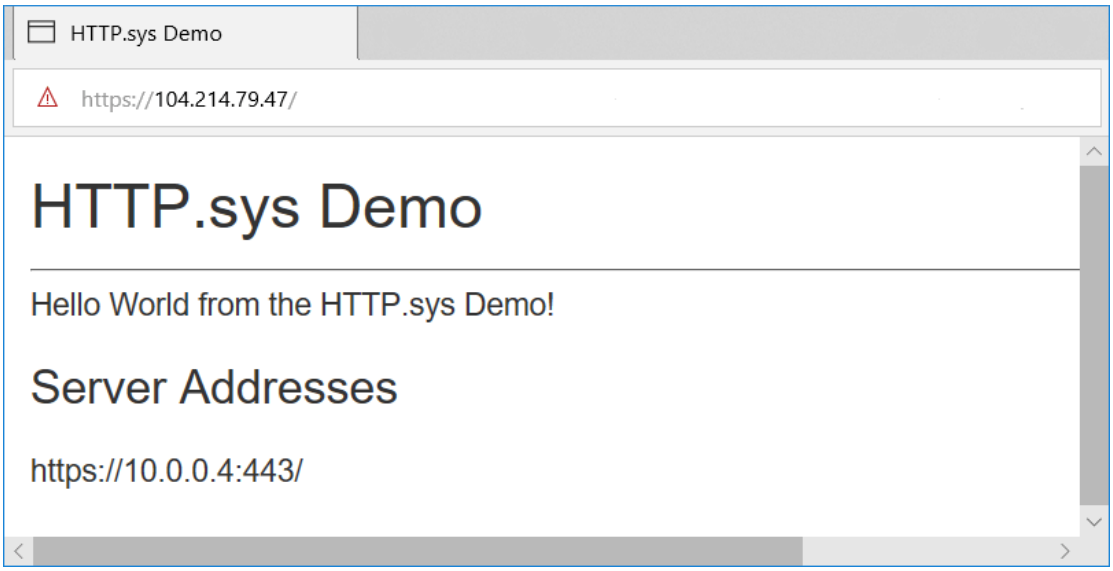
- Netsh Commands for Hypertext Transfer Protocol (HTTP)
- UrlPrefix Strings

8. Run the app.

   Administrator privileges aren't required to run the app when binding to localhost using HTTP (not HTTPS) with a port number greater than 1024. For other configurations (for example, using a local IP address or binding to port 443), run the app with administrator privileges.

   The app responds at the server's public IP address. In this example, the server is reached from the Internet at its public IP address of `104.214.79.47`.

   A development certificate is used in this example. The page loads securely after bypassing the browser's untrusted certificate warning.



Proxy server and load balancer scenarios

For apps hosted by HTTP.sys that interact with requests from the Internet or a corporate network, additional configuration might be required when hosting behind proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Additional resources

- Enable Windows Authentication with HTTP.sys
- HTTP Server API
- aspnet/HttpSysServer GitHub repository (source code)

- The host
- Troubleshoot and debug ASP.NET Core projects

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. HTTP.sys is an alternative to Kestrel server and offers some features that Kestrel doesn't provide.

> **IMPORTANT**
> HTTP.sys isn't compatible with the ASP.NET Core Module and can't be used with IIS or IIS Express.

HTTP.sys supports the following features:

- Windows Authentication
- Port sharing
- HTTPS with SNI
- HTTP/2 over TLS (Windows 10 or later)
- Direct file transmission
- Response caching
- WebSockets (Windows 8 or later)

Supported Windows versions:

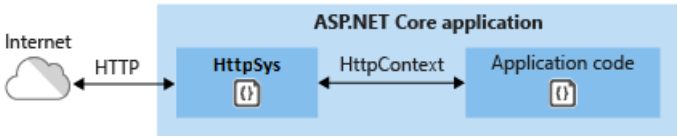
- Windows 7 or later
- Windows Server 2008 R2 or later

View or download sample code (how to download)

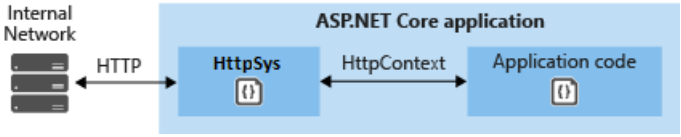
When to use HTTP.sys

HTTP.sys is useful for deployments where:

- There's a need to expose the server directly to the Internet without using IIS.



- An internal deployment requires a feature not available in Kestrel, such as Windows Authentication.



HTTP.sys is mature technology that protects against many types of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of HTTP.sys.

HTTP/2 support

HTTP/2 is enabled for ASP.NET Core apps if the following base requirements are met:

- Windows Server 2016/Windows 10 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

HTTP/2 is enabled by default. If an HTTP/2 connection isn't established, the connection falls back to HTTP/1.1. In a future release of Windows, HTTP/2 configuration flags will be available, including the ability to disable HTTP/2 with HTTP.sys.

Kernel mode authentication with Kerberos

HTTP.sys delegates to kernel mode authentication with the Kerberos authentication protocol. User mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

How to use HTTP.sys

Configure the ASP.NET Core app to use HTTP.sys

A package reference in the project file isn't required when using the Microsoft.AspNetCore.App metapackage (nuget.org). When not using the `Microsoft.AspNetCore.App` metapackage, add a package reference to Microsoft.AspNetCore.Server.HttpSys.

Call the UseHttpSys extension method when building the host, specifying any required HttpSysOptions. The following example sets options to their default values:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseHttpSys(options =>
        {
            options.AllowSynchronousIO = true;
            options.Authentication.Schemes = AuthenticationSchemes.None;
            options.Authentication.AllowAnonymous = true;
            options.MaxConnections = null;
            options.MaxRequestBodySize = 30000000;
            options.UrlPrefixes.Add("http://localhost:5000");
        });
```

Additional HTTP.sys configuration is handled through registry settings.

HTTP.sys options

PROPERTY	DESCRIPTION	DEFAULT
AllowSynchronousIO	Control whether synchronous input/output is allowed for the `HttpContext.Request.Body` and `HttpContext.Response.Body`.	`true`
Authentication.AllowAnonymous	Allow anonymous requests.	`true`
Authentication.Schemes	Specify the allowed authentication schemes. May be modified at any time prior to disposing the listener. Values are provided by the AuthenticationSchemes enum: `Basic`, `Kerberos`, `Negotiate`, `None`, and `NTLM`.	`None`

PROPERTY	DESCRIPTION	DEFAULT
EnableResponseCaching	Attempt kernel-mode caching for responses with eligible headers. The response may not include `Set-Cookie`, `Vary`, or `Pragma` headers. It must include a `Cache-Control` header that's `public` and either a `shared-max-age` or `max-age` value, or an `Expires` header.	`true`
MaxAccepts	The maximum number of concurrent accepts.	5 × Environment.ProcessorCount
MaxConnections	The maximum number of concurrent connections to accept. Use `-1` for infinite. Use `null` to use the registry's machine-wide setting.	`null` (machine-wide setting)
MaxRequestBodySize	See the MaxRequestBodySize section.	30000000 bytes (~28.6 MB)
RequestQueueLimit	The maximum number of requests that can be queued.	1000
ThrowWriteExceptions	Indicate if response body writes that fail due to client disconnects should throw exceptions or complete normally.	`false` (complete normally)

PROPERTY	DESCRIPTION	DEFAULT
Timeouts	Expose the HTTP.sys TimeoutManager configuration, which may also be configured in the registry. Follow the API links to learn more about each setting, including default values: • TimeoutManager.DrainEntityBody: Time allowed for the HTTP Server API to drain the entity body on a Keep-Alive connection. • TimeoutManager.EntityBody: Time allowed for the request entity body to arrive. • TimeoutManager.HeaderWait: Time allowed for the HTTP Server API to parse the request header. • TimeoutManager.IdleConnection: Time allowed for an idle connection. • TimeoutManager.MinSendBytesPerSecond: The minimum send rate for the response. • TimeoutManager.RequestQueue: Time allowed for the request to remain in the request queue before the app picks it up.	
UrlPrefixes	Specify the UrlPrefixCollection to register with HTTP.sys. The most useful is UrlPrefixCollection.Add, which is used to add a prefix to the collection. These may be modified at any time prior to disposing the listener.	

MaxRequestBodySize

The maximum allowed size of any request body in bytes. When set to `null`, the maximum request body size is unlimited. This limit has no effect on upgraded connections, which are always unlimited.

The recommended method to override the limit in an ASP.NET Core MVC app for a single `IActionResult` is to use the RequestSizeLimitAttribute attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

An exception is thrown if the app attempts to configure the limit on a request after the app has started reading the request. An `IsReadOnly` property can be used to indicate if the `MaxRequestBodySize` property is in a read-only state, meaning it's too late to configure the limit.

If the app should override MaxRequestBodySize per-request, use the IHttpMaxRequestBodySizeFeature:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILogger<Startup> logger, IServer server)
{
    app.Use(async (context, next) =>
    {
        context.Features.Get<IHttpMaxRequestBodySizeFeature>()
            .MaxRequestBodySize = 10 * 1024;

        var serverAddressesFeature =
            app.ServerFeatures.Get<IServerAddressesFeature>();
        var addresses = string.Join(", ", serverAddressesFeature?.Addresses);

        logger.LogInformation("Addresses: {Addresses}", addresses);

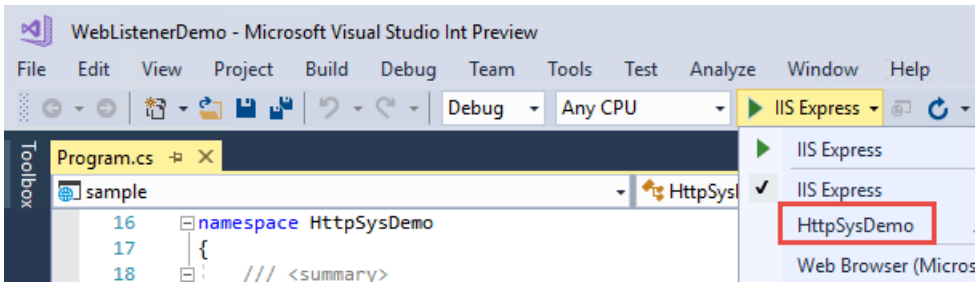
        await next.Invoke();
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    // Enable HTTPS Redirection Middleware when hosting the app securely.
    //app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseMvc();
}
```

If using Visual Studio, make sure the app isn't configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console app, manually change the selected profile, as shown in the following screen shot:



Configure Windows Server

1. Determine the ports to open for the app and use Windows Firewall or the New-NetFirewallRule PowerShell cmdlet to open firewall ports to allow traffic to reach HTTP.sys. In the following commands and app configuration, port 443 is used.

2. When deploying to an Azure VM, open the ports in the Network Security Group. In the following commands and app configuration, port 443 is used.

3. Obtain and install X.509 certificates, if required.

   On Windows, create self-signed certificates using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

   Install either self-signed or CA-signed certificates in the server's **Local Machine** > **Personal** store.

4. If the app is a framework-dependent deployment, install .NET Core, .NET Framework, or both (if the app is a .NET Core app targeting the .NET Framework).

   - **.NET Core**: If the app requires .NET Core, obtain and run the **.NET Core Runtime** installer from .NET Core Downloads. Don't install the full SDK on the server.
   - **.NET Framework**: If the app requires .NET Framework, see the .NET Framework installation guide. Install the required .NET Framework. The installer for the latest .NET Framework is available from the .NET Core Downloads page.

   If the app is a self-contained deployment, the app includes the runtime in its deployment. No framework installation is required on the server.

5. Configure URLs and ports in the app.

   By default, ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, options include:

   - UseUrls
   - `urls` command-line argument
   - `ASPNETCORE_URLS` environment variable
   - UrlPrefixes

   The following code example shows how to use UrlPrefixes with the server's local IP address `10.0.0.4` on port 443:

   ```
   public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
       WebHost.CreateDefaultBuilder(args)
           .UseStartup<Startup>()
           .UseHttpSys(options =>
           {
               options.UrlPrefixes.Add("https://10.0.0.4:443");
           });
   ```

   An advantage of `UrlPrefixes` is that an error message is generated immediately for improperly formatted prefixes.

   The settings in `UrlPrefixes` override `UseUrls` / `urls` / `ASPNETCORE_URLS` settings. Therefore, an advantage of `UseUrls`, `urls`, and the `ASPNETCORE_URLS` environment variable is that it's easier to switch between Kestrel and HTTP.sys.

   HTTP.sys uses the HTTP Server API UrlPrefix string formats.

   > **WARNING**
   > Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings create app security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names or IP addresses rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) isn't a security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 7230: Section 5.4: Host.

6. Preregister URL prefixes on the server.

   The built-in tool for configuring HTTP.sys is *netsh.exe*. *netsh.exe* is used to reserve URL prefixes and assign X.509 certificates. The tool requires administrator privileges.

   Use the *netsh.exe* tool to register URLs for the app:

```
netsh http add urlacl url=<URL> user=<USER>
```

- `<URL>`: The fully qualified Uniform Resource Locator (URL). Don't use a wildcard binding. Use a valid hostname or local IP address. *The URL must include a trailing slash.*
- `<USER>`: Specifies the user or user-group name.

In the following example, the local IP address of the server is `10.0.0.4`:

```
netsh http add urlacl url=https://10.0.0.4:443/ user=Users
```

When a URL is registered, the tool responds with `URL reservation successfully added`.

To delete a registered URL, use the `delete urlacl` command:

```
netsh http delete urlacl url=<URL>
```

7. Register X.509 certificates on the server.

   Use the *netsh.exe* tool to register certificates for the app:

   ```
   netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="{<GUID>}"
   ```

   - `<IP>`: Specifies the local IP address for the binding. Don't use a wildcard binding. Use a valid IP address.
   - `<PORT>`: Specifies the port for the binding.
   - `<THUMBPRINT>`: The X.509 certificate thumbprint.
   - `<GUID>`: A developer-generated GUID to represent the app for informational purposes.

   For reference purposes, store the GUID in the app as a package tag:

   - In Visual Studio:
     - Open the app's project properties by right-clicking on the app in **Solution Explorer** and selecting **Properties**.
     - Select the **Package** tab.
     - Enter the GUID that you created in the **Tags** field.
   - When not using Visual Studio:
     - Open the app's project file.
     - Add a `<PackageTags>` property to a new or existing `<PropertyGroup>` with the GUID that you created:

       ```
       <PropertyGroup>
         <PackageTags>9412ee86-c21b-4eb8-bd89-f650fbf44931</PackageTags>
       </PropertyGroup>
       ```

   In the following example:

   - The local IP address of the server is `10.0.0.4`.
   - An online random GUID generator provides the `appid` value.

```
netsh http add sslcert
    ipport=10.0.0.4:443
    certhash=b66ee04419d4ee37464ab8785ff02449980eae10
    appid="{9412ee86-c21b-4eb8-bd89-f650fbf44931}"
```

When a certificate is registered, the tool responds with `SSL Certificate successfully added`.

To delete a certificate registration, use the `delete sslcert` command:

```
netsh http delete sslcert ipport=<IP>:<PORT>
```

Reference documentation for *netsh.exe*:

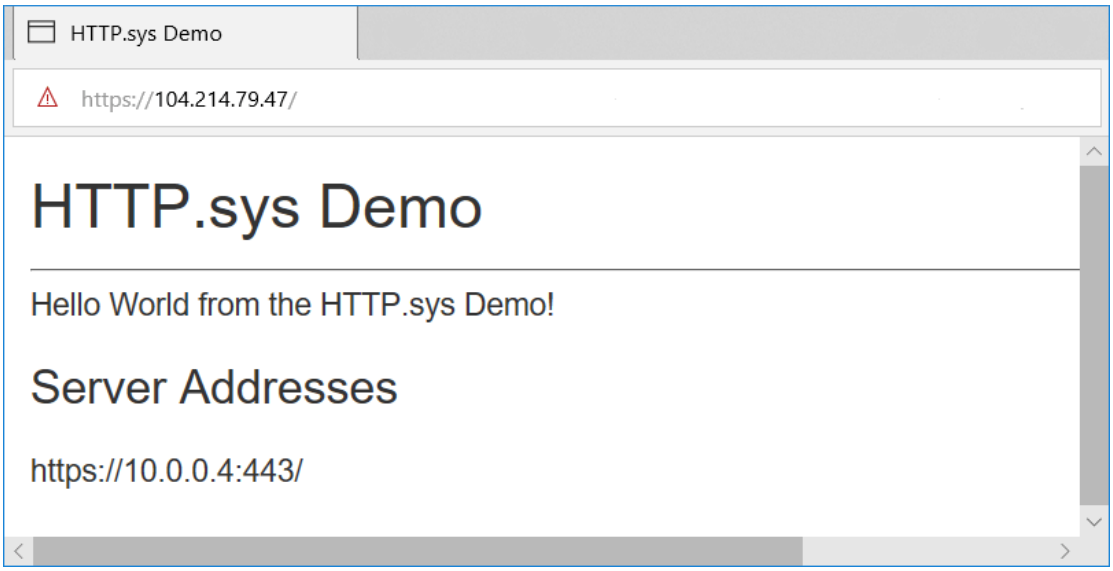
- Netsh Commands for Hypertext Transfer Protocol (HTTP)
- UrlPrefix Strings

8. Run the app.

   Administrator privileges aren't required to run the app when binding to localhost using HTTP (not HTTPS) with a port number greater than 1024. For other configurations (for example, using a local IP address or binding to port 443), run the app with administrator privileges.

   The app responds at the server's public IP address. In this example, the server is reached from the Internet at its public IP address of `104.214.79.47`.

   A development certificate is used in this example. The page loads securely after bypassing the browser's untrusted certificate warning.



Proxy server and load balancer scenarios

For apps hosted by HTTP.sys that interact with requests from the Internet or a corporate network, additional configuration might be required when hosting behind proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Additional resources

- Enable Windows Authentication with HTTP.sys
- HTTP Server API
- aspnet/HttpSysServer GitHub repository (source code)

- The host
- Troubleshoot and debug ASP.NET Core projects

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. HTTP.sys is an alternative to Kestrel server and offers some features that Kestrel doesn't provide.

> **IMPORTANT**
> HTTP.sys isn't compatible with the ASP.NET Core Module and can't be used with IIS or IIS Express.

HTTP.sys supports the following features:

- Windows Authentication
- Port sharing
- HTTPS with SNI
- HTTP/2 over TLS (Windows 10 or later)
- Direct file transmission
- Response caching
- WebSockets (Windows 8 or later)

Supported Windows versions:

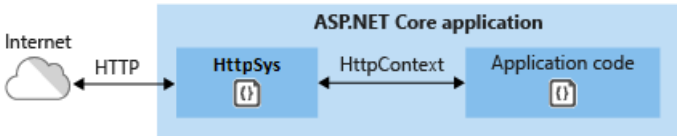
- Windows 7 or later
- Windows Server 2008 R2 or later

View or download sample code (how to download)

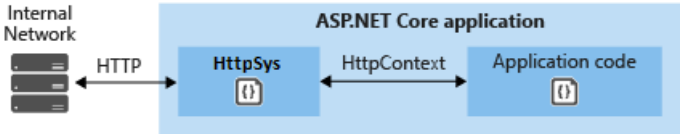
When to use HTTP.sys

HTTP.sys is useful for deployments where:

- There's a need to expose the server directly to the Internet without using IIS.



- An internal deployment requires a feature not available in Kestrel, such as Windows Authentication.



HTTP.sys is mature technology that protects against many types of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of HTTP.sys.

HTTP/2 support

HTTP/2 is enabled for ASP.NET Core apps if the following base requirements are met:

- Windows Server 2016/Windows 10 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/1.1`.

HTTP/2 is enabled by default. If an HTTP/2 connection isn't established, the connection falls back to HTTP/1.1. In a future release of Windows, HTTP/2 configuration flags will be available, including the ability to disable HTTP/2 with HTTP.sys.

Kernel mode authentication with Kerberos

HTTP.sys delegates to kernel mode authentication with the Kerberos authentication protocol. User mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

How to use HTTP.sys

Configure the ASP.NET Core app to use HTTP.sys

A package reference in the project file isn't required when using the Microsoft.AspNetCore.App metapackage (nuget.org). When not using the `Microsoft.AspNetCore.App` metapackage, add a package reference to Microsoft.AspNetCore.Server.HttpSys.

Call the UseHttpSys extension method when building the host, specifying any required HttpSysOptions. The following example sets options to their default values:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .UseHttpSys(options =>
        {
            options.AllowSynchronousIO = true;
            options.Authentication.Schemes = AuthenticationSchemes.None;
            options.Authentication.AllowAnonymous = true;
            options.MaxConnections = null;
            options.MaxRequestBodySize = 30000000;
            options.UrlPrefixes.Add("http://localhost:5000");
        });
```

Additional HTTP.sys configuration is handled through registry settings.

HTTP.sys options

PROPERTY	DESCRIPTION	DEFAULT
AllowSynchronousIO	Control whether synchronous input/output is allowed for the `HttpContext.Request.Body` and `HttpContext.Response.Body`.	`true`
Authentication.AllowAnonymous	Allow anonymous requests.	`true`
Authentication.Schemes	Specify the allowed authentication schemes. May be modified at any time prior to disposing the listener. Values are provided by the AuthenticationSchemes enum: `Basic`, `Kerberos`, `Negotiate`, `None`, and `NTLM`.	`None`

PROPERTY	DESCRIPTION	DEFAULT
EnableResponseCaching	Attempt kernel-mode caching for responses with eligible headers. The response may not include `Set-Cookie`, `Vary`, or `Pragma` headers. It must include a `Cache-Control` header that's `public` and either a `shared-max-age` or `max-age` value, or an `Expires` header.	`true`
MaxAccepts	The maximum number of concurrent accepts.	5 × Environment.ProcessorCount
MaxConnections	The maximum number of concurrent connections to accept. Use `-1` for infinite. Use `null` to use the registry's machine-wide setting.	`null` (machine-wide setting)
MaxRequestBodySize	See the MaxRequestBodySize section.	30000000 bytes (~28.6 MB)
RequestQueueLimit	The maximum number of requests that can be queued.	1000
ThrowWriteExceptions	Indicate if response body writes that fail due to client disconnects should throw exceptions or complete normally.	`false` (complete normally)

PROPERTY	DESCRIPTION	DEFAULT
Timeouts	Expose the HTTP.sys TimeoutManager configuration, which may also be configured in the registry. Follow the API links to learn more about each setting, including default values: • TimeoutManager.DrainEntityBody: Time allowed for the HTTP Server API to drain the entity body on a Keep-Alive connection. • TimeoutManager.EntityBody: Time allowed for the request entity body to arrive. • TimeoutManager.HeaderWait: Time allowed for the HTTP Server API to parse the request header. • TimeoutManager.IdleConnection: Time allowed for an idle connection. • TimeoutManager.MinSendBytesPerSecond: The minimum send rate for the response. • TimeoutManager.RequestQueue: Time allowed for the request to remain in the request queue before the app picks it up.	
UrlPrefixes	Specify the UrlPrefixCollection to register with HTTP.sys. The most useful is UrlPrefixCollection.Add, which is used to add a prefix to the collection. These may be modified at any time prior to disposing the listener.	

MaxRequestBodySize

The maximum allowed size of any request body in bytes. When set to `null`, the maximum request body size is unlimited. This limit has no effect on upgraded connections, which are always unlimited.

The recommended method to override the limit in an ASP.NET Core MVC app for a single `IActionResult` is to use the RequestSizeLimitAttribute attribute on an action method:

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

An exception is thrown if the app attempts to configure the limit on a request after the app has started reading the request. An `IsReadOnly` property can be used to indicate if the `MaxRequestBodySize` property is in a read-only state, meaning it's too late to configure the limit.

If the app should override MaxRequestBodySize per-request, use the IHttpMaxRequestBodySizeFeature:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    ILogger<Startup> logger, IServer server)
{
    app.Use(async (context, next) =>
    {
        context.Features.Get<IHttpMaxRequestBodySizeFeature>()
            .MaxRequestBodySize = 10 * 1024;

        var serverAddressesFeature =
            app.ServerFeatures.Get<IServerAddressesFeature>();
        var addresses = string.Join(", ", serverAddressesFeature?.Addresses);

        logger.LogInformation("Addresses: {Addresses}", addresses);

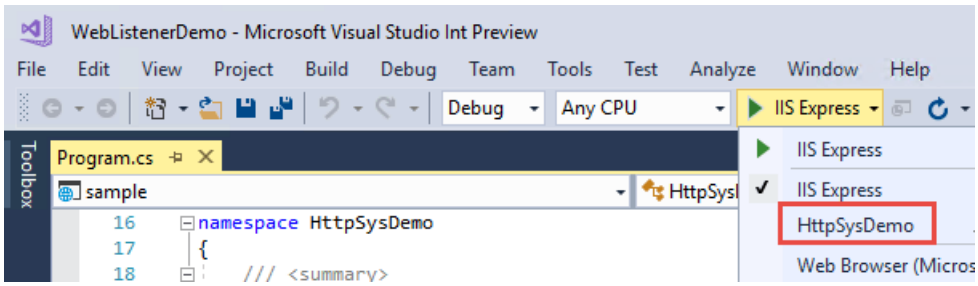
        await next.Invoke();
    });

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    // Enable HTTPS Redirection Middleware when hosting the app securely.
    //app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseMvc();
}
```

If using Visual Studio, make sure the app isn't configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console app, manually change the selected profile, as shown in the following screen shot:



Configure Windows Server

1. Determine the ports to open for the app and use Windows Firewall or the New-NetFirewallRule PowerShell cmdlet to open firewall ports to allow traffic to reach HTTP.sys. In the following commands and app configuration, port 443 is used.

2. When deploying to an Azure VM, open the ports in the Network Security Group. In the following commands and app configuration, port 443 is used.

3. Obtain and install X.509 certificates, if required.

   On Windows, create self-signed certificates using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

   Install either self-signed or CA-signed certificates in the server's **Local Machine** > **Personal** store.

4. If the app is a framework-dependent deployment, install .NET Core, .NET Framework, or both (if the app is a .NET Core app targeting the .NET Framework).

   - **.NET Core**: If the app requires .NET Core, obtain and run the **.NET Core Runtime** installer from .NET Core Downloads. Don't install the full SDK on the server.
   - **.NET Framework**: If the app requires .NET Framework, see the .NET Framework installation guide. Install the required .NET Framework. The installer for the latest .NET Framework is available from the .NET Core Downloads page.

   If the app is a self-contained deployment, the app includes the runtime in its deployment. No framework installation is required on the server.

5. Configure URLs and ports in the app.

   By default, ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, options include:

   - UseUrls
   - `urls` command-line argument
   - `ASPNETCORE_URLS` environment variable
   - UrlPrefixes

   The following code example shows how to use UrlPrefixes with the server's local IP address `10.0.0.4` on port 443:

   ```
   public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
       WebHost.CreateDefaultBuilder(args)
           .UseStartup<Startup>()
           .UseHttpSys(options =>
           {
               options.UrlPrefixes.Add("https://10.0.0.4:443");
           });
   ```

   An advantage of `UrlPrefixes` is that an error message is generated immediately for improperly formatted prefixes.

   The settings in `UrlPrefixes` override `UseUrls` / `urls` / `ASPNETCORE_URLS` settings. Therefore, an advantage of `UseUrls`, `urls`, and the `ASPNETCORE_URLS` environment variable is that it's easier to switch between Kestrel and HTTP.sys.

   HTTP.sys uses the HTTP Server API UrlPrefix string formats.

   > **WARNING**
   > Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings create app security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names or IP addresses rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) isn't a security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 7230: Section 5.4: Host.

6. Preregister URL prefixes on the server.

   The built-in tool for configuring HTTP.sys is *netsh.exe*. *netsh.exe* is used to reserve URL prefixes and assign X.509 certificates. The tool requires administrator privileges.

   Use the *netsh.exe* tool to register URLs for the app:

```
netsh http add urlacl url=<URL> user=<USER>
```

- `<URL>`: The fully qualified Uniform Resource Locator (URL). Don't use a wildcard binding. Use a valid hostname or local IP address. *The URL must include a trailing slash.*
- `<USER>`: Specifies the user or user-group name.

In the following example, the local IP address of the server is `10.0.0.4`:

```
netsh http add urlacl url=https://10.0.0.4:443/ user=Users
```

When a URL is registered, the tool responds with `URL reservation successfully added`.

To delete a registered URL, use the `delete urlacl` command:

```
netsh http delete urlacl url=<URL>
```

7. Register X.509 certificates on the server.

   Use the *netsh.exe* tool to register certificates for the app:

   ```
   netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="{<GUID>}"
   ```

   - `<IP>`: Specifies the local IP address for the binding. Don't use a wildcard binding. Use a valid IP address.
   - `<PORT>`: Specifies the port for the binding.
   - `<THUMBPRINT>`: The X.509 certificate thumbprint.
   - `<GUID>`: A developer-generated GUID to represent the app for informational purposes.

   For reference purposes, store the GUID in the app as a package tag:

   - In Visual Studio:
     - Open the app's project properties by right-clicking on the app in **Solution Explorer** and selecting **Properties**.
     - Select the **Package** tab.
     - Enter the GUID that you created in the **Tags** field.
   - When not using Visual Studio:
     - Open the app's project file.
     - Add a `<PackageTags>` property to a new or existing `<PropertyGroup>` with the GUID that you created:

       ```
       <PropertyGroup>
         <PackageTags>9412ee86-c21b-4eb8-bd89-f650fbf44931</PackageTags>
       </PropertyGroup>
       ```

   In the following example:

   - The local IP address of the server is `10.0.0.4`.
   - An online random GUID generator provides the `appid` value.

```
netsh http add sslcert
    ipport=10.0.0.4:443
    certhash=b66ee04419d4ee37464ab8785ff02449980eae10
    appid="{9412ee86-c21b-4eb8-bd89-f650fbf44931}"
```

When a certificate is registered, the tool responds with `SSL Certificate successfully added`.

To delete a certificate registration, use the `delete sslcert` command:

```
netsh http delete sslcert ipport=<IP>:<PORT>
```

Reference documentation for *netsh.exe*:

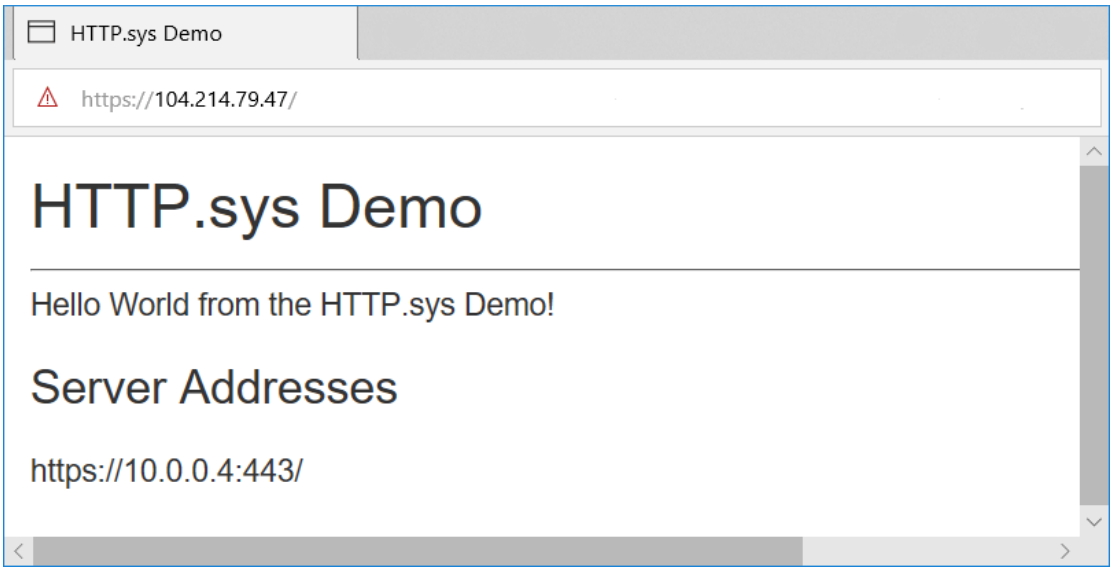
- Netsh Commands for Hypertext Transfer Protocol (HTTP)
- UrlPrefix Strings

8. Run the app.

   Administrator privileges aren't required to run the app when binding to localhost using HTTP (not HTTPS) with a port number greater than 1024. For other configurations (for example, using a local IP address or binding to port 443), run the app with administrator privileges.

   The app responds at the server's public IP address. In this example, the server is reached from the Internet at its public IP address of `104.214.79.47`.

   A development certificate is used in this example. The page loads securely after bypassing the browser's untrusted certificate warning.



Proxy server and load balancer scenarios

For apps hosted by HTTP.sys that interact with requests from the Internet or a corporate network, additional configuration might be required when hosting behind proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Additional resources

- Enable Windows Authentication with HTTP.sys
- HTTP Server API
- aspnet/HttpSysServer GitHub repository (source code)

- The host
- Troubleshoot and debug ASP.NET Core projects

Host ASP.NET Core in a Windows Service

9/22/2020 • 32 minutes to read • Edit Online

An ASP.NET Core app can be hosted on Windows as a Windows Service without using IIS. When hosted as a Windows Service, the app automatically starts after server reboots.

View or download sample code (how to download)

Prerequisites

- ASP.NET Core SDK 2.1 or later
- PowerShell 6.2 or later

Worker Service template

The ASP.NET Core Worker Service template provides a starting point for writing long running service apps. To use the template as a basis for a Windows Service app:

1. Create a Worker Service app from the .NET Core template.
2. Follow the guidance in the App configuration section to update the Worker Service app so that it can run as a Windows Service.

- Visual Studio
- Visual Studio for Mac
- .NET Core CLI

1. Create a new project.
2. Select **Worker Service**. Select **Next**.
3. Provide a project name in the **Project name** field or accept the default project name. Select **Create**.
4. In the **Create a new Worker service** dialog, select **Create**.

App configuration

The app requires a package reference for Microsoft.Extensions.Hosting.WindowsServices.

`IHostBuilder.UseWindowsService` is called when building the host. If the app is running as a Windows Service, the method:

- Sets the host lifetime to `WindowsServiceLifetime`.
- Sets the content root to AppContext.BaseDirectory. For more information, see the Current directory and content root section.
- Enables logging to the event log:
  - The application name is used as the default source name.
  - The default log level is *Warning* or higher for an app based on an ASP.NET Core template that calls `CreateDefaultBuilder` to build the host.
  - Override the default log level with the `Logging:EventLog:LogLevel:Default` key in *appsettings.json*/*appsettings.{Environment}.json* or other configuration provider.
  - Only administrators can create new event sources. When an event source can't be created using the application name, a warning is logged to the *Application* source and event logs are disabled.

In `CreateHostBuilder` of *Program.cs*:

```
Host.CreateDefaultBuilder(args)
    .UseWindowsService()
    ...
```

The following sample apps accompany this topic:

- Background Worker Service Sample: A non-web app sample based on the Worker Service template that uses hosted services for background tasks.
- Web App Service Sample: A Razor Pages web app sample that runs as a Windows Service with hosted services for background tasks.

For MVC guidance, see the articles under Overview of ASP.NET Core MVC and Migrate from ASP.NET Core 2.2 to 3.0.

Deployment type

For information and advice on deployment scenarios, see .NET Core application deployment.

SDK

For a web app-based service that uses the Razor Pages or MVC frameworks, specify the Web SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

If the service only executes background tasks (for example, hosted services), specify the Worker SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

Framework-dependent deployment (FDD)

Framework-dependent deployment (FDD) relies on the presence of a shared system-wide version of .NET Core on the target system. When the FDD scenario is adopted following the guidance in this article, the SDK produces an executable (*.exe*), called a *framework-dependent executable*.

If using the Web SDK, a *web.config* file, which is normally produced when publishing an ASP.NET Core app, is unnecessary for a Windows Services app. To disable the creation of the *web.config* file, add the `<IsTransformWebConfigDisabled>` property set to `true`.

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

Self-contained deployment (SCD)

Self-contained deployment (SCD) doesn't rely on the presence of a shared framework on the host system. The runtime and the app's dependencies are deployed with the app.

A Windows Runtime Identifier (RID) is included in the `<PropertyGroup>` that contains the target framework:

```
<RuntimeIdentifier>win7-x64</RuntimeIdentifier>
```

To publish for multiple RIDs:

- Provide the RIDs in a semicolon-delimited list.
- Use the property name <RuntimeIdentifiers> (plural).

For more information, see .NET Core RID Catalog.

Service user account

To create a user account for a service, use the New-LocalUser cmdlet from an administrative PowerShell 6 command shell.

On Windows 10 October 2018 Update (version 1809/build 10.0.17763) or later:

```
New-LocalUser -Name {SERVICE NAME}
```

On Windows OS earlier than the Windows 10 October 2018 Update (version 1809/build 10.0.17763):

```
powershell -Command "New-LocalUser -Name {SERVICE NAME}"
```

Provide a strong password when prompted.

Unless the `-AccountExpires` parameter is supplied to the New-LocalUser cmdlet with an expiration DateTime, the account doesn't expire.

For more information, see Microsoft.PowerShell.LocalAccounts and Service User Accounts.

An alternative approach to managing users when using Active Directory is to use Managed Service Accounts. For more information, see Group Managed Service Accounts Overview.

Log on as a service rights

To establish *Log on as a service* rights for a service user account:

1. Open the Local Security Policy editor by running *secpol.msc*.
2. Expand the **Local Policies** node and select **User Rights Assignment**.
3. Open the **Log on as a service** policy.
4. Select **Add User or Group**.
5. Provide the object name (user account) using either of the following approaches:
   a. Type the user account (`{DOMAIN OR COMPUTER NAME\USER}`) in the object name field and select **OK** to add the user to the policy.
   b. Select **Advanced**. Select **Find Now**. Select the user account from the list. Select **OK**. Select **OK** again to add the user to the policy.
6. Select **OK** or **Apply** to accept the changes.

Create and manage the Windows Service

Create a service

Use PowerShell commands to register a service. From an administrative PowerShell 6 command shell, execute the following commands:

```
$acl = Get-Acl "{EXE PATH}"
$aclRuleArgs = {DOMAIN OR COMPUTER NAME\USER}, "Read,Write,ReadAndExecute", "ContainerInherit,ObjectInherit",
"None", "Allow"
$accessRule = New-Object System.Security.AccessControl.FileSystemAccessRule($aclRuleArgs)
$acl.SetAccessRule($accessRule)
$acl | Set-Acl "{EXE PATH}"

New-Service -Name {SERVICE NAME} -BinaryPathName {EXE FILE PATH} -Credential {DOMAIN OR COMPUTER NAME\USER} -
Description "{DESCRIPTION}" -DisplayName "{DISPLAY NAME}" -StartupType Automatic
```

- `{EXE PATH}`: Path to the app's folder on the host (for example, `d:\myservice`). Don't include the app's executable in the path. A trailing slash isn't required.
- `{DOMAIN OR COMPUTER NAME\USER}`: Service user account (for example, `Contoso\ServiceUser`).
- `{SERVICE NAME}`: Service name (for example, `MyService`).
- `{EXE FILE PATH}`: The app's executable path (for example, `d:\myservice\myservice.exe`). Include the executable's file name with extension.
- `{DESCRIPTION}`: Service description (for example, `My sample service`).
- `{DISPLAY NAME}`: Service display name (for example, `My Service`).

Start a service

Start a service with the following PowerShell 6 command:

```
Start-Service -Name {SERVICE NAME}
```

The command takes a few seconds to start the service.

Determine a service's status

To check the status of a service, use the following PowerShell 6 command:

```
Get-Service -Name {SERVICE NAME}
```

The status is reported as one of the following values:

- `Starting`
- `Running`
- `Stopping`
- `Stopped`

Stop a service

Stop a service with the following Powershell 6 command:

```
Stop-Service -Name {SERVICE NAME}
```

Remove a service

After a short delay to stop a service, remove a service with the following Powershell 6 command:

```
Remove-Service -Name {SERVICE NAME}
```

Proxy server and load balancer scenarios

Services that interact with requests from the Internet or a corporate network and are behind a proxy or load

balancer might require additional configuration. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Configure endpoints

By default, ASP.NET Core binds to `http://localhost:5000`. Configure the URL and port by setting the `ASPNETCORE_URLS` environment variable.

For additional URL and port configuration approaches, see the relevant server article:

- Kestrel web server implementation in ASP.NET Core
- HTTP.sys web server implementation in ASP.NET Core

The preceding guidance covers support for HTTPS endpoints. For example, configure the app for HTTPS when authentication is used with a Windows Service.

> **NOTE**
> Use of the ASP.NET Core HTTPS development certificate to secure a service endpoint isn't supported.

Current directory and content root

The current working directory returned by calling GetCurrentDirectory for a Windows Service is the *C:\WINDOWS\system32* folder. The *system32* folder isn't a suitable location to store a service's files (for example, settings files). Use one of the following approaches to maintain and access a service's assets and settings files.

Use ContentRootPath or ContentRootFileProvider

Use IHostEnvironment.ContentRootPath or ContentRootFileProvider to locate an app's resources.

When the app runs as a service, UseWindowsService sets the ContentRootPath to AppContext.BaseDirectory.

The app's default settings files, *appsettings.json* and *appsettings.{Environment}.json*, are loaded from the app's content root by calling CreateDefaultBuilder during host construction.

For other settings files loaded by developer code in ConfigureAppConfiguration, there's no need to call SetBasePath. In the following example, the *custom_settings.json* file exists in the app's content root and is loaded without explicitly setting a base path:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseWindowsService()
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("custom_settings.json");
            })
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHostedService<Worker>();
            });
}
```

Don't attempt to use GetCurrentDirectory to obtain a resource path because a Windows Service app returns the *C:\WINDOWS\system32* folder as its current directory.

Store a service's files in a suitable location on disk

Specify an absolute path with SetBasePath when using an IConfigurationBuilder to the folder containing the files.

Troubleshoot

To troubleshoot a Windows Service app, see Troubleshoot and debug ASP.NET Core projects.

Common errors

- An old or pre-release version of PowerShell is in use.
- The registered service doesn't use the app's **published** output from the dotnet publish command. Output of the dotnet build command isn't supported for app deployment. Published assets are found in either of the following folders depending on the deployment type:
  - *bin/Release/{TARGET FRAMEWORK}/publish* (FDD)
  - *bin/Release/{TARGET FRAMEWORK}/{RUNTIME IDENTIFIER}/publish* (SCD)
- The service isn't in the RUNNING state.
- The paths to resources that the app uses (for example, certificates) are incorrect. The base path of a Windows Service is *c:\Windows\System32*.
- The user doesn't have *Log on as a service* rights.
- The user's password is expired or incorrectly passed when executing the `New-Service` PowerShell command.
- The app requires ASP.NET Core authentication but isn't configured for secure connections (HTTPS).
- The request URL port is incorrect or not configured correctly in the app.

System and Application Event Logs

Access the System and Application Event Logs:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **System** to open the System Event Log. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app.

Run the app at a command prompt

Many startup errors don't produce useful information in the event logs. You can find the cause of some errors by running the app at a command prompt on the hosting system. To log additional detail from the app, lower the log level or run the app in the Development environment.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Slow or hanging app

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`.
2. Run the EnableDumps PowerShell script with the application executable name:

   ```
   .\EnableDumps {APPLICATION EXE} c:\dumps
   ```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:

   ```
   .\DisableDumps {APPLICATION EXE}
   ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Additional resources

- Kestrel endpoint configuration (includes HTTPS configuration and SNI support)
- .NET Generic Host
- Troubleshoot and debug ASP.NET Core projects

An ASP.NET Core app can be hosted on Windows as a Windows Service without using IIS. When hosted as a Windows Service, the app automatically starts after server reboots.

View or download sample code (how to download)

Prerequisites

- ASP.NET Core SDK 2.1 or later
- PowerShell 6.2 or later

App configuration

The app requires package references for Microsoft.AspNetCore.Hosting.WindowsServices and Microsoft.Extensions.Logging.EventLog.

To test and debug when running outside of a service, add code to determine if the app is running as a service or a

console app. Inspect if the debugger is attached or a `--console` switch is present. If either condition is true (the app isn't run as a service), call Run. If the conditions are false (the app is run as a service):

- Call SetCurrentDirectory and use a path to the app's published location. Don't call GetCurrentDirectory to obtain the path because a Windows Service app returns the *C:\WINDOWS\system32* folder when GetCurrentDirectory is called. For more information, see the Current directory and content root section. This step is performed before the app is configured in `CreateWebHostBuilder`.
- Call RunAsService to run the app as a service.

Because the Command-line Configuration Provider requires name-value pairs for command-line arguments, the `--console` switch is removed from the arguments before CreateDefaultBuilder receives the arguments.

To write to the Windows Event Log, add the EventLog provider to ConfigureLogging. Set the logging level with the `Logging:LogLevel:Default` key in the *appsettings.Production.json* file.

In the following example from the sample app, `RunAsCustomService` is called instead of RunAsService in order to handle lifetime events within the app. For more information, see the Handle starting and stopping events section.

```
public class Program
{
    public static void Main(string[] args)
    {
        var isService = !(Debugger.IsAttached || args.Contains("--console"));

        if (isService)
        {
            var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
            var pathToContentRoot = Path.GetDirectoryName(pathToExe);
            Directory.SetCurrentDirectory(pathToContentRoot);
        }

        var builder = CreateWebHostBuilder(
            args.Where(arg => arg != "--console").ToArray());

        var host = builder.Build();

        if (isService)
        {
            // To run the app without the CustomWebHostService change the
            // next line to host.RunAsService();
            host.RunAsCustomService();
        }
        else
        {
            host.Run();
        }
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureLogging((hostingContext, logging) =>
            {
                logging.AddEventLog();
            })
            .ConfigureAppConfiguration((context, config) =>
            {
                // Configure the app here.
            })
            .UseStartup<Startup>();
}
```

Deployment type

For information and advice on deployment scenarios, see .NET Core application deployment.

SDK

For a web app-based service that uses the Razor Pages or MVC frameworks, specify the Web SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

If the service only executes background tasks (for example, hosted services), specify the Worker SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

Framework-dependent deployment (FDD)

Framework-dependent deployment (FDD) relies on the presence of a shared system-wide version of .NET Core on the target system. When the FDD scenario is adopted following the guidance in this article, the SDK produces an executable (*.exe*), called a *framework-dependent executable*.

The Windows Runtime Identifier (RID) (<RuntimeIdentifier>) contains the target framework. In the following example, the RID is set to `win7-x64`. The `<SelfContained>` property is set to `false`. These properties instruct the SDK to generate an executable (*.exe*) file for Windows and an app that depends on the shared .NET Core framework.

A *web.config* file, which is normally produced when publishing an ASP.NET Core app, is unnecessary for a Windows Services app. To disable the creation of the *web.config* file, add the `<IsTransformWebConfigDisabled>` property set to `true`.

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.2</TargetFramework>
  <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
  <SelfContained>false</SelfContained>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

Self-contained deployment (SCD)

Self-contained deployment (SCD) doesn't rely on the presence of a shared framework on the host system. The runtime and the app's dependencies are deployed with the app.

A Windows Runtime Identifier (RID) is included in the `<PropertyGroup>` that contains the target framework:

```
<RuntimeIdentifier>win7-x64</RuntimeIdentifier>
```

To publish for multiple RIDs:

- Provide the RIDs in a semicolon-delimited list.
- Use the property name <RuntimeIdentifiers> (plural).

For more information, see .NET Core RID Catalog.

A `<SelfContained>` property is set to `true`:

```
<SelfContained>true</SelfContained>
```

Service user account

To create a user account for a service, use the New-LocalUser cmdlet from an administrative PowerShell 6 command shell.

On Windows 10 October 2018 Update (version 1809/build 10.0.17763) or later:

```
New-LocalUser -Name {SERVICE NAME}
```

On Windows OS earlier than the Windows 10 October 2018 Update (version 1809/build 10.0.17763):

```
powershell -Command "New-LocalUser -Name {SERVICE NAME}"
```

Provide a strong password when prompted.

Unless the `-AccountExpires` parameter is supplied to the New-LocalUser cmdlet with an expiration DateTime, the account doesn't expire.

For more information, see Microsoft.PowerShell.LocalAccounts and Service User Accounts.

An alternative approach to managing users when using Active Directory is to use Managed Service Accounts. For more information, see Group Managed Service Accounts Overview.

Log on as a service rights

To establish *Log on as a service* rights for a service user account:

1. Open the Local Security Policy editor by running *secpol.msc*.
2. Expand the **Local Policies** node and select **User Rights Assignment**.
3. Open the **Log on as a service** policy.
4. Select **Add User or Group**.
5. Provide the object name (user account) using either of the following approaches:
   a. Type the user account (`{DOMAIN OR COMPUTER NAME\USER}`) in the object name field and select **OK** to add the user to the policy.
   b. Select **Advanced**. Select **Find Now**. Select the user account from the list. Select **OK**. Select **OK** again to add the user to the policy.
6. Select **OK** or **Apply** to accept the changes.

Create and manage the Windows Service

Create a service

Use PowerShell commands to register a service. From an administrative PowerShell 6 command shell, execute the following commands:

```
$acl = Get-Acl "{EXE PATH}"
$aclRuleArgs = {DOMAIN OR COMPUTER NAME\USER}, "Read,Write,ReadAndExecute", "ContainerInherit,ObjectInherit",
"None", "Allow"
$accessRule = New-Object System.Security.AccessControl.FileSystemAccessRule($aclRuleArgs)
$acl.SetAccessRule($accessRule)
$acl | Set-Acl "{EXE PATH}"

New-Service -Name {SERVICE NAME} -BinaryPathName {EXE FILE PATH} -Credential {DOMAIN OR COMPUTER NAME\USER} -
Description "{DESCRIPTION}" -DisplayName "{DISPLAY NAME}" -StartupType Automatic
```

- `{EXE PATH}`: Path to the app's folder on the host (for example, `d:\myservice`). Don't include the app's executable in the path. A trailing slash isn't required.
- `{DOMAIN OR COMPUTER NAME\USER}`: Service user account (for example, `Contoso\ServiceUser`).
- `{SERVICE NAME}`: Service name (for example, `MyService`).
- `{EXE FILE PATH}`: The app's executable path (for example, `d:\myservice\myservice.exe`). Include the executable's file name with extension.
- `{DESCRIPTION}`: Service description (for example, `My sample service`).
- `{DISPLAY NAME}`: Service display name (for example, `My Service`).

Start a service

Start a service with the following PowerShell 6 command:

```
Start-Service -Name {SERVICE NAME}
```

The command takes a few seconds to start the service.

Determine a service's status

To check the status of a service, use the following PowerShell 6 command:

```
Get-Service -Name {SERVICE NAME}
```

The status is reported as one of the following values:

- `Starting`
- `Running`
- `Stopping`
- `Stopped`

Stop a service

Stop a service with the following Powershell 6 command:

```
Stop-Service -Name {SERVICE NAME}
```

Remove a service

After a short delay to stop a service, remove a service with the following Powershell 6 command:

```
Remove-Service -Name {SERVICE NAME}
```

Handle starting and stopping events

To handle OnStarting, OnStarted, and OnStopping events:

1. Create a class that derives from WebHostService with the `OnStarting`, `OnStarted`, and `OnStopping` methods:

```
[DesignerCategory("Code")]
internal class CustomWebHostService : WebHostService
{
    private ILogger _logger;

    public CustomWebHostService(IWebHost host) : base(host)
    {
        _logger = host.Services
            .GetRequiredService<ILogger<CustomWebHostService>>();
    }

    protected override void OnStarting(string[] args)
    {
        _logger.LogInformation("OnStarting method called.");
        base.OnStarting(args);
    }

    protected override void OnStarted()
    {
        _logger.LogInformation("OnStarted method called.");
        base.OnStarted();
    }

    protected override void OnStopping()
    {
        _logger.LogInformation("OnStopping method called.");
        base.OnStopping();
    }
}
```

2. Create an extension method for IWebHost that passes the `CustomWebHostService` to Run:

```
public static class WebHostServiceExtensions
{
    public static void RunAsCustomService(this IWebHost host)
    {
        var webHostService = new CustomWebHostService(host);
        ServiceBase.Run(webHostService);
    }
}
```

3. In `Program.Main`, call the `RunAsCustomService` extension method instead of RunAsService:

```
host.RunAsCustomService();
```

   To see the location of RunAsService in `Program.Main`, refer to the code sample shown in the Deployment type section.

Proxy server and load balancer scenarios

Services that interact with requests from the Internet or a corporate network and are behind a proxy or load balancer might require additional configuration. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Configure endpoints

By default, ASP.NET Core binds to `http://localhost:5000`. Configure the URL and port by setting the `ASPNETCORE_URLS` environment variable.

For additional URL and port configuration approaches, see the relevant server article:

- Kestrel web server implementation in ASP.NET Core
- HTTP.sys web server implementation in ASP.NET Core

The preceding guidance covers support for HTTPS endpoints. For example, configure the app for HTTPS when authentication is used with a Windows Service.

> **NOTE**
> Use of the ASP.NET Core HTTPS development certificate to secure a service endpoint isn't supported.

Current directory and content root

The current working directory returned by calling GetCurrentDirectory for a Windows Service is the *C:\WINDOWS\system32* folder. The *system32* folder isn't a suitable location to store a service's files (for example, settings files). Use one of the following approaches to maintain and access a service's assets and settings files.

Set the content root path to the app's folder

The ContentRootPath is the same path provided to the `binPath` argument when a service is created. Instead of calling `GetCurrentDirectory` to create paths to settings files, call SetCurrentDirectory with the path to the app's content root.

In `Program.Main`, determine the path to the folder of the service's executable and use the path to establish the app's content root:

```
var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
var pathToContentRoot = Path.GetDirectoryName(pathToExe);
Directory.SetCurrentDirectory(pathToContentRoot);

CreateWebHostBuilder(args)
    .Build()
    .RunAsService();
```

Store a service's files in a suitable location on disk

Specify an absolute path with SetBasePath when using an IConfigurationBuilder to the folder containing the files.

Troubleshoot

To troubleshoot a Windows Service app, see Troubleshoot and debug ASP.NET Core projects.

Common errors

- An old or pre-release version of PowerShell is in use.
- The registered service doesn't use the app's **published** output from the dotnet publish command. Output of the dotnet build command isn't supported for app deployment. Published assets are found in either of the following folders depending on the deployment type:
  - *bin/Release/{TARGET FRAMEWORK}/publish* (FDD)
  - *bin/Release/{TARGET FRAMEWORK}/{RUNTIME IDENTIFIER}/publish* (SCD)
- The service isn't in the RUNNING state.
- The paths to resources that the app uses (for example, certificates) are incorrect. The base path of a Windows Service is *c:\Windows\System32*.
- The user doesn't have *Log on as a service* rights.
- The user's password is expired or incorrectly passed when executing the `New-Service` PowerShell command.

- The app requires ASP.NET Core authentication but isn't configured for secure connections (HTTPS).
- The request URL port is incorrect or not configured correctly in the app.

System and Application Event Logs

Access the System and Application Event Logs:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **System** to open the System Event Log. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app.

Run the app at a command prompt

Many startup errors don't produce useful information in the event logs. You can find the cause of some errors by running the app at a command prompt on the hosting system. To log additional detail from the app, lower the log level or run the app in the Development environment.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Slow or hanging app

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`.
2. Run the EnableDumps PowerShell script with the application executable name:

   ```
   .\EnableDumps {APPLICATION EXE} c:\dumps
   ```
3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:

   ```
   .\DisableDumps {APPLICATION EXE}
   ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Additional resources

- Kestrel endpoint configuration (includes HTTPS configuration and SNI support)
- ASP.NET Core Web Host
- Troubleshoot and debug ASP.NET Core projects

An ASP.NET Core app can be hosted on Windows as a Windows Service without using IIS. When hosted as a Windows Service, the app automatically starts after server reboots.

View or download sample code (how to download)

Prerequisites

- ASP.NET Core SDK 2.1 or later
- PowerShell 6.2 or later

App configuration

The app requires package references for Microsoft.AspNetCore.Hosting.WindowsServices and Microsoft.Extensions.Logging.EventLog.

To test and debug when running outside of a service, add code to determine if the app is running as a service or a console app. Inspect if the debugger is attached or a `--console` switch is present. If either condition is true (the app isn't run as a service), call Run. If the conditions are false (the app is run as a service):

- Call SetCurrentDirectory and use a path to the app's published location. Don't call GetCurrentDirectory to obtain the path because a Windows Service app returns the *C:\WINDOWS\system32* folder when GetCurrentDirectory is called. For more information, see the Current directory and content root section. This step is performed before the app is configured in `CreateWebHostBuilder`.
- Call RunAsService to run the app as a service.

Because the Command-line Configuration Provider requires name-value pairs for command-line arguments, the `--console` switch is removed from the arguments before CreateDefaultBuilder receives the arguments.

To write to the Windows Event Log, add the EventLog provider to ConfigureLogging. Set the logging level with the `Logging:LogLevel:Default` key in the *appsettings.Production.json* file.

In the following example from the sample app, `RunAsCustomService` is called instead of RunAsService in order to handle lifetime events within the app. For more information, see the Handle starting and stopping events section.

```
public class Program
{
    public static void Main(string[] args)
    {
        var isService = !(Debugger.IsAttached || args.Contains("--console"));

        if (isService)
        {
            var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
            var pathToContentRoot = Path.GetDirectoryName(pathToExe);
            Directory.SetCurrentDirectory(pathToContentRoot);
        }

        var builder = CreateWebHostBuilder(
            args.Where(arg => arg != "--console").ToArray());

        var host = builder.Build();

        if (isService)
        {
            // To run the app without the CustomWebHostService change the
            // next line to host.RunAsService();
            host.RunAsCustomService();
        }
        else
        {
            host.Run();
        }
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .ConfigureLogging((hostingContext, logging) =>
            {
                logging.AddEventLog();
            })
            .ConfigureAppConfiguration((context, config) =>
            {
                // Configure the app here.
            })
            .UseStartup<Startup>();
}
```

Deployment type

For information and advice on deployment scenarios, see .NET Core application deployment.

SDK

For a web app-based service that uses the Razor Pages or MVC frameworks, specify the Web SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

If the service only executes background tasks (for example, hosted services), specify the Worker SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

Framework-dependent deployment (FDD)

Framework-dependent deployment (FDD) relies on the presence of a shared system-wide version of .NET Core on

the target system. When the FDD scenario is adopted following the guidance in this article, the SDK produces an executable (*.exe*), called a *framework-dependent executable*.

The Windows Runtime Identifier (RID) (<RuntimeIdentifier>) contains the target framework. In the following example, the RID is set to `win7-x64`. The `<SelfContained>` property is set to `false`. These properties instruct the SDK to generate an executable (*.exe*) file for Windows and an app that depends on the shared .NET Core framework.

The `<UseAppHost>` property is set to `true`. This property provides the service with an activation path (an executable, *.exe*) for an FDD.

A *web.config* file, which is normally produced when publishing an ASP.NET Core app, is unnecessary for a Windows Services app. To disable the creation of the *web.config* file, add the `<IsTransformWebConfigDisabled>` property set to `true`.

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.2</TargetFramework>
  <RuntimeIdentifier>win7-x64</RuntimeIdentifier>
  <UseAppHost>true</UseAppHost>
  <SelfContained>false</SelfContained>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

Self-contained deployment (SCD)

Self-contained deployment (SCD) doesn't rely on the presence of a shared framework on the host system. The runtime and the app's dependencies are deployed with the app.

A Windows Runtime Identifier (RID) is included in the `<PropertyGroup>` that contains the target framework:

```
<RuntimeIdentifier>win7-x64</RuntimeIdentifier>
```

To publish for multiple RIDs:

- Provide the RIDs in a semicolon-delimited list.
- Use the property name <RuntimeIdentifiers> (plural).

For more information, see .NET Core RID Catalog.

A `<SelfContained>` property is set to `true`:

```
<SelfContained>true</SelfContained>
```

Service user account

To create a user account for a service, use the New-LocalUser cmdlet from an administrative PowerShell 6 command shell.

On Windows 10 October 2018 Update (version 1809/build 10.0.17763) or later:

```
New-LocalUser -Name {SERVICE NAME}
```

On Windows OS earlier than the Windows 10 October 2018 Update (version 1809/build 10.0.17763):

```
powershell -Command "New-LocalUser -Name {SERVICE NAME}"
```

Provide a strong password when prompted.

Unless the `-AccountExpires` parameter is supplied to the New-LocalUser cmdlet with an expiration DateTime, the account doesn't expire.

For more information, see Microsoft.PowerShell.LocalAccounts and Service User Accounts.

An alternative approach to managing users when using Active Directory is to use Managed Service Accounts. For more information, see Group Managed Service Accounts Overview.

Log on as a service rights

To establish *Log on as a service* rights for a service user account:

1. Open the Local Security Policy editor by running *secpol.msc*.
2. Expand the **Local Policies** node and select **User Rights Assignment**.
3. Open the **Log on as a service** policy.
4. Select **Add User or Group**.
5. Provide the object name (user account) using either of the following approaches:
   a. Type the user account (`{DOMAIN OR COMPUTER NAME\USER}`) in the object name field and select **OK** to add the user to the policy.
   b. Select **Advanced**. Select **Find Now**. Select the user account from the list. Select **OK**. Select **OK** again to add the user to the policy.
6. Select **OK** or **Apply** to accept the changes.

Create and manage the Windows Service

Create a service

Use PowerShell commands to register a service. From an administrative PowerShell 6 command shell, execute the following commands:

```
$acl = Get-Acl "{EXE PATH}"
$aclRuleArgs = {DOMAIN OR COMPUTER NAME\USER}, "Read,Write,ReadAndExecute", "ContainerInherit,ObjectInherit",
"None", "Allow"
$accessRule = New-Object System.Security.AccessControl.FileSystemAccessRule($aclRuleArgs)
$acl.SetAccessRule($accessRule)
$acl | Set-Acl "{EXE PATH}"

New-Service -Name {SERVICE NAME} -BinaryPathName {EXE FILE PATH} -Credential {DOMAIN OR COMPUTER NAME\USER} -
Description "{DESCRIPTION}" -DisplayName "{DISPLAY NAME}" -StartupType Automatic
```

- `{EXE PATH}` : Path to the app's folder on the host (for example, `d:\myservice`). Don't include the app's executable in the path. A trailing slash isn't required.
- `{DOMAIN OR COMPUTER NAME\USER}` : Service user account (for example, `Contoso\ServiceUser`).
- `{SERVICE NAME}` : Service name (for example, `MyService`).
- `{EXE FILE PATH}` : The app's executable path (for example, `d:\myservice\myservice.exe`). Include the executable's file name with extension.
- `{DESCRIPTION}` : Service description (for example, `My sample service`).
- `{DISPLAY NAME}` : Service display name (for example, `My Service`).

Start a service

Start a service with the following PowerShell 6 command:

```
Start-Service -Name {SERVICE NAME}
```

The command takes a few seconds to start the service.

Determine a service's status

To check the status of a service, use the following PowerShell 6 command:

```
Get-Service -Name {SERVICE NAME}
```

The status is reported as one of the following values:

- `Starting`
- `Running`
- `Stopping`
- `Stopped`

Stop a service

Stop a service with the following Powershell 6 command:

```
Stop-Service -Name {SERVICE NAME}
```

Remove a service

After a short delay to stop a service, remove a service with the following Powershell 6 command:

```
Remove-Service -Name {SERVICE NAME}
```

Handle starting and stopping events

To handle OnStarting, OnStarted, and OnStopping events:

1. Create a class that derives from WebHostService with the `OnStarting`, `OnStarted`, and `OnStopping` methods:

```
[DesignerCategory("Code")]
internal class CustomWebHostService : WebHostService
{
    private ILogger _logger;

    public CustomWebHostService(IWebHost host) : base(host)
    {
        _logger = host.Services
            .GetRequiredService<ILogger<CustomWebHostService>>();
    }

    protected override void OnStarting(string[] args)
    {
        _logger.LogInformation("OnStarting method called.");
        base.OnStarting(args);
    }

    protected override void OnStarted()
    {
        _logger.LogInformation("OnStarted method called.");
        base.OnStarted();
    }

    protected override void OnStopping()
    {
        _logger.LogInformation("OnStopping method called.");
        base.OnStopping();
    }
}
```

2. Create an extension method for IWebHost that passes the `CustomWebHostService` to Run:

```
public static class WebHostServiceExtensions
{
    public static void RunAsCustomService(this IWebHost host)
    {
        var webHostService = new CustomWebHostService(host);
        ServiceBase.Run(webHostService);
    }
}
```

3. In `Program.Main`, call the `RunAsCustomService` extension method instead of RunAsService:

```
host.RunAsCustomService();
```

   To see the location of RunAsService in `Program.Main`, refer to the code sample shown in the Deployment type section.

Proxy server and load balancer scenarios

Services that interact with requests from the Internet or a corporate network and are behind a proxy or load balancer might require additional configuration. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Configure endpoints

By default, ASP.NET Core binds to `http://localhost:5000`. Configure the URL and port by setting the `ASPNETCORE_URLS` environment variable.

For additional URL and port configuration approaches, see the relevant server article:

- Kestrel web server implementation in ASP.NET Core
- HTTP.sys web server implementation in ASP.NET Core

The preceding guidance covers support for HTTPS endpoints. For example, configure the app for HTTPS when authentication is used with a Windows Service.

> **NOTE**
> Use of the ASP.NET Core HTTPS development certificate to secure a service endpoint isn't supported.

Current directory and content root

The current working directory returned by calling GetCurrentDirectory for a Windows Service is the *C:\WINDOWS\system32* folder. The *system32* folder isn't a suitable location to store a service's files (for example, settings files). Use one of the following approaches to maintain and access a service's assets and settings files.

Set the content root path to the app's folder

The ContentRootPath is the same path provided to the `binPath` argument when a service is created. Instead of calling `GetCurrentDirectory` to create paths to settings files, call SetCurrentDirectory with the path to the app's content root.

In `Program.Main`, determine the path to the folder of the service's executable and use the path to establish the app's content root:

```
var pathToExe = Process.GetCurrentProcess().MainModule.FileName;
var pathToContentRoot = Path.GetDirectoryName(pathToExe);
Directory.SetCurrentDirectory(pathToContentRoot);

CreateWebHostBuilder(args)
    .Build()
    .RunAsService();
```

Store a service's files in a suitable location on disk

Specify an absolute path with SetBasePath when using an IConfigurationBuilder to the folder containing the files.

Troubleshoot

To troubleshoot a Windows Service app, see Troubleshoot and debug ASP.NET Core projects.

Common errors

- An old or pre-release version of PowerShell is in use.
- The registered service doesn't use the app's **published** output from the dotnet publish command. Output of the dotnet build command isn't supported for app deployment. Published assets are found in either of the following folders depending on the deployment type:
  - *bin/Release/{TARGET FRAMEWORK}/publish* (FDD)
  - *bin/Release/{TARGET FRAMEWORK}/{RUNTIME IDENTIFIER}/publish* (SCD)
- The service isn't in the RUNNING state.
- The paths to resources that the app uses (for example, certificates) are incorrect. The base path of a Windows Service is *c:\Windows\System32*.
- The user doesn't have *Log on as a service* rights.
- The user's password is expired or incorrectly passed when executing the `New-Service` PowerShell command.

- The app requires ASP.NET Core authentication but isn't configured for secure connections (HTTPS).
- The request URL port is incorrect or not configured correctly in the app.

System and Application Event Logs

Access the System and Application Event Logs:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **System** to open the System Event Log. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app.

Run the app at a command prompt

Many startup errors don't produce useful information in the event logs. You can find the cause of some errors by running the app at a command prompt on the hosting system. To log additional detail from the app, lower the log level or run the app in the Development environment.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Slow or hanging app

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`.
2. Run the EnableDumps PowerShell script with the application executable name:

   ```
   .\EnableDumps {APPLICATION EXE} c:\dumps
   ```
3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:

   ```
   .\DisableDumps {APPLICATION EXE}
   ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> **WARNING**
>
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App hangs, fails during startup, or runs normally

When an app *hangs* (stops responding but doesn't crash), fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Additional resources

- Kestrel endpoint configuration (includes HTTPS configuration and SNI support)
- ASP.NET Core Web Host
- Troubleshoot and debug ASP.NET Core projects

Host ASP.NET Core on Linux with Nginx

9/22/2020 • 14 minutes to read • Edit Online

By Sourabh Shirhatti

This guide explains setting up a production-ready ASP.NET Core environment on an Ubuntu 16.04 server. These instructions likely work with newer versions of Ubuntu, but the instructions haven't been tested with newer versions.

For information on other Linux distributions supported by ASP.NET Core, see Prerequisites for .NET Core on Linux.

> **NOTE**
> For Ubuntu 14.04, *supervisord* is recommended as a solution for monitoring the Kestrel process. *systemd* isn't available on Ubuntu 14.04. For Ubuntu 14.04 instructions, see the previous version of this topic.

This guide:

- Places an existing ASP.NET Core app behind a reverse proxy server.
- Sets up the reverse proxy server to forward requests to the Kestrel web server.
- Ensures the web app runs on startup as a daemon.
- Configures a process management tool to help restart the web app.

Prerequisites

1. Access to an Ubuntu 16.04 server with a standard user account with sudo privilege.
2. Install the .NET Core runtime on the server.
   a. Visit the Download .NET Core page.
   b. Select the latest non-preview .NET Core version.
   c. Download the latest non-preview runtime in the table under **Run apps - Runtime**.
   d. Select the Linux **Package manager instructions** link and follow the Ubuntu instructions for your version of Ubuntu.
3. An existing ASP.NET Core app.

At any point in the future after upgrading the shared framework, restart the ASP.NET Core apps hosted by the server.

Publish and copy over the app

Configure the app for a framework-dependent deployment.

If the app is run locally and isn't configured to make secure connections (HTTPS), adopt either of the following approaches:

- Configure the app to handle secure local connections. For more information, see the HTTPS configuration section.
- Remove `https://localhost:5001` (if present) from the `applicationUrl` property in the *Properties/launchSettings.json* file.

Run dotnet publish from the development environment to package an app into a directory (for example, *bin/Release/<target_framework_moniker>/publish*) that can run on the server:

```
dotnet publish --configuration Release
```

The app can also be published as a self-contained deployment if you prefer not to maintain the .NET Core runtime on the server.

Copy the ASP.NET Core app to the server using a tool that integrates into the organization's workflow (for example, SCP, SFTP). It's common to locate web apps under the *var* directory (for example, *var/www/helloapp*).

> **NOTE**
> Under a production deployment scenario, a continuous integration workflow does the work of publishing the app and copying the assets to the server.

Test the app:

1. From the command line, run the app: `dotnet <app_assembly>.dll`.
2. In a browser, navigate to `http://<serveraddress>:<port>` to verify the app works on Linux locally.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web apps. A reverse proxy terminates the HTTP request and forwards it to the ASP.NET Core app.

Use a reverse proxy server

Kestrel is great for serving dynamic content from ASP.NET Core. However, the web serving capabilities aren't as feature rich as servers such as IIS, Apache, or Nginx. A reverse proxy server can offload work such as serving static content, caching requests, compressing requests, and HTTPS termination from the HTTP server. A reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, a single instance of Nginx is used. It runs on the same server, alongside the HTTP server. Based on requirements, a different setup may be chosen.

Because requests are forwarded by reverse proxy, use the Forwarded Headers Middleware from the Microsoft.AspNetCore.HttpOverrides package. The middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header, so that redirect URIs and other security policies work correctly.

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Invoke the UseForwardedHeaders method at the top of `Startup.Configure` before calling other middleware. Configure the middleware to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers:

```
// using Microsoft.AspNetCore.HttpOverrides;

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

If no ForwardedHeadersOptions are specified to the middleware, the default headers to forward are `None`.

Proxies running on loopback addresses (127.0.0.0/8, [::1]), including the standard localhost address (127.0.0.1), are trusted by default. If other trusted proxies or networks within the organization handle requests between the Internet and the web server, add them to the list of KnownProxies or KnownNetworks with ForwardedHeadersOptions. The following example adds a trusted proxy server at IP address 10.0.0.100 to the Forwarded Headers Middleware `KnownProxies` in `Startup.ConfigureServices`:

```
// using System.Net;

services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});
```

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Install Nginx

Use `apt-get` to install Nginx. The installer creates a *systemd* init script that runs Nginx as daemon on system startup. Follow the installation instructions for Ubuntu at Nginx: Official Debian/Ubuntu packages.

> **NOTE**
> If optional Nginx modules are required, building Nginx from source might be required.

Since Nginx was installed for the first time, explicitly start it by running:

```
sudo service nginx start
```

Verify a browser displays the default landing page for Nginx. The landing page is reachable at `http://<server_IP_address>/index.nginx-debian.html`.

Configure Nginx

To configure Nginx as a reverse proxy to forward requests to your ASP.NET Core app, modify */etc/nginx/sites-available/default*. Open it in a text editor, and replace the contents with the following:

```
server {
    listen        80;
    server_name   example.com *.example.com;
    location / {
        proxy_pass         http://localhost:5000;
        proxy_http_version 1.1;
        proxy_set_header   Upgrade $http_upgrade;
        proxy_set_header   Connection keep-alive;
        proxy_set_header   Host $host;
        proxy_cache_bypass $http_upgrade;
        proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }
}
```

If the app is a Blazor Server app that relies on SignalR WebSockets, see Host and deploy ASP.NET Core Blazor Server for information on how to set the `Connection` header.

When no `server_name` matches, Nginx uses the default server. If no default server is defined, the first server in the configuration file is the default server. As a best practice, add a specific default server which returns a status

code of 444 in your configuration file. A default server configuration example is:

```
server {
    listen   80 default_server;
    # listen [::]:80 default_server deferred;
    return   444;
}
```

With the preceding configuration file and default server, Nginx accepts public traffic on port 80 with host header `example.com` or `*.example.com`. Requests not matching these hosts won't get forwarded to Kestrel. Nginx forwards the matching requests to Kestrel at `http://localhost:5000`. See How nginx processes a request for more information. To change Kestrel's IP/port, see Kestrel: Endpoint configuration.

> **WARNING**
> Failure to specify a proper server_name directive exposes your app to security vulnerabilities. Subdomain wildcard binding (for example, `*.example.com`) doesn't pose this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See rfc7230 section-5.4 for more information.

Once the Nginx configuration is established, run `sudo nginx -t` to verify the syntax of the configuration files. If the configuration file test is successful, force Nginx to pick up the changes by running `sudo nginx -s reload`.

To directly run the app on the server:

1. Navigate to the app's directory.
2. Run the app: `dotnet <app_assembly.dll>`, where `app_assembly.dll` is the assembly file name of the app.

If the app runs on the server but fails to respond over the Internet, check the server's firewall and confirm that port 80 is open. If using an Azure Ubuntu VM, add a Network Security Group (NSG) rule that enables inbound port 80 traffic. There's no need to enable an outbound port 80 rule, as the outbound traffic is automatically granted when the inbound rule is enabled.

When done testing the app, shut the app down with `Ctrl+C` at the command prompt.

Monitor the app

The server is setup to forward requests made to `http://<serveraddress>:80` on to the ASP.NET Core app running on Kestrel at `http://127.0.0.1:5000`. However, Nginx isn't set up to manage the Kestrel process. *systemd* can be used to create a service file to start and monitor the underlying web app. *systemd* is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

```
sudo nano /etc/systemd/system/kestrel-helloapp.service
```

The following is an example service file for the app:

```
[Unit]
Description=Example .NET Web API App running on Ubuntu

[Service]
WorkingDirectory=/var/www/helloapp
ExecStart=/usr/bin/dotnet /var/www/helloapp/helloapp.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_PRINT_TELEMETRY_MESSAGE=false

[Install]
WantedBy=multi-user.target
```

In the preceding example, the user that manages the service is specified by the `User` option. The user (`www-data`) must exist and have proper ownership of the app's files.

Use `TimeoutStopSec` to configure the duration of time to wait for the app to shut down after it receives the initial interrupt signal. If the app doesn't shut down in this period, SIGKILL is issued to terminate the app. Provide the value as unitless seconds (for example, `150`), a time span value (for example, `2min 30s`), or `infinity` to disable the timeout. `TimeoutStopSec` defaults to the value of `DefaultTimeoutStopSec` in the manager configuration file (*systemd-system.conf*, *system.conf.d*, *systemd-user.conf*, *user.conf.d*). The default timeout for most distributions is 90 seconds.

```
# The default value is 90 seconds for most distributions.
TimeoutStopSec=90
```

Linux has a case-sensitive file system. Setting ASPNETCORE_ENVIRONMENT to "Production" results in searching for the configuration file *appsettings.Production.json*, not *appsettings.production.json*.

Some values (for example, SQL connection strings) must be escaped for the configuration providers to read the environment variables. Use the following command to generate a properly escaped value for use in the configuration file:

```
systemd-escape "<value-to-escape>"
```

Colon (`:`) separators aren't supported in environment variable names. Use a double underscore (`__`) in place of a colon. The Environment Variables configuration provider converts double-underscores into colons when environment variables are read into configuration. In the following example, the connection string key `ConnectionStrings:DefaultConnection` is set into the service definition file as `ConnectionStrings__DefaultConnection` :

Colon (`:`) separators aren't supported in environment variable names. Use a double underscore (`__`) in place of a colon. The Environment Variables configuration provider converts double-underscores into colons when environment variables are read into configuration. In the following example, the connection string key `ConnectionStrings:DefaultConnection` is set into the service definition file as `ConnectionStrings__DefaultConnection` :

```
Environment=ConnectionStrings__DefaultConnection={Connection String}
```

Save the file and enable the service.

```
sudo systemctl enable kestrel-helloapp.service
```

Start the service and verify that it's running.

```
sudo systemctl start kestrel-helloapp.service
sudo systemctl status kestrel-helloapp.service

● kestrel-helloapp.service - Example .NET Web API App running on Ubuntu
   Loaded: loaded (/etc/systemd/system/kestrel-helloapp.service; enabled)
   Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
Main PID: 9021 (dotnet)
   CGroup: /system.slice/kestrel-helloapp.service
           └─9021 /usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
```

With the reverse proxy configured and Kestrel managed through systemd, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. It's also accessible from a remote machine, barring any firewall that might be blocking. Inspecting the response headers, the `Server` header shows the ASP.NET Core app being served by Kestrel.

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

View logs

Since the web app using Kestrel is managed using `systemd`, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by `systemd`. To view the `kestrel-helloapp.service`-specific items, use the following command:

```
sudo journalctl -fu kestrel-helloapp.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago` or a combination of these can reduce the amount of entries returned.

```
sudo journalctl -fu kestrel-helloapp.service --since "2016-10-18" --until "2016-10-18 04:00"
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including authentication middleware (for example, cookie middleware) and cross-site request forgery (CSRF) protections. Even if Data Protection APIs aren't called by user code, data protection should be configured to create a persistent cryptographic key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection to persist and encrypt the key ring, see:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Long request header fields

Proxy server default settings typically limit request header fields to 4 K or 8 K depending on the platform. An app may require fields longer than the default (for example, apps that use Azure Active Directory). If longer fields are required, the proxy server's default settings require adjustment. The values to apply depend on the scenario. For more information, see your server's documentation.

- proxy_buffer_size
- proxy_buffers
- proxy_busy_buffers_size
- large_client_header_buffers

> **WARNING**
> Don't increase the default values of proxy buffers unless necessary. Increasing these values increases the risk of buffer overrun (overflow) and Denial of Service (DoS) attacks by malicious users.

Secure the app

Enable AppArmor

Linux Security Modules (LSM) is a framework that's part of the Linux kernel since Linux 2.6. LSM supports different implementations of security modules. AppArmor is a LSM that implements a Mandatory Access Control system which allows confining the program to a limited set of resources. Ensure AppArmor is enabled and properly configured.

Configure the firewall

Close off all external ports that are not in use. Uncomplicated firewall (ufw) provides a front end for `iptables` by providing a CLI for configuring the firewall.

> **WARNING**
> A firewall will prevent access to the whole system if not configured correctly. Failure to specify the correct SSH port will effectively lock you out of the system if you are using SSH to connect to it. The default port is 22. For more information, see the introduction to ufw and the manual.

Install `ufw` and configure it to allow traffic on any ports needed.

```
sudo apt-get install ufw

sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

sudo ufw enable
```

Secure Nginx

Change the Nginx response name

Edit *src/http/ngx_http_header_filter_module.c*.

```
static char ngx_http_server_string[] = "Server: Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Web Server" CRLF;
```

Configure options

Configure the server with additional required modules. Consider using a web app firewall, such as ModSecurity, to harden the app.

HTTPS configuration

Configure the app for secure (HTTPS) local connections

The dotnet run command uses the app's *Properties/launchSettings.json* file, which configures the app to listen on the URLs provided by the `applicationUrl` property (for example, `https://localhost:5001;http://localhost:5000`).

Configure the app to use a certificate in development for the `dotnet run` command or development environment (F5 or Ctrl+F5 in Visual Studio Code) using one of the following approaches:

- Replace the default certificate from configuration (*Recommended*)
- KestrelServerOptions.ConfigureHttpsDefaults

Configure the reverse proxy for secure (HTTPS) client connections

- Configure the server to listen to HTTPS traffic on port `443` by specifying a valid certificate issued by a trusted Certificate Authority (CA).
- Harden the security by employing some of the practices depicted in the following */etc/nginx/nginx.conf* file. Examples include choosing a stronger cipher and redirecting all traffic over HTTP to HTTPS.
- Adding an `HTTP Strict-Transport-Security` (HSTS) header ensures all subsequent requests made by the client are over HTTPS.
- If HTTPS will be disabled in the future, use one of the following approaches:
  - Don't add the HSTS header.
  - Choose a short `max-age` value.

Add the */etc/nginx/proxy.conf* configuration file:

```
proxy_redirect          off;
proxy_set_header        Host $host;
proxy_set_header        X-Real-IP $remote_addr;
proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header        X-Forwarded-Proto $scheme;
client_max_body_size    10m;
client_body_buffer_size 128k;
proxy_connect_timeout   90;
proxy_send_timeout      90;
proxy_read_timeout      90;
proxy_buffers           32 4k;
```

Edit the */etc/nginx/nginx.conf* configuration file. The example contains both `http` and `server` sections in one configuration file.

```
http {
    include        /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens  off;

    sendfile on;
    keepalive_timeout   29; # Adjust to the lowest possible value that makes sense for your use case.
    client_body_timeout 10; client_header_timeout 10; send_timeout 10;

    upstream helloapp{
        server localhost:5000;
    }

    server {
        listen     *:80;
        add_header Strict-Transport-Security max-age=15768000;
        return     301 https://$host$request_uri;
    }

    server {
        listen                    *:443 ssl;
        server_name               example.com;
        ssl_certificate           /etc/ssl/certs/testCert.crt;
        ssl_certificate_key       /etc/ssl/certs/testCert.key;
        ssl_protocols             TLSv1.1 TLSv1.2;
        ssl_prefer_server_ciphers on;
        ssl_ciphers               "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
        ssl_ecdh_curve            secp384r1;
        ssl_session_cache         shared:SSL:10m;
        ssl_session_tickets       off;
        ssl_stapling              on; #ensure your cert is capable
        ssl_stapling_verify       on; #ensure your cert is capable

        add_header Strict-Transport-Security "max-age=63072000; includeSubdomains; preload";
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        #Redirects all traffic
        location / {
            proxy_pass http://helloapp;
            limit_req  zone=one burst=10 nodelay;
        }
    }
}
```

> **NOTE**
>
> Blazor WebAssembly apps require a larger `burst` parameter value to accommodate the larger number of requests made by an app. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

Secure Nginx from clickjacking

Clickjacking, also known as a *UI redress attack*, is a malicious attack where a website visitor is tricked into clicking a link or button on a different page than they're currently visiting. Use `X-FRAME-OPTIONS` to secure the site.

To mitigate clickjacking attacks:

1. Edit the *nginx.conf* file:

```
sudo nano /etc/nginx/nginx.conf
```

Add the line `add_header X-Frame-Options "SAMEORIGIN";`.

2. Save the file.
3. Restart Nginx.

MIME-type sniffing

This header prevents most browsers from MIME-sniffing a response away from the declared content type, as the header instructs the browser not to override the response content type. With the `nosniff` option, if the server says the content is "text/html", the browser renders it as "text/html".

Edit the *nginx.conf* file:

```
sudo nano /etc/nginx/nginx.conf
```

Add the line `add_header X-Content-Type-Options "nosniff";` and save the file, then restart Nginx.

Additional Nginx suggestions

After upgrading the shared framework on the server, restart the ASP.NET Core apps hosted by the server.

Additional resources

- Prerequisites for .NET Core on Linux
- Nginx: Binary Releases: Official Debian/Ubuntu packages
- Troubleshoot and debug ASP.NET Core projects
- Configure ASP.NET Core to work with proxy servers and load balancers
- NGINX: Using the Forwarded header

Host ASP.NET Core on Linux with Apache

9/22/2020 • 13 minutes to read • Edit Online

By Shayne Boyer

Using this guide, learn how to set up Apache as a reverse proxy server on CentOS 7 to redirect HTTP traffic to an ASP.NET Core web app running on Kestrel server. The mod_proxy extension and related modules create the server's reverse proxy.

Prerequisites

- Server running CentOS 7 with a standard user account with sudo privilege.
- Install the .NET Core runtime on the server.
  1. Visit the Download .NET Core page.
  2. Select the latest non-preview .NET Core version.
  3. Download the latest non-preview runtime in the table under **Run apps - Runtime**.
  4. Select the Linux **Package manager instructions** link and follow the CentOS instructions.
- An existing ASP.NET Core app.

At any point in the future after upgrading the shared framework, restart the ASP.NET Core apps hosted by the server.

Publish and copy over the app

Configure the app for a framework-dependent deployment.

If the app is run locally and isn't configured to make secure connections (HTTPS), adopt either of the following approaches:

- Configure the app to handle secure local connections. For more information, see the HTTPS configuration section.
- Remove `https://localhost:5001` (if present) from the `applicationUrl` property in the *Properties/launchSettings.json* file.

Run dotnet publish from the development environment to package an app into a directory (for example, *bin/Release/<target_framework_moniker>/publish*) that can run on the server:

```
dotnet publish --configuration Release
```

The app can also be published as a self-contained deployment if you prefer not to maintain the .NET Core runtime on the server.

Copy the ASP.NET Core app to the server using a tool that integrates into the organization's workflow (for example, SCP, SFTP). It's common to locate web apps under the *var* directory (for example, *var/www/helloapp*).

> **NOTE**
> Under a production deployment scenario, a continuous integration workflow does the work of publishing the app and copying the assets to the server.

Configure a proxy server

A reverse proxy is a common setup for serving dynamic web apps. The reverse proxy terminates the HTTP request and forwards it to the ASP.NET app.

A proxy server is one which forwards client requests to another server instead of fulfilling requests itself. A reverse proxy forwards to a fixed destination, typically on behalf of arbitrary clients. In this guide, Apache is configured as the reverse proxy running on the same server that Kestrel is serving the ASP.NET Core app.

Because requests are forwarded by reverse proxy, use the Forwarded Headers Middleware from the Microsoft.AspNetCore.HttpOverrides package. The middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header, so that redirect URIs and other security policies work correctly.

Any component that depends on the scheme, such as authentication, link generation, redirects, and geolocation, must be placed after invoking the Forwarded Headers Middleware.

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Invoke the UseForwardedHeaders method at the top of `Startup.Configure` before calling other middleware. Configure the middleware to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers:

```
// using Microsoft.AspNetCore.HttpOverrides;

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();
```

If no ForwardedHeadersOptions are specified to the middleware, the default headers to forward are `None`.

Proxies running on loopback addresses (127.0.0.0/8, [::1]), including the standard localhost address (127.0.0.1), are trusted by default. If other trusted proxies or networks within the organization handle requests between the Internet and the web server, add them to the list of KnownProxies or KnownNetworks with ForwardedHeadersOptions. The following example adds a trusted proxy server at IP address 10.0.0.100 to the Forwarded Headers Middleware `KnownProxies` in `Startup.ConfigureServices`:

```
// using System.Net;

services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});
```

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Install Apache

Update CentOS packages to their latest stable versions:

```
sudo yum update -y
```

Install the Apache web server on CentOS with a single `yum` command:

```
sudo yum -y install httpd mod_ssl
```

Sample output after running the command:

```
Downloading packages:
httpd-2.4.6-40.el7.centos.4.x86_64.rpm               | 2.7 MB  00:00:01
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
Installing : httpd-2.4.6-40.el7.centos.4.x86_64      1/1
Verifying  : httpd-2.4.6-40.el7.centos.4.x86_64      1/1

Installed:
httpd.x86_64 0:2.4.6-40.el7.centos.4

Complete!
```

> **NOTE**
> In this example, the output reflects httpd.86_64 since the CentOS 7 version is 64 bit. To verify where Apache is installed, run `whereis httpd` from a command prompt.

Configure Apache

Configuration files for Apache are located within the `/etc/httpd/conf.d/` directory. Any file with the *.conf* extension is processed in alphabetical order in addition to the module configuration files in `/etc/httpd/conf.modules.d/`, which contains any configuration files necessary to load modules.

Create a configuration file, named *helloapp.conf*, for the app:

```
<VirtualHost *:*>
    RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
</VirtualHost>

<VirtualHost *:80>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ServerName www.example.com
    ServerAlias *.example.com
    ErrorLog ${APACHE_LOG_DIR}helloapp-error.log
    CustomLog ${APACHE_LOG_DIR}helloapp-access.log common
</VirtualHost>
```

The `VirtualHost` block can appear multiple times, in one or more files on a server. In the preceding configuration file, Apache accepts public traffic on port 80. The domain `www.example.com` is being served, and the `*.example.com` alias resolves to the same website. See Name-based virtual host support for more information. Requests are proxied at the root to port 5000 of the server at 127.0.0.1. For bi-directional communication, `ProxyPass` and `ProxyPassReverse` are required. To change Kestrel's IP/port, see Kestrel: Endpoint configuration.

> **WARNING**
> Failure to specify a proper ServerName directive in the **VirtualHost** block exposes your app to security vulnerabilities. Subdomain wildcard binding (for example, `*.example.com`) doesn't pose this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See rfc7230 section-5.4 for more information.

Logging can be configured per `VirtualHost` using `ErrorLog` and `CustomLog` directives. `ErrorLog` is the location where the server logs errors, and `CustomLog` sets the filename and format of log file. In this case, this is where request information is logged. There's one line for each request.

Save the file and test the configuration. If everything passes, the response should be `Syntax [OK]`.

```
sudo service httpd configtest
```

Restart Apache:

```
sudo systemctl restart httpd
sudo systemctl enable httpd
```

Monitor the app

Apache is now setup to forward requests made to `http://localhost:80` to the ASP.NET Core app running on Kestrel at `http://127.0.0.1:5000`. However, Apache isn't set up to manage the Kestrel process. Use *systemd* and create a service file to start and monitor the underlying web app. *systemd* is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

```
sudo nano /etc/systemd/system/kestrel-helloapp.service
```

An example service file for the app:

```
[Unit]
Description=Example .NET Web API App running on CentOS 7

[Service]
WorkingDirectory=/var/www/helloapp
ExecStart=/usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=apache
Environment=ASPNETCORE_ENVIRONMENT=Production

[Install]
WantedBy=multi-user.target
```

In the preceding example, the user that manages the service is specified by the `User` option. The user (`apache`) must exist and have proper ownership of the app's files.

Use `TimeoutStopSec` to configure the duration of time to wait for the app to shut down after it receives the initial interrupt signal. If the app doesn't shut down in this period, SIGKILL is issued to terminate the app. Provide the value as unitless seconds (for example, `150`), a time span value (for example, `2min 30s`), or `infinity` to disable the timeout. `TimeoutStopSec` defaults to the value of `DefaultTimeoutStopSec` in the manager configuration file (*systemd-system.conf*, *system.conf.d*, *systemd-user.conf*, *user.conf.d*). The default timeout for most distributions is 90 seconds.

```
# The default value is 90 seconds for most distributions.
TimeoutStopSec=90
```

Some values (for example, SQL connection strings) must be escaped for the configuration providers to read the environment variables. Use the following command to generate a properly escaped value for use in the configuration file:

```
systemd-escape "<value-to-escape>"
```

Colon (`:`) separators aren't supported in environment variable names. Use a double underscore (`__`) in place of a colon. The Environment Variables configuration provider converts double-underscores into colons when environment variables are read into configuration. In the following example, the connection string key `ConnectionStrings:DefaultConnection` is set into the service definition file as `ConnectionStrings__DefaultConnection`:

Colon (`:`) separators aren't supported in environment variable names. Use a double underscore (`__`) in place of a colon. The Environment Variables configuration provider converts double-underscores into colons when environment variables are read into configuration. In the following example, the connection string key `ConnectionStrings:DefaultConnection` is set into the service definition file as `ConnectionStrings__DefaultConnection`:

```
Environment=ConnectionStrings__DefaultConnection={Connection String}
```

Save the file and enable the service:

```
sudo systemctl enable kestrel-helloapp.service
```

Start the service and verify that it's running:

```
sudo systemctl start kestrel-helloapp.service
sudo systemctl status kestrel-helloapp.service

● kestrel-helloapp.service - Example .NET Web API App running on CentOS 7
   Loaded: loaded (/etc/systemd/system/kestrel-helloapp.service; enabled)
   Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
Main PID: 9021 (dotnet)
   CGroup: /system.slice/kestrel-helloapp.service
           └─9021 /usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
```

With the reverse proxy configured and Kestrel managed through *systemd*, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. Inspecting the response headers, the **`Server`** header indicates that the ASP.NET Core app is served by Kestrel:

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

View logs

Since the web app using Kestrel is managed using *systemd*, events and processes are logged to a centralized

journal. However, this journal includes entries for all of the services and processes managed by *systemd*. To view the `kestrel-helloapp.service`-specific items, use the following command:

```
sudo journalctl -fu kestrel-helloapp.service
```

For time filtering, specify time options with the command. For example, use `--since today` to filter for the current day or `--until 1 hour ago` to see the previous hour's entries. For more information, see the man page for journalctl.

```
sudo journalctl -fu kestrel-helloapp.service --since "2016-10-18" --until "2016-10-18 04:00"
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including authentication middleware (for example, cookie middleware) and cross-site request forgery (CSRF) protections. Even if Data Protection APIs aren't called by user code, data protection should be configured to create a persistent cryptographic key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection to persist and encrypt the key ring, see:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Secure the app

Configure firewall

Firewalld is a dynamic daemon to manage the firewall with support for network zones. Ports and packet filtering can still be managed by iptables. *Firewalld* should be installed by default. `yum` can be used to install the package or verify it's installed.

```
sudo yum install firewalld -y
```

Use `firewalld` to open only the ports needed for the app. In this case, port 80 and 443 are used. The following commands permanently set ports 80 and 443 to open:

```
sudo firewall-cmd --add-port=80/tcp --permanent
sudo firewall-cmd --add-port=443/tcp --permanent
```

Reload the firewall settings. Check the available services and ports in the default zone. Options are available by inspecting `firewall-cmd -h`.

```
sudo firewall-cmd --reload
sudo firewall-cmd --list-all
```

```
public (default, active)
interfaces: eth0
sources:
services: dhcpv6-client
ports: 443/tcp 80/tcp
masquerade: no
forward-ports:
icmp-blocks:
rich rules:
```

HTTPS configuration

Configure the app for secure (HTTPS) local connections

The dotnet run command uses the app's *Properties/launchSettings.json* file, which configures the app to listen on the URLs provided by the `applicationUrl` property (for example, `https://localhost:5001;http://localhost:5000`).

Configure the app to use a certificate in development for the `dotnet run` command or development environment (F5 or Ctrl+F5 in Visual Studio Code) using one of the following approaches:

- Replace the default certificate from configuration (*Recommended*)
- KestrelServerOptions.ConfigureHttpsDefaults

Configure the reverse proxy for secure (HTTPS) client connections

To configure Apache for HTTPS, the *mod_ssl* module is used. When the *httpd* module was installed, the *mod_ssl* module was also installed. If it wasn't installed, use `yum` to add it to the configuration.

```
sudo yum install mod_ssl
```

To enforce HTTPS, install the `mod_rewrite` module to enable URL rewriting:

```
sudo yum install mod_rewrite
```

Modify the *helloapp.conf* file to enable URL rewriting and secure communication on port 443:

```
<VirtualHost *:*>
    RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
</VirtualHost>

<VirtualHost *:80>
    RewriteEngine On
    RewriteCond %{HTTPS} !=on
    RewriteRule ^/?(.*) https://%{SERVER_NAME}/$1 [R,L]
</VirtualHost>

<VirtualHost *:443>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ErrorLog /var/log/httpd/helloapp-error.log
    CustomLog /var/log/httpd/helloapp-access.log common
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
</VirtualHost>
```

> **NOTE**
>
> This example is using a locally-generated certificate. **SSLCertificateFile** should be the primary certificate file for the domain name. **SSLCertificateKeyFile** should be the key file generated when CSR is created. **SSLCertificateChainFile** should be the intermediate certificate file (if any) that was supplied by the certificate authority.

Save the file and test the configuration:

```
sudo service httpd configtest
```

Restart Apache:

```
sudo systemctl restart httpd
```

Additional Apache suggestions

Restart apps with shared framework updates

After upgrading the shared framework on the server, restart the ASP.NET Core apps hosted by the server.

Additional headers

In order to secure against malicious attacks, there are a few headers that should either be modified or added. Ensure that the `mod_headers` module is installed:

```
sudo yum install mod_headers
```

Secure Apache from clickjacking attacks

Clickjacking, also known as a *UI redress attack*, is a malicious attack where a website visitor is tricked into clicking a link or button on a different page than they're currently visiting. Use `X-FRAME-OPTIONS` to secure the site.

To mitigate clickjacking attacks:

1. Edit the *httpd.conf* file:

   ```
   sudo nano /etc/httpd/conf/httpd.conf
   ```

   Add the line `Header append X-FRAME-OPTIONS "SAMEORIGIN"`.

2. Save the file.

3. Restart Apache.

MIME-type sniffing

The `X-Content-Type-Options` header prevents Internet Explorer from *MIME-sniffing* (determining a file's `Content-Type` from the file's content). If the server sets the `Content-Type` header to `text/html` with the `nosniff` option set, Internet Explorer renders the content as `text/html` regardless of the file's content.

Edit the *httpd.conf* file:

```
sudo nano /etc/httpd/conf/httpd.conf
```

Add the line `Header set X-Content-Type-Options "nosniff"`. Save the file. Restart Apache.

Load Balancing

This example shows how to setup and configure Apache on CentOS 7 and Kestrel on the same instance machine. In order to not have a single point of failure; using *mod_proxy_balancer* and modifying the **VirtualHost** would allow for managing multiple instances of the web apps behind the Apache proxy server.

```
sudo yum install mod_proxy_balancer
```

In the configuration file shown below, an additional instance of the `helloapp` is set up to run on port 5001. The *Proxy* section is set with a balancer configuration with two members to load balance *byrequests*.

```
<VirtualHost *:*>
    RequestHeader set "X-Forwarded-Proto" expr=%{REQUEST_SCHEME}
</VirtualHost>

<VirtualHost *:80>
    RewriteEngine On
    RewriteCond %{HTTPS} !=on
    RewriteRule ^/?(.*) https://%{SERVER_NAME}/$1 [R,L]
</VirtualHost>

<VirtualHost *:443>
    ProxyPass / balancer://mycluster/

    ProxyPassReverse / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5001/

    <Proxy balancer://mycluster>
        BalancerMember http://127.0.0.1:5000
        BalancerMember http://127.0.0.1:5001
        ProxySet lbmethod=byrequests
    </Proxy>

    <Location />
        SetHandler balancer
    </Location>
    ErrorLog /var/log/httpd/helloapp-error.log
    CustomLog /var/log/httpd/helloapp-access.log common
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:!RC4+RSA:+HIGH:+MEDIUM:!LOW:!RC4
    SSLCertificateFile /etc/pki/tls/certs/localhost.crt
    SSLCertificateKeyFile /etc/pki/tls/private/localhost.key
</VirtualHost>
```

Rate Limits

Using *mod_ratelimit*, which is included in the *httpd* module, the bandwidth of clients can be limited:

```
sudo nano /etc/httpd/conf.d/ratelimit.conf
```

The example file limits bandwidth as 600 KB/sec under the root location:

```
<IfModule mod_ratelimit.c>
    <Location />
        SetOutputFilter RATE_LIMIT
        SetEnv rate-limit 600
    </Location>
</IfModule>
```

Long request header fields

Proxy server default settings typically limit request header fields to 8,190 bytes. An app may require fields longer than the default (for example, apps that use Azure Active Directory). If longer fields are required, the proxy server's LimitRequestFieldSize directive requires adjustment. The value to apply depends on the scenario. For more information, see your server's documentation.

> **WARNING**
>
> Don't increase the default value of `LimitRequestFieldSize` unless necessary. Increasing the value increases the risk of buffer overrun (overflow) and Denial of Service (DoS) attacks by malicious users.

Additional resources

- Prerequisites for .NET Core on Linux
- Troubleshoot and debug ASP.NET Core projects
- Configure ASP.NET Core to work with proxy servers and load balancers

Host ASP.NET Core in Docker containers

9/22/2020 • 2 minutes to read • Edit Online

The following articles are available for learning about hosting ASP.NET Core apps in Docker:

Introduction to Containers and Docker

See how containerization is an approach to software development in which an application or service, its dependencies, and its configuration are packaged together as a container image. The image can be tested and then deployed to a host.

What is Docker

Discover how Docker is an open-source project for automating the deployment of apps as portable, self-sufficient containers that can run on the cloud or on-premises.

Docker Terminology

Learn terms and definitions for Docker technology.

Docker containers, images, and registries

Find out how Docker container images are stored in an image registry for consistent deployment across environments.

Docker images for ASP.NET Core Learn how to build and dockerize an ASP.NET Core app. Explore Docker images maintained by Microsoft and examine use cases.

Visual Studio Container Tools

Discover how Visual Studio supports building, debugging, and running ASP.NET Core apps targeting either .NET Framework or .NET Core on Docker for Windows. Both Windows and Linux containers are supported.

Publish to Azure Container Registry

Find out how to use the Visual Studio Container Tools extension to deploy an ASP.NET Core app to a Docker host on Azure using PowerShell.

Configure ASP.NET Core to work with proxy servers and load balancers

Additional configuration might be required for apps hosted behind proxy servers and load balancers. Passing requests through a proxy often obscures information about the original request, such as the scheme and client IP. It might be necessary to forwarded some information about the request manually to the app.

GC using Docker and small containers Discusses GC selection with small containers.

Docker images for ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

This tutorial shows how to run an ASP.NET Core app in Docker containers.

In this tutorial, you:

- Learn about Microsoft .NET Core Docker images
- Download an ASP.NET Core sample app
- Run the sample app locally
- Run the sample app in Linux containers
- Run the sample app in Windows containers
- Build and deploy manually

ASP.NET Core Docker images

For this tutorial, you download an ASP.NET Core sample app and run it in Docker containers. The sample works with both Linux and Windows containers.

The sample Dockerfile uses the Docker multi-stage build feature to build and run in different containers. The build and run containers are created from images that are provided in Docker Hub by Microsoft:

- `dotnet/core/sdk`

  The sample uses this image for building the app. The image contains the .NET Core SDK, which includes the Command Line Tools (CLI). The image is optimized for local development, debugging, and unit testing. The tools installed for development and compilation make this a relatively large image.

- `dotnet/core/aspnet`

  The sample uses this image for running the app. The image contains the ASP.NET Core runtime and libraries and is optimized for running apps in production. Designed for speed of deployment and app startup, the image is relatively small, so network performance from Docker Registry to Docker host is optimized. Only the binaries and content needed to run an app are copied to the container. The contents are ready to run, enabling the fastest time from `Docker run` to app startup. Dynamic code compilation isn't needed in the Docker model.

Prerequisites

- .NET Core 2.2 SDK
- .NET Core SDK 3.0
- Docker client 18.03 or later
  - Linux distributions
    - CentOS
    - Debian
    - Fedora
    - Ubuntu
  - macOS
  - Windows

- Git

Download the sample app

- Download the sample by cloning the .NET Core Docker repository:

```
git clone https://github.com/dotnet/dotnet-docker
```

Run the app locally

- Navigate to the project folder at *dotnet-docker/samples/aspnetapp/aspnetapp*.
- Run the following command to build and run the app locally:

```
dotnet run
```

- Go to `http://localhost:5000` in a browser to test the app.
- Press Ctrl+C at the command prompt to stop the app.

Run in a Linux container

- In the Docker client, switch to Linux containers.
- Navigate to the Dockerfile folder at *dotnet-docker/samples/aspnetapp*.
- Run the following commands to build and run the sample in Docker:

```
docker build -t aspnetapp .
docker run -it --rm -p 5000:80 --name aspnetcore_sample aspnetapp
```

  The `build` command arguments:

  - Name the image aspnetapp.
  - Look for the Dockerfile in the current folder (the period at the end).

  The run command arguments:

  - Allocate a pseudo-TTY and keep it open even if not attached. (Same effect as `--interactive --tty`.)
  - Automatically remove the container when it exits.
  - Map port 5000 on the local machine to port 80 in the container.
  - Name the container aspnetcore_sample.
  - Specify the aspnetapp image.
- Go to `http://localhost:5000` in a browser to test the app.

Run in a Windows container

- In the Docker client, switch to Windows containers.

Navigate to the docker file folder at `dotnet-docker/samples/aspnetapp`.

- Run the following commands to build and run the sample in Docker:

```
docker build -t aspnetapp .
docker run -it --rm --name aspnetcore_sample aspnetapp
```

- For Windows containers, you need the IP address of the container (browsing to `http://localhost:5000` won't work):
  - Open up another command prompt.
  - Run `docker ps` to see the running containers. Verify that the "aspnetcore_sample" container is there.
  - Run `docker exec aspnetcore_sample ipconfig` to display the IP address of the container. The output from the command looks like this example:

    ```
    Ethernet adapter Ethernet:

       Connection-specific DNS Suffix  . : contoso.com
       Link-local IPv6 Address . . . . . : fe80::1967:6598:124:cfa3%4
       IPv4 Address. . . . . . . . . . . : 172.29.245.43
       Subnet Mask . . . . . . . . . . . : 255.255.240.0
       Default Gateway . . . . . . . . . : 172.29.240.1
    ```

- Copy the container IPv4 address (for example, 172.29.245.43) and paste into the browser address bar to test the app.

Build and deploy manually

In some scenarios, you might want to deploy an app to a container by copying to it the application files that are needed at run time. This section shows how to deploy manually.

- Navigate to the project folder at *dotnet-docker/samples/aspnetapp/aspnetapp*.
- Run the dotnet publish command:

  ```
  dotnet publish -c Release -o published
  ```

  The command arguments:

  - Build the application in release mode (the default is debug mode).
  - Create the files in the *published* folder.
- Run the application.
  - Windows:

    ```
    dotnet published\aspnetapp.dll
    ```

  - Linux:

    ```
    dotnet published/aspnetapp.dll
    ```

- Browse to `http://localhost:5000` to see the home page.

To use the manually published application within a Docker container, create a new Dockerfile and use the `docker build .` command to build the container.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS runtime
WORKDIR /app
COPY published/aspnetapp.dll ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

The Dockerfile

Here's the *Dockerfile* used by the `docker build` command you ran earlier. It uses `dotnet publish` the same way you did in this section to build and deploy.

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
WORKDIR /app

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore

# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /app/aspnetapp
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS runtime
WORKDIR /app
COPY --from=build /app/aspnetapp/out ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.0 AS runtime
WORKDIR /app
COPY published/aspnetapp.dll ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

The Dockerfile

Here's the *Dockerfile* used by the `docker build` command you ran earlier. It uses `dotnet publish` the same way you did in this section to build and deploy.

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.0 AS build
WORKDIR /app

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore

# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /app/aspnetapp
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/core/aspnet:3.0 AS runtime
WORKDIR /app
COPY --from=build /app/aspnetapp/out ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

As noted in the preceding Dockerfile, the `*.csproj` files are copied and restored as distinct *layers*. When the `docker build` command builds an image, it uses a built-in cache. If the `*.csproj` files haven't changed since the `docker build` command last ran, the `dotnet restore` command doesn't need to run again. Instead, the built-in cache for the corresponding `dotnet restore` layer is reused. For more information, see Best practices for writing

Dockerfiles.

Additional resources

- Docker build command
- Docker run command
- ASP.NET Core Docker sample (The one used in this tutorial.)
- Configure ASP.NET Core to work with proxy servers and load balancers
- Working with Visual Studio Docker Tools
- Debugging with Visual Studio Code
- GC using Docker and small containers

Next steps

The Git repository that contains the sample app also includes documentation. For an overview of the resources available in the repository, see the README file. In particular, learn how to implement HTTPS:

Developing ASP.NET Core Applications with Docker over HTTPS

Visual Studio Container Tools with ASP.NET Core

9/22/2020 • 9 minutes to read • Edit Online

Visual Studio 2017 and later versions support building, debugging, and running containerized ASP.NET Core apps targeting .NET Core. Both Windows and Linux containers are supported.

View or download sample code (how to download)

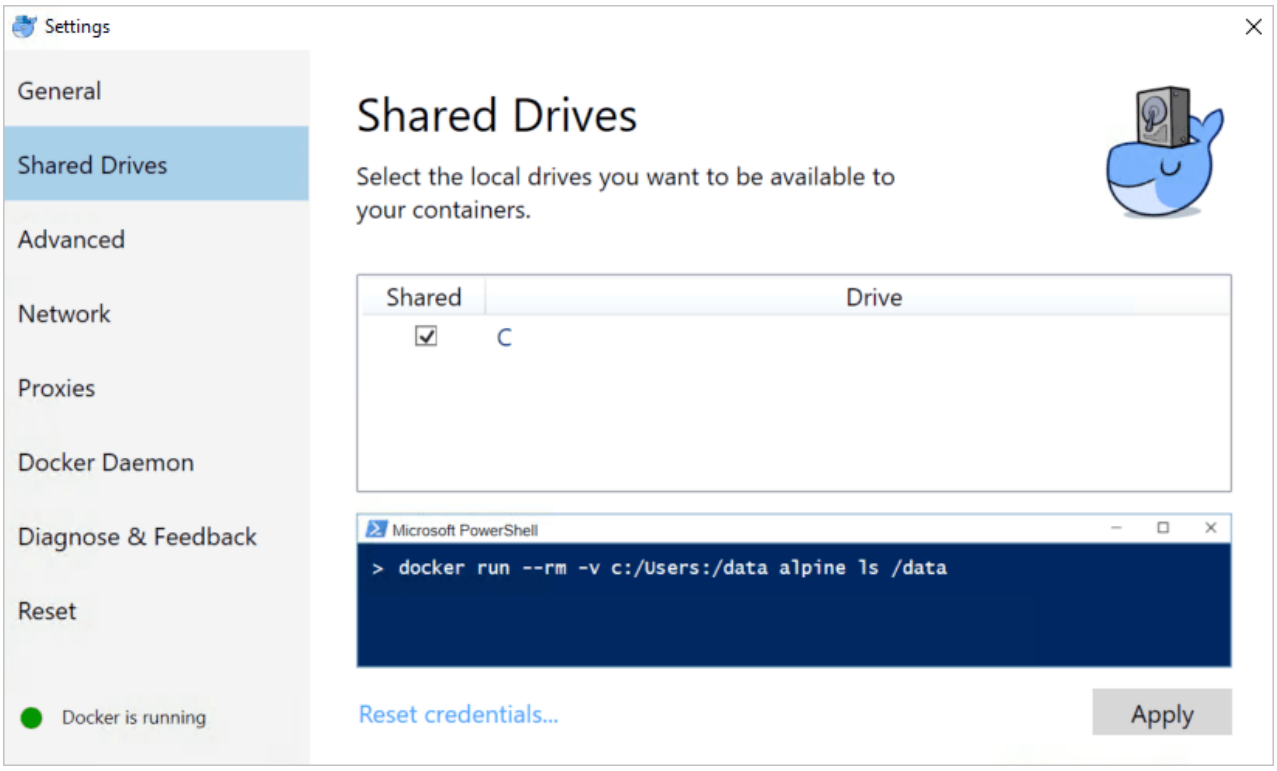
Prerequisites

- Docker for Windows
- Visual Studio 2019 with the **.NET Core cross-platform development** workload

Installation and setup

For Docker installation, first review the information at Docker for Windows: What to know before you install. Next, install Docker For Windows.

Shared Drives in Docker for Windows must be configured to support volume mapping and debugging. Right-click the System Tray's Docker icon, select **Settings**, and select **Shared Drives**. Select the drive where Docker stores files. Click **Apply**.



TIP

Visual Studio 2017 versions 15.6 and later prompt when **Shared Drives** aren't configured.

Add a project to a Docker container

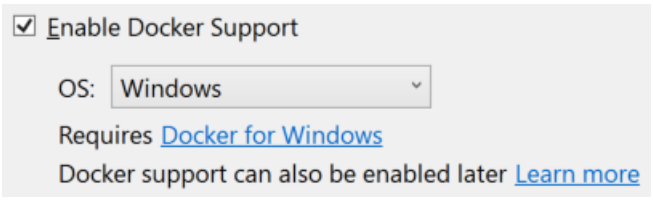
To containerize an ASP.NET Core project, the project must target .NET Core. Both Linux and Windows containers are

supported.

When adding Docker support to a project, choose either a Windows or a Linux container. The Docker host must be running the same container type. To change the container type in the running Docker instance, right-click the System Tray's Docker icon and choose **Switch to Windows containers...** or **Switch to Linux containers...**.

New app

When creating a new app with the **ASP.NET Core Web Application** project templates, select the **Enable Docker Support** check box:



If the target framework is .NET Core, the **OS** drop-down allows for the selection of a container type.

Existing app

For ASP.NET Core projects targeting .NET Core, there are two options for adding Docker support via the tooling. Open the project in Visual Studio, and choose one of the following options:

- Select **Docker Support** from the **Project** menu.
- Right-click the project in **Solution Explorer** and select **Add** > **Docker Support**.

The Visual Studio Container Tools don't support adding Docker to an existing ASP.NET Core project targeting .NET Framework.

Dockerfile overview

A *Dockerfile*, the recipe for creating a final Docker image, is added to the project root. Refer to Dockerfile reference for an understanding of the commands within it. This particular *Dockerfile* uses a multi-stage build with four distinct, named build stages:

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.1 AS base
WORKDIR /app
EXPOSE 59518
EXPOSE 44364

FROM mcr.microsoft.com/dotnet/core/sdk:2.1 AS build
WORKDIR /src
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore HelloDockerTools/HelloDockerTools.csproj
COPY . .
WORKDIR /src/HelloDockerTools
RUN dotnet build HelloDockerTools.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish HelloDockerTools.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
```

The preceding *Dockerfile* is based on the microsoft/dotnet image. This base image includes the ASP.NET Core runtime and NuGet packages. The packages are just-in-time (JIT) compiled to improve startup performance.

When the new project dialog's **Configure for HTTPS** check box is checked, the *Dockerfile* exposes two ports. One

port is used for HTTP traffic; the other port is used for HTTPS. If the check box isn't checked, a single port (80) is exposed for HTTP traffic.

```
FROM microsoft/aspnetcore:2.0 AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/aspnetcore-build:2.0 AS build
WORKDIR /src
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore HelloDockerTools/HelloDockerTools.csproj
COPY . .
WORKDIR /src/HelloDockerTools
RUN dotnet build HelloDockerTools.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish HelloDockerTools.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
```

The preceding *Dockerfile* is based on the microsoft/aspnetcore image. This base image includes the ASP.NET Core NuGet packages, which are just-in-time (JIT) compiled to improve startup performance.

Add container orchestrator support to an app

Visual Studio 2017 versions 15.7 or earlier support Docker Compose as the sole container orchestration solution. The Docker Compose artifacts are added via **Add** > **Docker Support**.

Visual Studio 2017 versions 15.8 or later add an orchestration solution only when instructed. Right-click the project in **Solution Explorer** and select **Add** > **Container Orchestrator Support**. The following choices are available:

- Docker Compose
- Service Fabric
- Kubernetes/Helm

Docker Compose

The Visual Studio Container Tools add a *docker-compose* project to the solution with the following files:

- *docker-compose.dcproj*: The file representing the project. Includes a `<DockerTargetOS>` element specifying the OS to be used.
- *.dockerignore*: Lists the file and directory patterns to exclude when generating a build context.
- *docker-compose.yml*: The base Docker Compose file used to define the collection of images built and run with `docker-compose build` and `docker-compose run`, respectively.
- *docker-compose.override.yml*: An optional file, read by Docker Compose, with configuration overrides for services. Visual Studio executes `docker-compose -f "docker-compose.yml" -f "docker-compose.override.yml"` to merge these files.

The *docker-compose.yml* file references the name of the image that's created when the project runs:

```
version: '3.4'

services:
  hellodockertools:
    image: ${DOCKER_REGISTRY}hellodockertools
    build:
      context: .
      dockerfile: HelloDockerTools/Dockerfile
```

In the preceding example, `image: hellodockertools` generates the image `hellodockertools:dev` when the app runs in **Debug** mode. The `hellodockertools:latest` image is generated when the app runs in **Release** mode.

Prefix the image name with the Docker Hub username (for example, `dockerhubusername/hellodockertools`) if the image is pushed to the registry. Alternatively, change the image name to include the private registry URL (for example, `privateregistry.domain.com/hellodockertools`) depending on the configuration.

If you want different behavior based on the build configuration (for example, Debug or Release), add configuration-specific *docker-compose* files. The files should be named according to the build configuration (for example, *docker-compose.vs.debug.yml* and *docker-compose.vs.release.yml*) and placed in the same location as the *docker-compose-override.yml* file.

Using the configuration-specific override files, you can specify different configuration settings (such as environment variables or entry points) for Debug and Release build configurations.

For Docker Compose to display an option to run in Visual Studio, the docker project must be the startup project.

Service Fabric

In addition to the base Prerequisites, the Service Fabric orchestration solution demands the following prerequisites:

- Microsoft Azure Service Fabric SDK version 2.6 or later
- Visual Studio's **Azure Development** workload

Service Fabric doesn't support running Linux containers in the local development cluster on Windows. If the project is already using a Linux container, Visual Studio prompts to switch to Windows containers.

The Visual Studio Container Tools do the following tasks:

- Adds a *<project_name>Application* **Service Fabric Application** project to the solution.
- Adds a *Dockerfile* and a *.dockerignore* file to the ASP.NET Core project. If a *Dockerfile* already exists in the ASP.NET Core project, it's renamed to *Dockerfile.original*. A new *Dockerfile*, similar to the following, is created:

  ```
  # See https://aka.ms/containerimagehelp for information on how to use Windows Server 1709 containers
  with Service Fabric.
  # FROM microsoft/aspnetcore:2.0-nanoserver-1709
  FROM microsoft/aspnetcore:2.0-nanoserver-sac2016
  ARG source
  WORKDIR /app
  COPY ${source:-obj/Docker/publish} .
  ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
  ```

- Adds an `<IsServiceFabricServiceProject>` element to the ASP.NET Core project's *.csproj* file:

  ```
  <IsServiceFabricServiceProject>True</IsServiceFabricServiceProject>
  ```

- Adds a *PackageRoot* folder to the ASP.NET Core project. The folder includes the service manifest and settings

for the new service.

For more information, see Deploy a .NET app in a Windows container to Azure Service Fabric.

Debug

Select **Docker** from the debug drop-down in the toolbar, and start debugging the app. The **Docker** view of the **Output** window shows the following actions taking place:

- The *2.1-aspnetcore-runtime* tag of the *microsoft/dotnet* runtime image is acquired (if not already in the cache). The image installs the ASP.NET Core and .NET Core runtimes and associated libraries. It's optimized for running ASP.NET Core apps in production.
- The `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development` within the container.
- Two dynamically assigned ports are exposed: one for HTTP and one for HTTPS. The port assigned to localhost can be queried with the `docker ps` command.
- The app is copied to the container.
- The default browser is launched with the debugger attached to the container using the dynamically assigned port.

The resulting Docker image of the app is tagged as *dev*. The image is based on the *2.1-aspnetcore-runtime* tag of the *microsoft/dotnet* base image. Run the `docker images` command in the **Package Manager Console** (PMC) window. The images on the machine are displayed:

```
REPOSITORY        TAG                   IMAGE ID      CREATED         SIZE
hellodockertools  dev                   d72ce0f1dfe7  30 seconds ago  255MB
microsoft/dotnet  2.1-aspnetcore-runtime  fcc3887985bb  6 days ago      255MB
```

- The *microsoft/aspnetcore* runtime image is acquired (if not already in the cache).
- The `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development` within the container.
- Port 80 is exposed and mapped to a dynamically assigned port for localhost. The port is determined by the Docker host and can be queried with the `docker ps` command.
- The app is copied to the container.
- The default browser is launched with the debugger attached to the container using the dynamically assigned port.

The resulting Docker image of the app is tagged as *dev*. The image is based on the *microsoft/aspnetcore* base image. Run the `docker images` command in the **Package Manager Console** (PMC) window. The images on the machine are displayed:

```
REPOSITORY            TAG  IMAGE ID      CREATED        SIZE
hellodockertools      dev  5fafe5d1ad5b  4 minutes ago  347MB
microsoft/aspnetcore  2.0  c69d39472da9  13 days ago    347MB
```

> **NOTE**
>
> The *dev* image lacks the app contents, as **Debug** configurations use volume mounting to provide the iterative experience. To push an image, use the **Release** configuration.

Run the `docker ps` command in PMC. Notice the app is running using the container:

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   21 seconds ago      Up 19 seconds
0.0.0.0:37630->80/tcp   dockercompose4642749010770307127_hellodockertools_1
```

Edit and continue

Changes to static files and Razor views are automatically updated without the need for a compilation step. Make the change, save, and refresh the browser to view the update.

Code file modifications require compilation and a restart of Kestrel within the container. After making the change, use `CTRL+F5` to perform the process and start the app within the container. The Docker container isn't rebuilt or stopped. Run the `docker ps` command in PMC. Notice the original container is still running as of 10 minutes ago:

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   10 minutes ago      Up 10 minutes
0.0.0.0:37630->80/tcp   dockercompose4642749010770307127_hellodockertools_1
```

Publish Docker images

Once the develop and debug cycle of the app is completed, the Visual Studio Container Tools assist in creating the production image of the app. Change the configuration drop-down to **Release** and build the app. The tooling acquires the compile/publish image from Docker Hub (if not already in the cache). An image is produced with the *latest* tag, which can be pushed to the private registry or Docker Hub.

Run the `docker images` command in PMC to see the list of images. Output similar to the following is displayed:

```
REPOSITORY        TAG                    IMAGE ID      CREATED             SIZE
hellodockertools  latest                 e3984a64230c  About a minute ago  258MB
hellodockertools  dev                    d72ce0f1dfe7  4 minutes ago       255MB
microsoft/dotnet  2.1-sdk                9e243db15f91  6 days ago          1.7GB
microsoft/dotnet  2.1-aspnetcore-runtime  fcc3887985bb  6 days ago          255MB
```

```
REPOSITORY                  TAG     IMAGE ID      CREATED         SIZE
hellodockertools            latest  cd28f0d4abbd  12 seconds ago  349MB
hellodockertools            dev     5fafe5d1ad5b  23 minutes ago  347MB
microsoft/aspnetcore-build  2.0     7fed40fbb647  13 days ago     2.02GB
microsoft/aspnetcore        2.0     c69d39472da9  13 days ago     347MB
```

The `microsoft/aspnetcore-build` and `microsoft/aspnetcore` images listed in the preceding output are replaced with `microsoft/dotnet` images as of .NET Core 2.1. For more information, see the Docker repositories migration announcement.

> **NOTE**
>
> The `docker images` command returns intermediary images with repository names and tags identified as *<none>* (not listed above). These unnamed images are produced by the multi-stage build *Dockerfile*. They improve the efficiency of building the final image—only the necessary layers are rebuilt when changes occur. When the intermediary images are no longer needed, delete them using the docker rmi command.

There may be an expectation for the production or release image to be smaller in size by comparison to the *dev* image. Because of the volume mapping, the debugger and app were running from the local machine and not

within the container. The *latest* image has packaged the necessary app code to run the app on a host machine. Therefore, the delta is the size of the app code.

Additional resources

- Container development with Visual Studio
- Azure Service Fabric: Prepare your development environment
- Deploy a .NET app in a Windows container to Azure Service Fabric
- Troubleshoot Visual Studio development with Docker
- Visual Studio Container Tools GitHub repository
- GC using Docker and small containers

Configure ASP.NET Core to work with proxy servers and load balancers

9/22/2020 • 26 minutes to read • Edit Online

By Chris Ross

In the recommended configuration for ASP.NET Core, the app is hosted using IIS/ASP.NET Core Module, Nginx, or Apache. Proxy servers, load balancers, and other network appliances often obscure information about the request before it reaches the app:

- When HTTPS requests are proxied over HTTP, the original scheme (HTTPS) is lost and must be forwarded in a header.
- Because an app receives a request from the proxy and not its true source on the Internet or corporate network, the originating client IP address must also be forwarded in a header.

This information may be important in request processing, for example in redirects, authentication, link generation, policy evaluation, and client geolocation.

Forwarded headers

By convention, proxies forward information in HTTP headers.

HEADER	DESCRIPTION
X-Forwarded-For	Holds information about the client that initiated the request and subsequent proxies in a chain of proxies. This parameter may contain IP addresses (and, optionally, port numbers). In a chain of proxy servers, the first parameter indicates the client where the request was first made. Subsequent proxy identifiers follow. The last proxy in the chain isn't in the list of parameters. The last proxy's IP address, and optionally a port number, are available as the remote IP address at the transport layer.
X-Forwarded-Proto	The value of the originating scheme (HTTP/HTTPS). The value may also be a list of schemes if the request has traversed multiple proxies.
X-Forwarded-Host	The original value of the Host header field. Usually, proxies don't modify the Host header. See Microsoft Security Advisory CVE-2018-0787 for information on an elevation-of-privileges vulnerability that affects systems where the proxy doesn't validate or restrict Host headers to known good values.

The Forwarded Headers Middleware, from the Microsoft.AspNetCore.HttpOverrides package, reads these headers and fills in the associated fields on HttpContext.

The middleware updates:

- HttpContext.Connection.RemoteIpAddress: Set using the `X-Forwarded-For` header value. Additional settings influence how the middleware sets `RemoteIpAddress`. For details, see the Forwarded Headers Middleware options.

- HttpContext.Request.Scheme: Set using the `X-Forwarded-Proto` header value.
- HttpContext.Request.Host: Set using the `X-Forwarded-Host` header value.

Forwarded Headers Middleware default settings can be configured. The default settings are:

- There is only *one proxy* between the app and the source of the requests.
- Only loopback addresses are configured for known proxies and known networks.
- The forwarded headers are named `X-Forwarded-For` and `X-Forwarded-Proto`.

Not all network appliances add the `X-Forwarded-For` and `X-Forwarded-Proto` headers without additional configuration. Consult your appliance manufacturer's guidance if proxied requests don't contain these headers when they reach the app. If the appliance uses different header names than `X-Forwarded-For` and `X-Forwarded-Proto`, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the appliance. For more information, see Forwarded Headers Middleware options and Configuration for a proxy that uses different header names.

IIS/IIS Express and ASP.NET Core Module

Forwarded Headers Middleware is enabled by default by IIS Integration Middleware when the app is hosted out-of-process behind IIS and the ASP.NET Core Module. Forwarded Headers Middleware is activated to run first in the middleware pipeline with a restricted configuration specific to the ASP.NET Core Module due to trust concerns with forwarded headers (for example, IP spoofing). The middleware is configured to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers and is restricted to a single localhost proxy. If additional configuration is required, see the Forwarded Headers Middleware options.

Other proxy server and load balancer scenarios

Outside of using IIS Integration when hosting out-of-process, Forwarded Headers Middleware isn't enabled by default. Forwarded Headers Middleware must be enabled for an app to process forwarded headers with UseForwardedHeaders. After enabling the middleware if no ForwardedHeadersOptions are specified to the middleware, the default ForwardedHeadersOptions.ForwardedHeaders are ForwardedHeaders.None.

Configure the middleware with ForwardedHeadersOptions to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers in `Startup.ConfigureServices`.

Forwarded Headers Middleware order

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. Forwarded Headers Middleware can run after diagnostics and error handling, but it must be run before calling `UseHsts`:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.Configure<ForwardedHeadersOptions>(options =>
        {
            options.ForwardedHeaders =
                ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
        });
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseForwardedHeaders();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseForwardedHeaders();
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

Alternatively, call `UseForwardedHeaders` before diagnostics:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseForwardedHeaders();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

NOTE

If no ForwardedHeadersOptions are specified in `Startup.ConfigureServices` or directly to the extension method with UseForwardedHeaders, the default headers to forward are ForwardedHeaders.None. The ForwardedHeaders property must be configured with the headers to forward.

Nginx configuration

To forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers, see Host ASP.NET Core on Linux with Nginx. For more information, see NGINX: Using the Forwarded header.

Apache configuration

`X-Forwarded-For` is added automatically (see Apache Module mod_proxy: Reverse Proxy Request Headers). For information on how to forward the `X-Forwarded-Proto` header, see Host ASP.NET Core on Linux with Apache.

Forwarded Headers Middleware options

ForwardedHeadersOptions control the behavior of the Forwarded Headers Middleware. The following example changes the default values:

- Limit the number of entries in the forwarded headers to `2`.
- Add a known proxy address of `127.0.10.1`.
- Change the forwarded header name from the default `X-Forwarded-For` to `X-Forwarded-For-My-Custom-Header-Name`.

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardLimit = 2;
    options.KnownProxies.Add(IPAddress.Parse("127.0.10.1"));
    options.ForwardedForHeaderName = "X-Forwarded-For-My-Custom-Header-Name";
});
```

OPTION	DESCRIPTION
AllowedHosts	Restricts hosts by the `X-Forwarded-Host` header to the values provided. • Values are compared using ordinal-ignore-case. • Port numbers must be excluded. • If the list is empty, all hosts are allowed. • A top-level wildcard `*` allows all non-empty hosts. • Subdomain wildcards are permitted but don't match the root domain. For example, `*.contoso.com` matches the subdomain `foo.contoso.com` but not the root domain `contoso.com`. • Unicode host names are allowed but are converted to Punycode for matching. • IPv6 addresses must include bounding brackets and be in conventional form (for example, `[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]`). IPv6 addresses aren't special-cased to check for logical equality between different formats, and no canonicalization is performed. • Failure to restrict the allowed hosts may allow an attacker to spoof links generated by the service. The default value is an empty `IList<string>`.
ForwardedForHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedForHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-For` header but uses some other header to forward the information. The default is `X-Forwarded-For`.
ForwardedHeaders	Identifies which forwarders should be processed. See the ForwardedHeaders Enum for the list of fields that apply. Typical values assigned to this property are `ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto`. The default value is ForwardedHeaders.None.

OPTION	DESCRIPTION
ForwardedHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedHostHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Host` header but uses some other header to forward the information. The default is `X-Forwarded-Host`.
ForwardedProtoHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedProtoHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Proto` header but uses some other header to forward the information. The default is `X-Forwarded-Proto`.
ForwardLimit	Limits the number of entries in the headers that are processed. Set to `null` to disable the limit, but this should only be done if `KnownProxies` or `KnownNetworks` are configured. Setting a non-`null` value is a precaution (but not a guarantee) to guard against misconfigured proxies and malicious requests arriving from side-channels on the network. Forwarded Headers Middleware processes headers in reverse order from right to left. If the default value (`1`) is used, only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased. The default is `1`.
KnownNetworks	Address ranges of known networks to accept forwarded headers from. Provide IP ranges using Classless Interdomain Routing (CIDR) notation. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. For more information, see the Configuration for an IPv4 address represented as an IPv6 address section. The default is an `IList`<IPNetwork> containing a single entry for `IPAddress.Loopback`.

OPTION	DESCRIPTION
KnownProxies	Addresses of known proxies to accept forwarded headers from. Use `KnownProxies` to specify exact IP address matches. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. For more information, see the Configuration for an IPv4 address represented as an IPv6 address section. The default is an `IList`<IPAddress> containing a single entry for `IPAddress.IPv6Loopback`.
OriginalForHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalForHeaderName. The default is `X-Original-For`.
OriginalHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalHostHeaderName. The default is `X-Original-Host`.
OriginalProtoHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalProtoHeaderName. The default is `X-Original-Proto`.
RequireHeaderSymmetry	Require the number of header values to be in sync between the ForwardedHeadersOptions.ForwardedHeaders being processed. The default in ASP.NET Core 1.x is `true`. The default in ASP.NET Core 2.0 or later is `false`.

Scenarios and use cases

When it isn't possible to add forwarded headers and all requests are secure

In some cases, it might not be possible to add forwarded headers to the requests proxied to the app. If the proxy is enforcing that all public external requests are HTTPS, the scheme can be manually set in `Startup.Configure` before using any type of middleware:

```
app.Use((context, next) =>
{
    context.Request.Scheme = "https";
    return next();
});
```

This code can be disabled with an environment variable or other configuration setting in a development or staging environment.

Deal with path base and proxies that change the request path

Some proxies pass the path intact but with an app base path that should be removed so that routing works properly. UsePathBaseExtensions.UsePathBase middleware splits the path into HttpRequest.Path and the app base path into HttpRequest.PathBase.

If `/foo` is the app base path for a proxy path passed as `/foo/api/1`, the middleware sets `Request.PathBase` to `/foo` and `Request.Path` to `/api/1` with the following command:

```
app.UsePathBase("/foo");
```

The original path and path base are reapplied when the middleware is called again in reverse. For more information on middleware order processing, see ASP.NET Core Middleware.

If the proxy trims the path (for example, forwarding `/foo/api/1` to `/api/1`), fix redirects and links by setting the request's PathBase property:

```
app.Use((context, next) =>
{
    context.Request.PathBase = new PathString("/foo");
    return next();
});
```

If the proxy is adding path data, discard part of the path to fix redirects and links by using StartsWithSegments and assigning to the Path property:

```
app.Use((context, next) =>
{
    if (context.Request.Path.StartsWithSegments("/foo", out var remainder))
    {
        context.Request.Path = remainder;
    }

    return next();
});
```

Configuration for a proxy that uses different header names

If the proxy doesn't use headers named `X-Forwarded-For` and `X-Forwarded-Proto` to forward the proxy address/port and originating scheme information, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the proxy:

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedForHeaderName = "Header_Name_Used_By_Proxy_For_X-Forwarded-For_Header";
    options.ForwardedProtoHeaderName = "Header_Name_Used_By_Proxy_For_X-Forwarded-Proto_Header";
});
```

Configuration for an IPv4 address represented as an IPv6 address

If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1` or `::ffff:a00:1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress.

In the following example, a network address that supplies forwarded headers is added to the

`KnownNetworks` list in IPv6 format.

IPv4 address: `10.11.12.1/8`

Converted IPv6 address: `::ffff:10.11.12.1`
Converted prefix length: 104

You can also supply the address in hexadecimal format (`10.11.12.1` represented in IPv6 as `::ffff:0a0b:0c01`). When converting an IPv4 address to IPv6, add 96 to the CIDR Prefix Length (`8` in the example) to account for the additional `::ffff:` IPv6 prefix (8 + 96 = 104).

```
// To access IPNetwork and IPAddress, add the following namespaces:
// using System.Net;
// using Microsoft.AspNetCore.HttpOverrides;
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    options.KnownNetworks.Add(new IPNetwork(
        IPAddress.Parse("::ffff:10.11.12.1"), 104));
});
```

Forward the scheme for Linux and non-IIS reverse proxies

Apps that call UseHttpsRedirection and UseHsts put a site into an infinite loop if deployed to an Azure Linux App Service, Azure Linux virtual machine (VM), or behind any other reverse proxy besides IIS. TLS is terminated by the reverse proxy, and Kestrel isn't made aware of the correct request scheme. OAuth and OIDC also fail in this configuration because they generate incorrect redirects. UseIISIntegration adds and configures Forwarded Headers Middleware when running behind IIS, but there's no matching automatic configuration for Linux (Apache or Nginx integration).

To forward the scheme from the proxy in non-IIS scenarios, add and configure Forwarded Headers Middleware. In `Startup.ConfigureServices`, use the following code:

```
// using Microsoft.AspNetCore.HttpOverrides;

if (string.Equals(
    Environment.GetEnvironmentVariable("ASPNETCORE_FORWARDEDHEADERS_ENABLED"),
    "true", StringComparison.OrdinalIgnoreCase))
{
    services.Configure<ForwardedHeadersOptions>(options =>
    {
        options.ForwardedHeaders = ForwardedHeaders.XForwardedFor |
            ForwardedHeaders.XForwardedProto;
        // Only loopback proxies are allowed by default.
        // Clear that restriction because forwarders are enabled by explicit
        // configuration.
        options.KnownNetworks.Clear();
        options.KnownProxies.Clear();
    });
}
```

Certificate forwarding

Azure

To configure Azure App Service for certificate forwarding, see Configure TLS mutual authentication for Azure App Service. The following guidance pertains to configuring the ASP.NET Core app.

In `Startup.Configure`, add the following code before the call to `app.UseAuthentication();`:

```
app.UseCertificateForwarding();
```

Configure Certificate Forwarding Middleware to specify the header name that Azure uses. In `Startup.ConfigureServices`, add the following code to configure the header from which the middleware builds a certificate:

```
services.AddCertificateForwarding(options =>
    options.CertificateHeader = "X-ARR-ClientCert");
```

Other web proxies

If a proxy is used that isn't IIS or Azure App Service's Application Request Routing (ARR), configure the proxy to forward the certificate that it received in an HTTP header. In `Startup.Configure`, add the following code before the call to `app.UseAuthentication();`:

```
app.UseCertificateForwarding();
```

Configure the Certificate Forwarding Middleware to specify the header name. In `Startup.ConfigureServices`, add the following code to configure the header from which the middleware builds a certificate:

```
services.AddCertificateForwarding(options =>
    options.CertificateHeader = "YOUR_CERTIFICATE_HEADER_NAME");
```

If the proxy isn't base64-encoding the certificate (as is the case with Nginx), set the `HeaderConverter` option. Consider the following example in `Startup.ConfigureServices`:

```
services.AddCertificateForwarding(options =>
{
    options.CertificateHeader = "YOUR_CUSTOM_HEADER_NAME";
    options.HeaderConverter = (headerValue) =>
    {
        var clientCertificate =
           /* some conversion logic to create an X509Certificate2 */
        return clientCertificate;
    }
});
```

Troubleshoot

When headers aren't forwarded as expected, enable logging. If the logs don't provide sufficient information to troubleshoot the problem, enumerate the request headers received by the server. Use inline middleware to write request headers to an app response or log the headers.

To write the headers to the app's response, place the following terminal inline middleware immediately after the call to UseForwardedHeaders in `Startup.Configure`:

```
app.Run(async (context) =>
{
    context.Response.ContentType = "text/plain";

    // Request method, scheme, and path
    await context.Response.WriteAsync(
        $"Request Method: {context.Request.Method}{Environment.NewLine}");
    await context.Response.WriteAsync(
        $"Request Scheme: {context.Request.Scheme}{Environment.NewLine}");
    await context.Response.WriteAsync(
        $"Request Path: {context.Request.Path}{Environment.NewLine}");

    // Headers
    await context.Response.WriteAsync($"Request Headers:{Environment.NewLine}");

    foreach (var header in context.Request.Headers)
    {
        await context.Response.WriteAsync($"{header.Key}: " +
            $"{header.Value}{Environment.NewLine}");
    }

    await context.Response.WriteAsync(Environment.NewLine);

    // Connection: RemoteIp
    await context.Response.WriteAsync(
        $"Request RemoteIp: {context.Connection.RemoteIpAddress}");
});
```

You can write to logs instead of the response body. Writing to logs allows the site to function normally while debugging.

To write logs rather than to the response body:

- Inject `ILogger<Startup>` into the `Startup` class as described in Create logs in Startup.
- Place the following inline middleware immediately after the call to UseForwardedHeaders in `Startup.Configure`.

```
app.Use(async (context, next) =>
{
    // Request method, scheme, and path
    _logger.LogDebug("Request Method: {Method}", context.Request.Method);
    _logger.LogDebug("Request Scheme: {Scheme}", context.Request.Scheme);
    _logger.LogDebug("Request Path: {Path}", context.Request.Path);

    // Headers
    foreach (var header in context.Request.Headers)
    {
        _logger.LogDebug("Header: {Key}: {Value}", header.Key, header.Value);
    }

    // Connection: RemoteIp
    _logger.LogDebug("Request RemoteIp: {RemoteIpAddress}",
        context.Connection.RemoteIpAddress);

    await next();
});
```

When processed, `X-Forwarded-{For|Proto|Host}` values are moved to `X-Original-{For|Proto|Host}`. If there are multiple values in a given header, Forwarded Headers Middleware processes headers in reverse order from right to left. The default `ForwardLimit` is `1` (one), so only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased.

The request's original remote IP must match an entry in the `KnownProxies` or `KnownNetworks` lists before forwarded headers are processed. This limits header spoofing by not accepting forwarders from untrusted proxies. When an unknown proxy is detected, logging indicates the address of the proxy:

```
September 20th 2018, 15:49:44.168 Unknown proxy: 10.0.0.100:54321
```

In the preceding example, 10.0.0.100 is a proxy server. If the server is a trusted proxy, add the server's IP address to `KnownProxies` (or add a trusted network to `KnownNetworks`) in `Startup.ConfigureServices`. For more information, see the Forwarded Headers Middleware options section.

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});
```

> **IMPORTANT**
> Only allow trusted proxies and networks to forward headers. Otherwise, IP spoofing attacks are possible.

Additional resources

- Host ASP.NET Core in a web farm
- Microsoft Security Advisory CVE-2018-0787: ASP.NET Core Elevation Of Privilege Vulnerability

In the recommended configuration for ASP.NET Core, the app is hosted using IIS/ASP.NET Core Module, Nginx, or Apache. Proxy servers, load balancers, and other network appliances often obscure information about the request before it reaches the app:

- When HTTPS requests are proxied over HTTP, the original scheme (HTTPS) is lost and must be forwarded in a header.
- Because an app receives a request from the proxy and not its true source on the Internet or corporate network, the originating client IP address must also be forwarded in a header.

This information may be important in request processing, for example in redirects, authentication, link generation, policy evaluation, and client geolocation.

Forwarded headers

By convention, proxies forward information in HTTP headers.

HEADER	DESCRIPTION
X-Forwarded-For	Holds information about the client that initiated the request and subsequent proxies in a chain of proxies. This parameter may contain IP addresses (and, optionally, port numbers). In a chain of proxy servers, the first parameter indicates the client where the request was first made. Subsequent proxy identifiers follow. The last proxy in the chain isn't in the list of parameters. The last proxy's IP address, and optionally a port number, are available as the remote IP address at the transport layer.

HEADER	DESCRIPTION
X-Forwarded-Proto	The value of the originating scheme (HTTP/HTTPS). The value may also be a list of schemes if the request has traversed multiple proxies.
X-Forwarded-Host	The original value of the Host header field. Usually, proxies don't modify the Host header. See Microsoft Security Advisory CVE-2018-0787 for information on an elevation-of-privileges vulnerability that affects systems where the proxy doesn't validate or restrict Host headers to known good values.

The Forwarded Headers Middleware, from the Microsoft.AspNetCore.HttpOverrides package, reads these headers and fills in the associated fields on HttpContext.

The middleware updates:

- HttpContext.Connection.RemoteIpAddress: Set using the `X-Forwarded-For` header value. Additional settings influence how the middleware sets `RemoteIpAddress`. For details, see the Forwarded Headers Middleware options.
- HttpContext.Request.Scheme: Set using the `X-Forwarded-Proto` header value.
- HttpContext.Request.Host: Set using the `X-Forwarded-Host` header value.

Forwarded Headers Middleware default settings can be configured. The default settings are:

- There is only *one proxy* between the app and the source of the requests.
- Only loopback addresses are configured for known proxies and known networks.
- The forwarded headers are named `X-Forwarded-For` and `X-Forwarded-Proto`.

Not all network appliances add the `X-Forwarded-For` and `X-Forwarded-Proto` headers without additional configuration. Consult your appliance manufacturer's guidance if proxied requests don't contain these headers when they reach the app. If the appliance uses different header names than `X-Forwarded-For` and `X-Forwarded-Proto`, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the appliance. For more information, see Forwarded Headers Middleware options and Configuration for a proxy that uses different header names.

IIS/IIS Express and ASP.NET Core Module

Forwarded Headers Middleware is enabled by default by IIS Integration Middleware when the app is hosted out-of-process behind IIS and the ASP.NET Core Module. Forwarded Headers Middleware is activated to run first in the middleware pipeline with a restricted configuration specific to the ASP.NET Core Module due to trust concerns with forwarded headers (for example, IP spoofing). The middleware is configured to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers and is restricted to a single localhost proxy. If additional configuration is required, see the Forwarded Headers Middleware options.

Other proxy server and load balancer scenarios

Outside of using IIS Integration when hosting out-of-process, Forwarded Headers Middleware isn't enabled by default. Forwarded Headers Middleware must be enabled for an app to process forwarded headers with UseForwardedHeaders. After enabling the middleware if no ForwardedHeadersOptions are specified to the middleware, the default ForwardedHeadersOptions.ForwardedHeaders are ForwardedHeaders.None.

Configure the middleware with ForwardedHeadersOptions to forward the `X-Forwarded-For` and

`X-Forwarded-Proto` headers in `Startup.ConfigureServices`. Invoke the UseForwardedHeaders method in `Startup.Configure` before calling other middleware:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.Configure<ForwardedHeadersOptions>(options =>
    {
        options.ForwardedHeaders =
            ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    });
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseForwardedHeaders();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();
    // In ASP.NET Core 1.x, replace the following line with: app.UseIdentity();
    app.UseAuthentication();
    app.UseMvc();
}
```

NOTE

If no ForwardedHeadersOptions are specified in `Startup.ConfigureServices` or directly to the extension method with UseForwardedHeaders, the default headers to forward are ForwardedHeaders.None. The ForwardedHeaders property must be configured with the headers to forward.

Nginx configuration

To forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers, see Host ASP.NET Core on Linux with Nginx. For more information, see NGINX: Using the Forwarded header.

Apache configuration

`X-Forwarded-For` is added automatically (see Apache Module mod_proxy: Reverse Proxy Request Headers). For information on how to forward the `X-Forwarded-Proto` header, see Host ASP.NET Core on Linux with Apache.

Forwarded Headers Middleware options

ForwardedHeadersOptions control the behavior of the Forwarded Headers Middleware. The following example changes the default values:

- Limit the number of entries in the forwarded headers to `2`.
- Add a known proxy address of `127.0.10.1`.
- Change the forwarded header name from the default `X-Forwarded-For` to

`X-Forwarded-For-My-Custom-Header-Name`.

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardLimit = 2;
    options.KnownProxies.Add(IPAddress.Parse("127.0.10.1"));
    options.ForwardedForHeaderName = "X-Forwarded-For-My-Custom-Header-Name";
});
```

OPTION	DESCRIPTION
AllowedHosts	Restricts hosts by the `X-Forwarded-Host` header to the values provided. • Values are compared using ordinal-ignore-case. • Port numbers must be excluded. • If the list is empty, all hosts are allowed. • A top-level wildcard `*` allows all non-empty hosts. • Subdomain wildcards are permitted but don't match the root domain. For example, `*.contoso.com` matches the subdomain `foo.contoso.com` but not the root domain `contoso.com`. • Unicode host names are allowed but are converted to Punycode for matching. • IPv6 addresses must include bounding brackets and be in conventional form (for example, `[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]`). IPv6 addresses aren't special-cased to check for logical equality between different formats, and no canonicalization is performed. • Failure to restrict the allowed hosts may allow an attacker to spoof links generated by the service. The default value is an empty `IList<string>`.
ForwardedForHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedForHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-For` header but uses some other header to forward the information. The default is `X-Forwarded-For`.
ForwardedHeaders	Identifies which forwarders should be processed. See the ForwardedHeaders Enum for the list of fields that apply. Typical values assigned to this property are `ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto`. The default value is ForwardedHeaders.None.

OPTION	DESCRIPTION
ForwardedHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedHostHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Host` header but uses some other header to forward the information. The default is `X-Forwarded-Host`.
ForwardedProtoHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedProtoHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Proto` header but uses some other header to forward the information. The default is `X-Forwarded-Proto`.
ForwardLimit	Limits the number of entries in the headers that are processed. Set to `null` to disable the limit, but this should only be done if `KnownProxies` or `KnownNetworks` are configured. Setting a non-`null` value is a precaution (but not a guarantee) to guard against misconfigured proxies and malicious requests arriving from side-channels on the network. Forwarded Headers Middleware processes headers in reverse order from right to left. If the default value (`1`) is used, only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased. The default is `1`.
KnownNetworks	Address ranges of known networks to accept forwarded headers from. Provide IP ranges using Classless Interdomain Routing (CIDR) notation. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. For more information, see the Configuration for an IPv4 address represented as an IPv6 address section. The default is an `IList`<IPNetwork> containing a single entry for `IPAddress.Loopback`.

OPTION	DESCRIPTION
KnownProxies	Addresses of known proxies to accept forwarded headers from. Use `KnownProxies` to specify exact IP address matches. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. For more information, see the Configuration for an IPv4 address represented as an IPv6 address section. The default is an `IList`<IPAddress> containing a single entry for `IPAddress.IPv6Loopback`.
OriginalForHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalForHeaderName. The default is `X-Original-For`.
OriginalHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalHostHeaderName. The default is `X-Original-Host`.
OriginalProtoHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalProtoHeaderName. The default is `X-Original-Proto`.
RequireHeaderSymmetry	Require the number of header values to be in sync between the ForwardedHeadersOptions.ForwardedHeaders being processed. The default in ASP.NET Core 1.x is `true`. The default in ASP.NET Core 2.0 or later is `false`.

Scenarios and use cases

When it isn't possible to add forwarded headers and all requests are secure

In some cases, it might not be possible to add forwarded headers to the requests proxied to the app. If the proxy is enforcing that all public external requests are HTTPS, the scheme can be manually set in `Startup.Configure` before using any type of middleware:

```
app.Use((context, next) =>
{
    context.Request.Scheme = "https";
    return next();
});
```

This code can be disabled with an environment variable or other configuration setting in a development or staging environment.

Deal with path base and proxies that change the request path

Some proxies pass the path intact but with an app base path that should be removed so that routing works properly. UsePathBaseExtensions.UsePathBase middleware splits the path into HttpRequest.Path and the app base path into HttpRequest.PathBase.

If `/foo` is the app base path for a proxy path passed as `/foo/api/1`, the middleware sets `Request.PathBase` to `/foo` and `Request.Path` to `/api/1` with the following command:

```
app.UsePathBase("/foo");
```

The original path and path base are reapplied when the middleware is called again in reverse. For more information on middleware order processing, see ASP.NET Core Middleware.

If the proxy trims the path (for example, forwarding `/foo/api/1` to `/api/1`), fix redirects and links by setting the request's PathBase property:

```
app.Use((context, next) =>
{
    context.Request.PathBase = new PathString("/foo");
    return next();
});
```

If the proxy is adding path data, discard part of the path to fix redirects and links by using StartsWithSegments and assigning to the Path property:

```
app.Use((context, next) =>
{
    if (context.Request.Path.StartsWithSegments("/foo", out var remainder))
    {
        context.Request.Path = remainder;
    }

    return next();
});
```

Configuration for a proxy that uses different header names

If the proxy doesn't use headers named `X-Forwarded-For` and `X-Forwarded-Proto` to forward the proxy address/port and originating scheme information, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the proxy:

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedForHeaderName = "Header_Name_Used_By_Proxy_For_X-Forwarded-For_Header";
    options.ForwardedProtoHeaderName = "Header_Name_Used_By_Proxy_For_X-Forwarded-Proto_Header";
});
```

Configuration for an IPv4 address represented as an IPv6 address

If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1` or `::ffff:a00:1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress.

In the following example, a network address that supplies forwarded headers is added to the

`KnownNetworks` list in IPv6 format.

IPv4 address: `10.11.12.1/8`

Converted IPv6 address: `::ffff:10.11.12.1`
Converted prefix length: 104

You can also supply the address in hexadecimal format (`10.11.12.1` represented in IPv6 as `::ffff:0a0b:0c01`). When converting an IPv4 address to IPv6, add 96 to the CIDR Prefix Length (`8` in the example) to account for the additional `::ffff:` IPv6 prefix (8 + 96 = 104).

```
// To access IPNetwork and IPAddress, add the following namespaces:
// using System.Net;
// using Microsoft.AspNetCore.HttpOverrides;
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    options.KnownNetworks.Add(new IPNetwork(
        IPAddress.Parse("::ffff:10.11.12.1"), 104));
});
```

Forward the scheme for Linux and non-IIS reverse proxies

Apps that call UseHttpsRedirection and UseHsts put a site into an infinite loop if deployed to an Azure Linux App Service, Azure Linux virtual machine (VM), or behind any other reverse proxy besides IIS. TLS is terminated by the reverse proxy, and Kestrel isn't made aware of the correct request scheme. OAuth and OIDC also fail in this configuration because they generate incorrect redirects. UseIISIntegration adds and configures Forwarded Headers Middleware when running behind IIS, but there's no matching automatic configuration for Linux (Apache or Nginx integration).

To forward the scheme from the proxy in non-IIS scenarios, add and configure Forwarded Headers Middleware. In `Startup.ConfigureServices`, use the following code:

```
// using Microsoft.AspNetCore.HttpOverrides;

if (string.Equals(
    Environment.GetEnvironmentVariable("ASPNETCORE_FORWARDEDHEADERS_ENABLED"),
    "true", StringComparison.OrdinalIgnoreCase))
{
    services.Configure<ForwardedHeadersOptions>(options =>
    {
        options.ForwardedHeaders = ForwardedHeaders.XForwardedFor |
            ForwardedHeaders.XForwardedProto;
        // Only loopback proxies are allowed by default.
        // Clear that restriction because forwarders are enabled by explicit
        // configuration.
        options.KnownNetworks.Clear();
        options.KnownProxies.Clear();
    });
}
```

Troubleshoot

When headers aren't forwarded as expected, enable logging. If the logs don't provide sufficient information to troubleshoot the problem, enumerate the request headers received by the server. Use inline middleware to write request headers to an app response or log the headers.

To write the headers to the app's response, place the following terminal inline middleware immediately after the call to UseForwardedHeaders in `Startup.Configure`:

```
app.Run(async (context) =>
{
    context.Response.ContentType = "text/plain";

    // Request method, scheme, and path
    await context.Response.WriteAsync(
        $"Request Method: {context.Request.Method}{Environment.NewLine}");
    await context.Response.WriteAsync(
        $"Request Scheme: {context.Request.Scheme}{Environment.NewLine}");
    await context.Response.WriteAsync(
        $"Request Path: {context.Request.Path}{Environment.NewLine}");

    // Headers
    await context.Response.WriteAsync($"Request Headers:{Environment.NewLine}");

    foreach (var header in context.Request.Headers)
    {
        await context.Response.WriteAsync($"{header.Key}: " +
            $"{header.Value}{Environment.NewLine}");
    }

    await context.Response.WriteAsync(Environment.NewLine);

    // Connection: RemoteIp
    await context.Response.WriteAsync(
        $"Request RemoteIp: {context.Connection.RemoteIpAddress}");
});
```

You can write to logs instead of the response body. Writing to logs allows the site to function normally while debugging.

To write logs rather than to the response body:

- Inject `ILogger<Startup>` into the `Startup` class as described in Create logs in Startup.
- Place the following inline middleware immediately after the call to UseForwardedHeaders in `Startup.Configure`.

```
app.Use(async (context, next) =>
{
    // Request method, scheme, and path
    _logger.LogDebug("Request Method: {Method}", context.Request.Method);
    _logger.LogDebug("Request Scheme: {Scheme}", context.Request.Scheme);
    _logger.LogDebug("Request Path: {Path}", context.Request.Path);

    // Headers
    foreach (var header in context.Request.Headers)
    {
        _logger.LogDebug("Header: {Key}: {Value}", header.Key, header.Value);
    }

    // Connection: RemoteIp
    _logger.LogDebug("Request RemoteIp: {RemoteIpAddress}",
        context.Connection.RemoteIpAddress);

    await next();
});
```

When processed, `X-Forwarded-{For|Proto|Host}` values are moved to `X-Original-{For|Proto|Host}`. If there are multiple values in a given header, Forwarded Headers Middleware processes headers in reverse

order from right to left. The default `ForwardLimit` is `1` (one), so only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased.

The request's original remote IP must match an entry in the `KnownProxies` or `KnownNetworks` lists before forwarded headers are processed. This limits header spoofing by not accepting forwarders from untrusted proxies. When an unknown proxy is detected, logging indicates the address of the proxy:

```
September 20th 2018, 15:49:44.168 Unknown proxy: 10.0.0.100:54321
```

In the preceding example, 10.0.0.100 is a proxy server. If the server is a trusted proxy, add the server's IP address to `KnownProxies` (or add a trusted network to `KnownNetworks`) in `Startup.ConfigureServices`. For more information, see the Forwarded Headers Middleware options section.

```
services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});
```

> **IMPORTANT**
>
> Only allow trusted proxies and networks to forward headers. Otherwise, IP spoofing attacks are possible.

Additional resources

- Host ASP.NET Core in a web farm
- Microsoft Security Advisory CVE-2018-0787: ASP.NET Core Elevation Of Privilege Vulnerability

Host ASP.NET Core in a web farm

9/22/2020 • 5 minutes to read • Edit Online

By Chris Ross

A *web farm* is a group of two or more web servers (or *nodes*) that host multiple instances of an app. When requests from users arrive to a web farm, a *load balancer* distributes the requests to the web farm's nodes. Web farms improve:

- **Reliability/availability**: When one or more nodes fail, the load balancer can route requests to other functioning nodes to continue processing requests.
- **Capacity/performance**: Multiple nodes can process more requests than a single server. The load balancer balances the workload by distributing requests to the nodes.
- **Scalability**: When more or less capacity is required, the number of active nodes can be increased or decreased to match the workload. Web farm platform technologies, such as Azure App Service, can automatically add or remove nodes at the request of the system administrator or automatically without human intervention.
- **Maintainability**: Nodes of a web farm can rely on a set of shared services, which results in easier system management. For example, the nodes of a web farm can rely upon a single database server and a common network location for static resources, such as images and downloadable files.

This topic describes configuration and dependencies for ASP.NET core apps hosted in a web farm that rely upon shared resources.

General configuration

Host and deploy ASP.NET Core
Learn how to set up hosting environments and deploy ASP.NET Core apps. Configure a process manager on each node of the web farm to automate app starts and restarts. Each node requires the ASP.NET Core runtime. For more information, see the topics in the Host and deploy area of the documentation.

Configure ASP.NET Core to work with proxy servers and load balancers
Learn about configuration for apps hosted behind proxy servers and load balancers, which often obscure important request information.

Deploy ASP.NET Core apps to Azure App Service
Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core. App Service is a fully managed platform that provides automatic scaling, load balancing, patching, and continuous deployment.

App data

When an app is scaled to multiple instances, there might be app state that requires sharing across nodes. If the state is transient, consider sharing an IDistributedCache. If the shared state requires persistence, consider storing the shared state in a database.

Required configuration

Data Protection and Caching require configuration for apps deployed to a web farm.

Data Protection

The ASP.NET Core Data Protection system is used by apps to protect data. Data Protection relies upon a set of cryptographic keys stored in a *key ring*. When the Data Protection system is initialized, it applies default settings that store the key ring locally. Under the default configuration, a unique key ring is stored on each node of the web farm. Consequently, each web farm node can't decrypt data that's encrypted by an app on any other node. The default configuration isn't generally appropriate for hosting apps in a web farm. An alternative to implementing a shared key ring is to always route user requests to the same node. For more information on Data Protection system configuration for web farm deployments, see Configure ASP.NET Core Data Protection.

Caching

In a web farm environment, the caching mechanism must share cached items across the web farm's nodes. Caching must either rely upon a common Redis cache, a shared SQL Server database, or a custom caching implementation that shares cached items across the web farm. For more information, see Distributed caching in ASP.NET Core.

Dependent components

The following scenarios don't require additional configuration, but they depend on technologies that require configuration for web farms.

SCENARIO	DEPENDS ON ...
Authentication	Data Protection (see Configure ASP.NET Core Data Protection). For more information, see Use cookie authentication without ASP.NET Core Identity and Share authentication cookies among ASP.NET apps.
Identity	Authentication and database configuration. For more information, see Introduction to Identity on ASP.NET Core.
Session	Data Protection (encrypted cookies) (see Configure ASP.NET Core Data Protection) and Caching (see Distributed caching in ASP.NET Core). For more information, see Session and state management: Session state.
TempData	Data Protection (encrypted cookies) (see Configure ASP.NET Core Data Protection) or Session (see Session and state management: Session state). For more information, see Session and state management: TempData.
Anti-forgery	Data Protection (see Configure ASP.NET Core Data Protection). For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Troubleshoot

Data Protection and caching

When Data Protection or caching isn't configured for a web farm environment, intermittent errors occur when

requests are processed. This occurs because nodes don't share the same resources and user requests aren't always routed back to the same node.

Consider a user who signs into the app using cookie authentication. The user signs into the app on one web farm node. If their next request arrives at the same node where they signed in, the app is able to decrypt the authentication cookie and allows access to the app's resource. If their next request arrives at a different node, the app can't decrypt the authentication cookie from the node where the user signed in, and authorization for the requested resource fails.

When any of the following symptoms occur **intermittently**, the problem is usually traced to improper Data Protection or caching configuration for a web farm environment:

- Authentication breaks: The authentication cookie is misconfigured or can't be decrypted. OAuth (Facebook, Microsoft, Twitter) or OpenIdConnect logins fail with the error "Correlation failed."
- Authorization breaks: Identity is lost.
- Session state loses data.
- Cached items disappear.
- TempData fails.
- POSTs fail: The anti-forgery check fails.

For more information on Data Protection configuration for web farm deployments, see Configure ASP.NET Core Data Protection. For more information on caching configuration for web farm deployments, see Distributed caching in ASP.NET Core.

Obtain data from apps

If the web farm apps are capable of responding to requests, obtain request, connection, and additional data from the apps using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Additional resources

- Custom Script Extension for Windows: Downloads and executes scripts on Azure virtual machines, which is useful for post-deployment configuration and software installation.
- Configure ASP.NET Core to work with proxy servers and load balancers

Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

9/22/2020 • 13 minutes to read • Edit Online

By Sayed Ibrahim Hashimi and Rick Anderson

This document focuses on using Visual Studio 2019 or later to create and use publish profiles. The publish profiles created with Visual Studio can be used with MSBuild and Visual Studio. For instructions on publishing to Azure, see Publish an ASP.NET Core app to Azure with Visual Studio.

The `dotnet new mvc` command produces a project file containing the following root-level <Project> element:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
    <!-- omitted for brevity -->
</Project>
```

The preceding `<Project>` element's `Sdk` attribute imports the MSBuild properties and targets from *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.props* and *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.targets*, respectively. The default location for `$(MSBuildSDKsPath)` (with Visual Studio 2019 Enterprise) is the *%programfiles(x86)%\Microsoft Visual Studio\2019\Enterprise\MSBuild\Sdks* folder.

`Microsoft.NET.Sdk.Web` (Web SDK) depends on other SDKs, including `Microsoft.NET.Sdk` (.NET Core SDK) and `Microsoft.NET.Sdk.Razor` (Razor SDK). The MSBuild properties and targets associated with each dependent SDK are imported. Publish targets import the appropriate set of targets based on the publish method used.

When MSBuild or Visual Studio loads a project, the following high-level actions occur:

- Build project
- Compute files to publish
- Publish files to destination

Compute project items

When the project is loaded, the MSBuild project items (files) are computed. The item type determines how the file is processed. By default, *.cs* files are included in the `Compile` item list. Files in the `Compile` item list are compiled.

The `Content` item list contains files that are published in addition to the build outputs. By default, files matching the patterns `wwwroot\**`, `**\*.config`, and `**\*.json` are included in the `Content` item list. For example, the `wwwroot\**` globbing pattern matches all files in the *wwwroot* folder and its subfolders.

The Web SDK imports the Razor SDK. As a result, files matching the patterns `**\*.cshtml` and `**\*.razor` are also included in the `Content` item list.

The Web SDK imports the Razor SDK. As a result, files matching the `**\*.cshtml` pattern are also included in the `Content` item list.

To explicitly add a file to the publish list, add the file directly in the *.csproj* file as shown in the Include Files section.

When selecting the **Publish** button in Visual Studio or when publishing from the command line:

- The properties/items are computed (the files that are needed to build).
- **Visual Studio only**: NuGet packages are restored. (Restore needs to be explicit by the user on the CLI.)
- The project builds.
- The publish items are computed (the files that are needed to publish).
- The project is published (the computed files are copied to the publish destination).

When an ASP.NET Core project references `Microsoft.NET.Sdk.Web` in the project file, an *app_offline.htm* file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the *app_offline.htm* file during the deployment. For more information, see the ASP.NET Core Module configuration reference.

Basic command-line publishing

Command-line publishing works on all .NET Core-supported platforms and doesn't require Visual Studio. In the following examples, the .NET Core CLI's dotnet publish command is run from the project directory (which contains the *.csproj* file). If the project folder isn't the current working directory, explicitly pass in the project file path. For example:

```
dotnet publish C:\Webs\Web1
```

Run the following commands to create and publish a web app:

```
dotnet new mvc
dotnet publish
```

The `dotnet publish` command produces a variation of the following output:

```
C:\Webs\Web1>dotnet publish
Microsoft (R) Build Engine version {VERSION} for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 36.81 ms for C:\Webs\Web1\Web1.csproj.
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\Web1.dll
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\Web1.Views.dll
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\publish\
```

The default publish folder format is *bin\Debug\{TARGET FRAMEWORK MONIKER}\publish*. For example, *bin\Debug\netcoreapp2.2\publish*.

The following command specifies a `Release` build and the publishing directory:

```
dotnet publish -c Release -o C:\MyWebs\test
```

The `dotnet publish` command calls MSBuild, which invokes the `Publish` target. Any parameters passed to `dotnet publish` are passed to MSBuild. The `-c` and `-o` parameters map to MSBuild's `Configuration` and `OutputPath` properties, respectively.

MSBuild properties can be passed using either of the following formats:

- `p:<NAME>=<VALUE>`
- `/p:<NAME>=<VALUE>`

For example, the following command publishes a `Release` build to a network share. The network share is

specified with forward slashes (*//r8/*) and works on all .NET Core supported platforms.

```
dotnet publish -c Release /p:PublishDir=//r8/release/AdminWeb
```

Confirm that the published app for deployment isn't running. Files in the *publish* folder are locked when the app is running. Deployment can't occur because locked files can't be copied.

Publish profiles

This section uses Visual Studio 2019 or later to create a publishing profile. Once the profile is created, publishing from Visual Studio or the command line is available. Publish profiles can simplify the publishing process, and any number of profiles can exist.

Create a publish profile in Visual Studio by choosing one of the following paths:

- Right-click the project in **Solution Explorer** and select **Publish**.
- Select **Publish {PROJECT NAME}** from the **Build** menu.

The **Publish** tab of the app capabilities page is displayed. If the project lacks a publish profile, the **Pick a publish target** page is displayed. You're asked to select one of the following publish targets:

- Azure App Service
- Azure App Service on Linux
- Azure Virtual Machines
- Folder
- IIS, FTP, Web Deploy (for any web server)
- Import Profile

To determine the most appropriate publish target, see What publishing options are right for me.

When the **Folder** publish target is selected, specify a folder path to store the published assets. The default folder path is *bin\{PROJECT CONFIGURATION}\{TARGET FRAMEWORK MONIKER}\publish*. For example, *bin\Release\netcoreapp2.2\publish*. Select the **Create Profile** button to finish.

Once a publish profile is created, the **Publish** tab's content changes. The newly created profile appears in a drop-down list. Below the drop-down list, select **Create new profile** to create another new profile.

Visual Studio's publish tool produces a *Properties/PublishProfiles/{PROFILE NAME}.pubxml* MSBuild file describing the publish profile. The *.pubxml* file:

- Contains publish configuration settings and is consumed by the publishing process.
- Can be modified to customize the build and publish process.

When publishing to an Azure target, the *.pubxml* file contains your Azure subscription identifier. With that target type, adding this file to source control is discouraged. When publishing to a non-Azure target, it's safe to check in the *.pubxml* file.

Sensitive information (like the publish password) is encrypted on a per user/machine level. It's stored in the *Properties/PublishProfiles/{PROFILE NAME}.pubxml.user* file. Because this file can store sensitive information, it shouldn't be checked into source control.

For an overview of how to publish an ASP.NET Core web app, see Host and deploy ASP.NET Core. The MSBuild tasks and targets necessary to publish an ASP.NET Core web app are open-source in the dotnet/websdk repository.

The following commands can use folder, MSDeploy, and Kudu publish profiles. Because MSDeploy lacks cross-

platform support, the following MSDeploy options are supported only on Windows.

Folder (works cross-platform):

```
dotnet publish WebApplication.csproj /p:PublishProfile=<FolderProfileName>
```

```
dotnet build WebApplication.csproj /p:DeployOnBuild=true /p:PublishProfile=<FolderProfileName>
```

MSDeploy:

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployProfileName> /p:Password=
<DeploymentPassword>
```

```
dotnet build WebApplication.csproj /p:DeployOnBuild=true /p:PublishProfile=<MsDeployProfileName>
/p:Password=<DeploymentPassword>
```

MSDeploy package:

```
dotnet publish WebApplication.csproj /p:PublishProfile=<MsDeployPackageProfileName>
```

```
dotnet build WebApplication.csproj /p:DeployOnBuild=true /p:PublishProfile=<MsDeployPackageProfileName>
```

In the preceding examples:

- `dotnet publish` and `dotnet build` support Kudu APIs to publish to Azure from any platform. Visual Studio publish supports the Kudu APIs, but it's supported by WebSDK for cross-platform publish to Azure.
- Don't pass `DeployOnBuild` to the `dotnet publish` command.

For more information, see Microsoft.NET.Sdk.Publish.

Add a publish profile to the project's *Properties/PublishProfiles* folder with the following content:

```
<Project>
  <PropertyGroup>
    <PublishProtocol>Kudu</PublishProtocol>
    <PublishSiteName>nodewebapp</PublishSiteName>
    <UserName>username</UserName>
    <Password>password</Password>
  </PropertyGroup>
</Project>
```

Folder publish example

When publishing with a profile named *FolderProfile*, use any of the following commands:

```
dotnet publish /p:Configuration=Release /p:PublishProfile=FolderProfile`
```

```
dotnet build /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

```
msbuild /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

The .NET Core CLI's dotnet build command calls `msbuild` to run the build and publish process. The `dotnet build` and `msbuild` commands are equivalent when passing in a folder profile. When calling `msbuild` directly on Windows, the .NET Framework version of MSBuild is used. Calling `dotnet build` on a non-folder profile:

- Invokes `msbuild`, which uses MSDeploy.
- Results in a failure (even when running on Windows). To publish with a non-folder profile, call `msbuild` directly.

The following folder publish profile was created with Visual Studio and publishes to a network share:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project.
You can customize the behavior of this process by editing this
MSBuild file.
-->
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework>netcoreapp1.1</PublishFramework>
    <ProjectGuid>c30c453c-312e-40c4-aec9-394a145dee0b</ProjectGuid>
    <publishUrl>\\r8\Release\AdminWeb</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
</Project>
```

In the preceding example:

- The `<ExcludeApp_Data>` property is present merely to satisfy an XML schema requirement. The `<ExcludeApp_Data>` property has no effect on the publish process, even if there's an *App_Data* folder in the project root. The *App_Data* folder doesn't receive special treatment as it does in ASP.NET 4.x projects.
- The `<LastUsedBuildConfiguration>` property is set to `Release`. When publishing from Visual Studio, the value of `<LastUsedBuildConfiguration>` is set using the value when the publish process is started. `<LastUsedBuildConfiguration>` is special and shouldn't be overridden in an imported MSBuild file. This property can, however, be overridden from the command line using one of the following approaches.
  - Using the .NET Core CLI:

    ```
    dotnet publish /p:Configuration=Release /p:PublishProfile=FolderProfile
    ```

    ```
    dotnet build -c Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
    ```

  - Using MSBuild:

    ```
    msbuild /p:Configuration=Release /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
    ```

For more information, see MSBuild: how to set the configuration property.

Publish to an MSDeploy endpoint from the command line

The following example uses an ASP.NET Core web app created by Visual Studio named *AzureWebApp*. An Azure Apps publish profile is added with Visual Studio. For more information on how to create a profile, see the Publish profiles section.

To deploy the app using a publish profile, execute the `msbuild` command from a Visual Studio **Developer Command Prompt**. The command prompt is available in the *Visual Studio* folder of the **Start** menu on the Windows taskbar. For easier access, you can add the command prompt to the **Tools** menu in Visual Studio. For more information, see Developer Command Prompt for Visual Studio.

MSBuild uses the following command syntax:

```
msbuild {PATH}
    /p:DeployOnBuild=true
    /p:PublishProfile={PROFILE}
    /p:Username={USERNAME}
    /p:Password={PASSWORD}
```

- `{PATH}`: Path to the app's project file.
- `{PROFILE}`: Name of the publish profile.
- `{USERNAME}`: MSDeploy username. The `{USERNAME}` can be found in the publish profile.
- `{PASSWORD}`: MSDeploy password. Obtain the `{PASSWORD}` from the *{PROFILE}.PublishSettings* file. Download the *.PublishSettings* file from either:
  - **Solution Explorer**: Select **View** > **Cloud Explorer**. Connect with your Azure subscription. Open **App Services**. Right-click the app. Select **Download Publish Profile**.
  - Azure portal: Select **Get publish profile** in the web app's **Overview** panel.

The following example uses a publish profile named *AzureWebApp - Web Deploy*:

```
msbuild "AzureWebApp.csproj"
    /p:DeployOnBuild=true
    /p:PublishProfile="AzureWebApp - Web Deploy"
    /p:Username="$AzureWebApp"
    /p:Password="........."
```

A publish profile can also be used with the .NET Core CLI's dotnet msbuild command from a Windows command shell:

```
dotnet msbuild "AzureWebApp.csproj"
    /p:DeployOnBuild=true
    /p:PublishProfile="AzureWebApp - Web Deploy"
    /p:Username="$AzureWebApp"
    /p:Password="........."
```

IMPORTANT

The `dotnet msbuild` command is a cross-platform command and can compile ASP.NET Core apps on macOS and Linux. However, MSBuild on macOS and Linux isn't capable of deploying an app to Azure or other MSDeploy endpoints.

Set the environment

Include the `<EnvironmentName>` property in the publish profile (*.pubxml*) or project file to set the app's environment:

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

If you require *web.config* transformations (for example, setting environment variables based on the configuration, profile, or environment), see Transform web.config.

Exclude files

When publishing ASP.NET Core web apps, the following assets are included:

- Build artifacts
- Folders and files matching the following globbing patterns:
  - `**\*.config` (for example, *web.config*)
  - `**\*.json` (for example, *appsettings.json*)
  - `wwwroot\**`

MSBuild supports globbing patterns. For example, the following `<Content>` element suppresses the copying of text (*.txt*) files in the *wwwroot\content* folder and its subfolders:

```
<ItemGroup>
  <Content Update="wwwroot/content/**/*.txt" CopyToPublishDirectory="Never" />
</ItemGroup>
```

The preceding markup can be added to a publish profile or the *.csproj* file. When added to the *.csproj* file, the rule is added to all publish profiles in the project.

The following `<MsDeploySkipRules>` element excludes all files from the *wwwroot\content* folder:

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFolder">
    <ObjectName>dirPath</ObjectName>
    <AbsolutePath>wwwroot\\content</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

`<MsDeploySkipRules>` won't delete the *skip* targets from the deployment site. `<Content>` targeted files and folders are deleted from the deployment site. For example, suppose a deployed web app had the following files:

- *Views/Home/About1.cshtml*
- *Views/Home/About2.cshtml*
- *Views/Home/About3.cshtml*

If the following `<MsDeploySkipRules>` elements are added, those files wouldn't be deleted on the deployment site.

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About1.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About2.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About3.cshtml</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

The preceding `<MsDeploySkipRules>` elements prevent the *skipped* files from being deployed. It won't delete those files once they're deployed.

The following `<Content>` element deletes the targeted files at the deployment site:

```
<ItemGroup>
  <Content Update="Views/Home/About?.cshtml" CopyToPublishDirectory="Never" />
</ItemGroup>
```

Using command-line deployment with the preceding `<Content>` element yields a variation of the following output:

```
MSDeployPublish:
  Starting Web deployment task from source: manifest(C:\Webs\Web1\obj\Release\{TARGET FRAMEWORK
MONIKER}\PubTmp\Web1.SourceManifest.
  xml) to Destination: auto().
  Deleting file (Web11112\Views\Home\About1.cshtml).
  Deleting file (Web11112\Views\Home\About2.cshtml).
  Deleting file (Web11112\Views\Home\About3.cshtml).
  Updating file (Web11112\web.config).
  Updating file (Web11112\Web1.deps.json).
  Updating file (Web11112\Web1.dll).
  Updating file (Web11112\Web1.pdb).
  Updating file (Web11112\Web1.runtimeconfig.json).
  Successfully executed Web deployment task.
  Publish Succeeded.
Done Building Project "C:\Webs\Web1\Web1.csproj" (default targets).
```

Include files

The following sections outline different approaches for file inclusion at publish time. The General file inclusion section uses the `DotNetPublishFiles` item, which is provided by a publish targets file in the Web SDK. The Selective file inclusion section uses the `ResolvedFileToPublish` item, which is provided by a publish targets file in the .NET Core SDK. Because the Web SDK depends on the .NET Core SDK, either item can be used in an ASP.NET Core project.

General file inclusion

The following example's `<ItemGroup>` element demonstrates copying a folder located outside of the project directory to a folder of the published site. Any files added to the following markup's `<ItemGroup>` are included by default.

```
<ItemGroup>
  <_CustomFiles Include="$(MSBuildProjectDirectory)/../images/**/*" />
  <DotNetPublishFiles Include="@(_CustomFiles)">
    <DestinationRelativePath>wwwroot/images/%(RecursiveDir)%(Filename)%(Extension)
</DestinationRelativePath>
  </DotNetPublishFiles>
</ItemGroup>
```

The preceding markup:

- Can be added to the *.csproj* file or the publish profile. If it's added to the *.csproj* file, it's included in each publish profile in the project.
- Declares a `_CustomFiles` item to store files matching the `Include` attribute's globbing pattern. The *images* folder referenced in the pattern is located outside of the project directory. A reserved property, named `$(MSBuildProjectDirectory)`, resolves to the project file's absolute path.
- Provides a list of files to the `DotNetPublishFiles` item. By default, the item's `<DestinationRelativePath>` element is empty. The default value is overridden in the markup and uses well-known item metadata such as `%(RecursiveDir)`. The inner text represents the *wwwroot/images* folder of the published site.

Selective file inclusion

The highlighted markup in the following example demonstrates:

- Copying a file located outside of the project into the published site's *wwwroot* folder. The file name of *ReadMe2.md* is maintained.
- Excluding the *wwwroot\Content* folder.
- Excluding *Views\Home\About2.cshtml*.

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework />
    <ProjectGuid>afa9f185-7ce0-4935-9da1-ab676229d68a</ProjectGuid>
    <publishUrl>bin\Release\PublishOutput</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
  <ItemGroup>
    <ResolvedFileToPublish Include="..\ReadMe2.md">
      <RelativePath>wwwroot\ReadMe2.md</RelativePath>
    </ResolvedFileToPublish>

    <Content Update="wwwroot\Content\**\*" CopyToPublishDirectory="Never" />
    <Content Update="Views\Home\About2.cshtml" CopyToPublishDirectory="Never" />
  </ItemGroup>
</Project>
```

The preceding example uses the `ResolvedFileToPublish` item, whose default behavior is to always copy the files provided in the `Include` attribute to the published site. Override the default behavior by including a `<CopyToPublishDirectory>` child element with inner text of either `Never` or `PreserveNewest`. For example:

```
<ResolvedFileToPublish Include="..\ReadMe2.md">
  <RelativePath>wwwroot\ReadMe2.md</RelativePath>
  <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
</ResolvedFileToPublish>
```

For more deployment samples, see the Web SDK README file.

Run a target before or after publishing

The built-in `BeforePublish` and `AfterPublish` targets execute a target before or after the publish target. Add the following elements to the publish profile to log console messages both before and after publishing:

```
<Target Name="CustomActionsBeforePublish" BeforeTargets="BeforePublish">
    <Message Text="Inside BeforePublish" Importance="high" />
  </Target>
  <Target Name="CustomActionsAfterPublish" AfterTargets="AfterPublish">
    <Message Text="Inside AfterPublish" Importance="high" />
</Target>
```

Publish to a server using an untrusted certificate

Add the `<AllowUntrustedCertificate>` property with a value of `True` to the publish profile:

```
<PropertyGroup>
  <AllowUntrustedCertificate>True</AllowUntrustedCertificate>
</PropertyGroup>
```

The Kudu service

To view the files in an Azure App Service web app deployment, use the Kudu service. Append the `scm` token to the web app name. For example:

URL	RESULT
`http://mysite.azurewebsites.net/`	Web App
`http://mysite.scm.azurewebsites.net/`	Kudu service

Select the Debug Console menu item to view, edit, delete, or add files.

Additional resources

- Web Deploy (MSDeploy) simplifies deployment of web apps and websites to IIS servers.
- Web SDK GitHub repository: File issues and request features for deployment.
- Publish an ASP.NET Web App to an Azure VM from Visual Studio
- Transform web.config

ASP.NET Core directory structure

9/22/2020 • 3 minutes to read • Edit Online

The *publish* directory contains the app's deployable assets produced by the dotnet publish command. The directory contains:

- Application files
- Configuration files
- Static assets
- Packages
- A runtime (self-contained deployment only)

APP TYPE	DIRECTORY STRUCTURE
Framework-dependent Executable (FDE)	• publish† ◦ Views† MVC apps; if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}{.EXTENSION} .exe extension on Windows, no extension on macOS or Linux ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.Views.dll ◦ {ASSEMBLY NAME}.Views.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json ◦ web.config (IIS deployments) ◦ createdump (Linux createdump utility) ◦ *.so (Linux shared object library) ◦ *.a (macOS archive) ◦ *.dylib (macOS dynamic library)
Self-contained Deployment (SCD)	• publish† ◦ Views† MVC apps, if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}.exe ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.Views.dll ◦ {ASSEMBLY NAME}.Views.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json ◦ web.config (IIS deployments)

†Indicates a directory

The *publish* directory represents the *content root path*, also called the *application base path*, of the deployment. Whatever name is given to the *publish* directory of the deployed app on the server, its location serves as the server's physical path to the hosted app.

The *wwwroot* directory, if present, only contains static assets.

Additional resources

- dotnet publish
- .NET Core application deployment
- Target frameworks
- .NET Core RID Catalog

The *publish* directory contains the app's deployable assets produced by the dotnet publish command. The directory contains:

- Application files
- Configuration files
- Static assets
- Packages
- A runtime (self-contained deployment only)

APP TYPE	DIRECTORY STRUCTURE
Framework-dependent Executable (FDE)	• publish† ◦ Views† MVC apps; if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}{.EXTENSION} .exe extension on Windows, no extension on macOS or Linux ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.Views.dll ◦ {ASSEMBLY NAME}.Views.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json ◦ web.config (IIS deployments) ◦ createdump (Linux createdump utility) ◦ *.so (Linux shared object library) ◦ *.a (macOS archive) ◦ *.dylib (macOS dynamic library)

APP TYPE	DIRECTORY STRUCTURE
Self-contained Deployment (SCD)	• publish† ◦ Views† MVC apps, if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}.exe ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.Views.dll ◦ {ASSEMBLY NAME}.Views.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json ◦ web.config (IIS deployments)

†Indicates a directory

The *publish* directory represents the *content root path*, also called the *application base path*, of the deployment. Whatever name is given to the *publish* directory of the deployed app on the server, its location serves as the server's physical path to the hosted app.

The *wwwroot* directory, if present, only contains static assets.

Creating a *Logs* folder is useful for ASP.NET Core Module enhanced debug logging. Folders in the path provided to the `<handlerSetting>` value aren't created by the module automatically and should pre-exist in the deployment to allow the module to write the debug log.

A *Logs* directory can be created for the deployment using one of the following two approaches:

- Add the following `<Target>` element to the project file:

  ```
  <Target Name="CreateLogsFolder" AfterTargets="Publish">
     <MakeDir Directories="$(PublishDir)Logs"
              Condition="!Exists('$(PublishDir)Logs')" />
     <WriteLinesToFile File="$(PublishDir)Logs\.log"
                       Lines="Generated file"
                       Overwrite="True"
                       Condition="!Exists('$(PublishDir)Logs\.log')" />
  </Target>
  ```

  The `<MakeDir>` element creates an empty *Logs* folder in the published output. The element uses the `PublishDir` property to determine the target location for creating the folder. Several deployment methods, such as Web Deploy, skip empty folders during deployment. The `<WriteLinesToFile>` element generates a file in the *Logs* folder, which guarantees deployment of the folder to the server. Folder creation using this approach fails if the worker process doesn't have write access to the target folder.

- Physically create the *Logs* directory on the server in the deployment.

The deployment directory requires Read/Execute permissions. The *Logs* directory requires Read/Write permissions. Additional directories where files are written require Read/Write permissions.

Additional resources

- dotnet publish
- .NET Core application deployment
- Target frameworks
- .NET Core RID Catalog

Health checks in ASP.NET Core

9/22/2020 • 44 minutes to read • Edit Online

By Glenn Condron

ASP.NET Core offers Health Checks Middleware and libraries for reporting the health of app infrastructure components.

Health checks are exposed by an app as HTTP endpoints. Health check endpoints can be configured for a variety of real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, a container orchestrator may respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

View or download sample code (how to download)

The sample app includes examples of the scenarios described in this topic. To run the sample app for a given scenario, use the dotnet run command from the project's folder in a command shell. See the sample app's *README.md* file and the scenario descriptions in this topic for details on how to use the sample app.

Prerequisites

Health checks are usually used with an external monitoring service or container orchestrator to check the status of an app. Before adding health checks to an app, decide on which monitoring system to use. The monitoring system dictates what types of health checks to create and how to configure their endpoints.

The Microsoft.AspNetCore.Diagnostics.HealthChecks package is referenced implicitly for ASP.NET Core apps. To perform health checks using Entity Framework Core, add a package reference to the Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore package.

The sample app provides startup code to demonstrate health checks for several scenarios. The database probe scenario checks the health of a database connection using AspNetCore.Diagnostics.HealthChecks. The DbContext probe scenario checks a database using an EF Core `DbContext`. To explore the database scenarios, the sample app:

- Creates a database and provides its connection string in the *appsettings.json* file.
- Has the following package references in its project file:
  - AspNetCore.HealthChecks.SqlServer
  - Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore

> **NOTE**
>
> AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Another health check scenario demonstrates how to filter health checks to a management port. The sample app requires you to create a *Properties/launchSettings.json* file that includes the management URL and management

port. For more information, see the Filter by port section.

Basic health probe

For many apps, a basic health probe configuration that reports the app's availability to process requests (*liveness*) is sufficient to discover the status of the app.

The basic configuration registers health check services and calls the Health Checks Middleware to respond at a URL endpoint with a health response. By default, no specific health checks are registered to test any particular dependency or subsystem. The app is considered healthy if it's capable of responding at the health endpoint URL. The default response writer writes the status (HealthStatus) as a plaintext response back to the client, indicating either a HealthStatus.Healthy, HealthStatus.Degraded or HealthStatus.Unhealthy status.

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. Create a health check endpoint by calling `MapHealthChecks` in `Startup.Configure`.

In the sample app, the health check endpoint is created at `/health` (*BasicStartup.cs*):

```
public class BasicStartup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHealthChecks();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapHealthChecks("/health");
        });
    }
}
```

To run the basic configuration scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario basic
```

Docker example

Docker offers a built-in `HEALTHCHECK` directive that can be used to check the status of an app that uses the basic health check configuration:

```
HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit
```

Create health checks

Health checks are created by implementing the IHealthCheck interface. The CheckHealthAsync method returns a HealthCheckResult that indicates the health as `Healthy`, `Degraded`, or `Unhealthy`. The result is written as a plaintext response with a configurable status code (configuration is described in the Health check options section). HealthCheckResult can also return optional key-value pairs.

The following `ExampleHealthCheck` class demonstrates the layout of a health check. The health checks logic is placed in the `CheckHealthAsync` method. The following example sets a dummy variable,

`healthCheckResultHealthy`, to `true`. If the value of `healthCheckResultHealthy` is set to `false`, the HealthCheckResult.Unhealthy status is returned.

```
public class ExampleHealthCheck : IHealthCheck
{
    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        var healthCheckResultHealthy = true;

        if (healthCheckResultHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("A healthy result."));
        }

        return Task.FromResult(
            HealthCheckResult.Unhealthy("An unhealthy result."));
    }
}
```

Register health check services

The `ExampleHealthCheck` type is added to health check services with AddCheck in `Startup.ConfigureServices`:

```
services.AddHealthChecks()
    .AddCheck<ExampleHealthCheck>("example_health_check");
```

The AddCheck overload shown in the following example sets the failure status (HealthStatus) to report when the health check reports a failure. If the failure status is set to `null` (default), HealthStatus.Unhealthy is reported. This overload is a useful scenario for library authors, where the failure status indicated by the library is enforced by the app when a health check failure occurs if the health check implementation honors the setting.

Tags can be used to filter health checks (described further in the Filter health checks section).

```
services.AddHealthChecks()
    .AddCheck<ExampleHealthCheck>(
        "example_health_check",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "example" });
```

AddCheck can also execute a lambda function. In the following example, the health check name is specified as `Example` and the check always returns a healthy state:

```
services.AddHealthChecks()
    .AddCheck("Example", () =>
        HealthCheckResult.Healthy("Example is OK!"), tags: new[] { "example" });
```

Call AddTypeActivatedCheck to pass arguments to a health check implementation. In the following example, `TestHealthCheckWithArgs` accepts an integer and a string for use when CheckHealthAsync is called:

```
private class TestHealthCheckWithArgs : IHealthCheck
{
    public TestHealthCheckWithArgs(int i, string s)
    {
        I = i;
        S = s;
    }

    public int I { get; set; }

    public string S { get; set; }

    public Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context,
        CancellationToken cancellationToken = default)
    {
        ...
    }
}
```

`TestHealthCheckWithArgs` is registered by calling `AddTypeActivatedCheck` with the integer and string passed to the implementation:

```
services.AddHealthChecks()
    .AddTypeActivatedCheck<TestHealthCheckWithArgs>(
        "test",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "example" },
        args: new object[] { 5, "string" });
```

Use Health Checks Routing

In `Startup.Configure`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
});
```

Require host

Call `RequireHost` to specify one or more permitted hosts for the health check endpoint. Hosts should be Unicode rather than punycode and may include a port. If a collection isn't supplied, any host is accepted.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health").RequireHost("www.contoso.com:5001");
});
```

For more information, see the Filter by port section.

Require authorization

Call `RequireAuthorization` to run Authorization Middleware on the health check request endpoint. A `RequireAuthorization` overload accepts one or more authorization policies. If a policy isn't provided, the default authorization policy is used.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health").RequireAuthorization();
});
```

Enable Cross-Origin Requests (CORS)

Although performing health checks manually from a browser isn't a common use scenario, CORS Middleware can be enabled by calling `RequireCors` on health checks endpoints. A `RequireCors` overload accepts a CORS policy builder delegate (`CorsPolicyBuilder`) or a policy name. If a policy isn't provided, the default CORS policy is used. For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core.

Health check options

HealthCheckOptions provide an opportunity to customize health check behavior:

- Filter health checks
- Customize the HTTP status code
- Suppress cache headers
- Customize output

Filter health checks

By default, Health Checks Middleware runs all registered health checks. To run a subset of health checks, provide a function that returns a boolean to the Predicate option. In the following example, the `Bar` health check is filtered out by its tag (`bar_tag`) in the function's conditional statement, where `true` is only returned if the health check's Tags property matches `foo_tag` or `baz_tag`:

In `Startup.ConfigureServices`:

```
services.AddHealthChecks()
    .AddCheck("Foo", () =>
        HealthCheckResult.Healthy("Foo is OK!"), tags: new[] { "foo_tag" })
    .AddCheck("Bar", () =>
        HealthCheckResult.Unhealthy("Bar is unhealthy!"), tags: new[] { "bar_tag" })
    .AddCheck("Baz", () =>
        HealthCheckResult.Healthy("Baz is OK!"), tags: new[] { "baz_tag" });
```

In `Startup.Configure`, the `Predicate` filters out the 'Bar' health check. Only Foo and Baz execute.:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health", new HealthCheckOptions()
    {
        Predicate = (check) => check.Tags.Contains("foo_tag") ||
            check.Tags.Contains("baz_tag")
    });
});
```

Customize the HTTP status code

Use ResultStatusCodes to customize the mapping of health status to HTTP status codes. The following StatusCodes assignments are the default values used by the middleware. Change the status code values to meet your requirements.

In `Startup.Configure`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health", new HealthCheckOptions()
    {
        ResultStatusCodes =
        {
            [HealthStatus.Healthy] = StatusCodes.Status200OK,
            [HealthStatus.Degraded] = StatusCodes.Status200OK,
            [HealthStatus.Unhealthy] = StatusCodes.Status503ServiceUnavailable
        }
    });
});
```

Suppress cache headers

AllowCachingResponses controls whether the Health Checks Middleware adds HTTP headers to a probe response to prevent response caching. If the value is `false` (default), the middleware sets or overrides the `Cache-Control`, `Expires`, and `Pragma` headers to prevent response caching. If the value is `true`, the middleware doesn't modify the cache headers of the response.

In `Startup.Configure`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health", new HealthCheckOptions()
    {
        AllowCachingResponses = false
    });
});
```

Customize output

In `Startup.Configure`, set the HealthCheckOptions.ResponseWriter option to a delegate for writing the response:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health", new HealthCheckOptions()
    {
        ResponseWriter = WriteResponse
    });
});
```

The default delegate writes a minimal plaintext response with the string value of HealthReport.Status. The following custom delegates output a custom JSON response.

The first example from the sample app demonstrates how to use System.Text.Json:

```
private static Task WriteResponse(HttpContext context, HealthReport result)
{
    context.Response.ContentType = "application/json; charset=utf-8";

    var options = new JsonWriterOptions
    {
        Indented = true
    };

    using (var stream = new MemoryStream())
    {
        using (var writer = new Utf8JsonWriter(stream, options))
        {
            writer.WriteStartObject();
            writer.WriteString("status", result.Status.ToString());
            writer.WriteStartObject("results");
            foreach (var entry in result.Entries)
            {
                writer.WriteStartObject(entry.Key);
                writer.WriteString("status", entry.Value.Status.ToString());
                writer.WriteString("description", entry.Value.Description);
                writer.WriteStartObject("data");
                foreach (var item in entry.Value.Data)
                {
                    writer.WritePropertyName(item.Key);
                    JsonSerializer.Serialize(
                        writer, item.Value, item.Value?.GetType() ??
                        typeof(object));
                }
                writer.WriteEndObject();
                writer.WriteEndObject();
            }
            writer.WriteEndObject();
            writer.WriteEndObject();
        }

        var json = Encoding.UTF8.GetString(stream.ToArray());

        return context.Response.WriteAsync(json);
    }
}
```

The second example demonstrates how to use Newtonsoft.Json:

```
private static Task WriteResponse(HttpContext context, HealthReport result)
{
    context.Response.ContentType = "application/json";

    var json = new JObject(
        new JProperty("status", result.Status.ToString()),
        new JProperty("results", new JObject(result.Entries.Select(pair =>
            new JProperty(pair.Key, new JObject(
                new JProperty("status", pair.Value.Status.ToString()),
                new JProperty("description", pair.Value.Description),
                new JProperty("data", new JObject(pair.Value.Data.Select(
                    p => new JProperty(p.Key, p.Value)))))))));

    return context.Response.WriteAsync(
        json.ToString(Formatting.Indented));
}
```

In the sample app, comment out the `SYSTEM_TEXT_JSON` preprocessor directive in *CustomWriterStartup.cs* to enable the `Newtonsoft.Json` version of `WriteResponse`.

The health checks API doesn't provide built-in support for complex JSON return formats because the format is specific to your choice of monitoring system. Customize the response in the preceding examples as needed. For more information on JSON serialization with `System.Text.Json`, see How to serialize and deserialize JSON in .NET.

Database probe

A health check can specify a database query to run as a boolean test to indicate if the database is responding normally.

The sample app uses AspNetCore.Diagnostics.HealthChecks, a health check library for ASP.NET Core apps, to perform a health check on a SQL Server database. `AspNetCore.Diagnostics.HealthChecks` executes a `SELECT 1` query against the database to confirm the connection to the database is healthy.

> **WARNING**
> When checking a database connection with a query, choose a query that returns quickly. The query approach runs the risk of overloading the database and degrading its performance. In most cases, running a test query isn't necessary. Merely making a successful connection to the database is sufficient. If you find it necessary to run a query, choose a simple SELECT query, such as `SELECT 1`.

Include a package reference to AspNetCore.HealthChecks.SqlServer.

Supply a valid database connection string in the *appsettings.json* file of the sample app. The app uses a SQL Server database named `HealthCheckSample`:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\MSSQLLocalDB;Database=HealthCheckSample;Trusted_Connection=True;MultipleActiveResultSets=true;Con
nectRetryCount=0"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    },
    "Console": {
      "IncludeScopes": "true"
    }
  },
  "AllowedHosts": "*"
}
```

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. The sample app calls the `AddSqlServer` method with the database's connection string (*DbHealthStartup.cs*):

```
services.AddHealthChecks()
    .AddSqlServer(Configuration["ConnectionStrings:DefaultConnection"]);
```

A health check endpoint is created by calling `MapHealthChecks` in `Startup.Configure`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
}
```

To run the database probe scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario db
```

> **NOTE**
> AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Entity Framework Core DbContext probe

The `DbContext` check confirms that the app can communicate with the database configured for an EF Core `DbContext`. The `DbContext` check is supported in apps that:

- Use Entity Framework (EF) Core.
- Include a package reference to Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore.

`AddDbContextCheck<TContext>` registers a health check for a `DbContext`. The `DbContext` is supplied as the `TContext` to the method. An overload is available to configure the failure status, tags, and a custom test query.

By default:

- The `DbContextHealthCheck` calls EF Core's `CanConnectAsync` method. You can customize what operation is run when checking health using `AddDbContextCheck` method overloads.
- The name of the health check is the name of the `TContext` type.

In the sample app, `AppDbContext` is provided to `AddDbContextCheck` and registered as a service in `Startup.ConfigureServices` (*DbContextHealthStartup.cs*):

```
services.AddHealthChecks()
    .AddDbContextCheck<AppDbContext>();

services.AddDbContext<AppDbContext>(options =>
{
    options.UseSqlServer(
        Configuration["ConnectionStrings:DefaultConnection"]);
});
```

A health check endpoint is created by calling `MapHealthChecks` in `Startup.Configure`:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
}
```

To run the `DbContext` probe scenario using the sample app, confirm that the database specified by the connection string doesn't exist in the SQL Server instance. If the database exists, delete it.

Execute the following command from the project's folder in a command shell:

```
dotnet run --scenario dbcontext
```

After the app is running, check the health status by making a request to the `/health` endpoint in a browser. The database and `AppDbContext` don't exist, so app provides the following response:

```
Unhealthy
```

Trigger the sample app to create the database. Make a request to `/createdatabase`. The app responds:

```
Creating the database...
Done!
Navigate to /health to see the health status.
```

Make a request to the `/health` endpoint. The database and context exist, so app responds:

```
Healthy
```

Trigger the sample app to delete the database. Make a request to `/deletedatabase`. The app responds:

```
Deleting the database...
Done!
Navigate to /health to see the health status.
```

Make a request to the `/health` endpoint. The app provides an unhealthy response:

```
Unhealthy
```

Separate readiness and liveness probes

In some hosting scenarios, a pair of health checks are used that distinguish two app states:

- *Readiness* indicates if the app is running normally but isn't ready to receive requests.
- *Liveness* indicates if an app has crashed and must be restarted.

Consider the following example: An app must download a large configuration file before it's ready to process requests. We don't want the app to be restarted if the initial download fails because the app can retry downloading the file several times. We use a *liveness probe* to describe the liveness of the process, no additional checks are performed. We also want to prevent requests from being sent to the app before the configuration file download has succeeded. We use a *readiness probe* to indicate a "not ready" state until the download succeeds and the app is ready to receive requests.

The sample app contains a health check to report the completion of long-running startup task in a Hosted Service. The `StartupHostedServiceHealthCheck` exposes a property, `StartupTaskCompleted`, that the hosted service can set to `true` when its long-running task is finished (*StartupHostedServiceHealthCheck.cs*):

```
public class StartupHostedServiceHealthCheck : IHealthCheck
{
    private volatile bool _startupTaskCompleted = false;

    public string Name => "slow_dependency_check";

    public bool StartupTaskCompleted
    {
        get => _startupTaskCompleted;
        set => _startupTaskCompleted = value;
    }

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        if (StartupTaskCompleted)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("The startup task is finished."));
        }

        return Task.FromResult(
            HealthCheckResult.Unhealthy("The startup task is still running."));
    }
}
```

The long-running background task is started by a Hosted Service (*Services/StartupHostedService*). At the conclusion of the task, `StartupHostedServiceHealthCheck.StartupTaskCompleted` is set to `true`:

```
public class StartupHostedService : IHostedService, IDisposable
{
    private readonly int _delaySeconds = 15;
    private readonly ILogger _logger;
    private readonly StartupHostedServiceHealthCheck _startupHostedServiceHealthCheck;

    public StartupHostedService(ILogger<StartupHostedService> logger,
        StartupHostedServiceHealthCheck startupHostedServiceHealthCheck)
    {
        _logger = logger;
        _startupHostedServiceHealthCheck = startupHostedServiceHealthCheck;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Startup Background Service is starting.");

        // Simulate the effect of a long-running startup task.
        Task.Run(async () =>
        {
            await Task.Delay(_delaySeconds * 1000);

            _startupHostedServiceHealthCheck.StartupTaskCompleted = true;

            _logger.LogInformation("Startup Background Service has started.");
        });

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Startup Background Service is stopping.");

        return Task.CompletedTask;
    }

    public void Dispose()
    {
    }
}
```

The health check is registered with AddCheck in `Startup.ConfigureServices` along with the hosted service. Because the hosted service must set the property on the health check, the health check is also registered in the service container (*LivenessProbeStartup.cs*):

```
services.AddHostedService<StartupHostedService>();
services.AddSingleton<StartupHostedServiceHealthCheck>();

services.AddHealthChecks()
    .AddCheck<StartupHostedServiceHealthCheck>(
        "hosted_service_startup",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "ready" });

services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.FromSeconds(2);
    options.Predicate = (check) => check.Tags.Contains("ready");
});

services.AddSingleton<IHealthCheckPublisher, ReadinessPublisher>();
```

A health check endpoint is created by calling `MapHealthChecks` in `Startup.Configure`. In the sample app, the

health check endpoints are created at:

- `/health/ready` for the readiness check. The readiness check filters health checks to the health check with the `ready` tag.
- `/health/live` for the liveness check. The liveness check filters out the `StartupHostedServiceHealthCheck` by returning `false` in the HealthCheckOptions.Predicate (for more information, see Filter health checks)

In the following example code:

- The readiness check uses all registered checks with the 'ready' tag.
- The `Predicate` excludes all checks and return a 200-Ok.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health/ready", new HealthCheckOptions()
    {
        Predicate = (check) => check.Tags.Contains("ready"),
    });

    endpoints.MapHealthChecks("/health/live", new HealthCheckOptions()
    {
        Predicate = (_) => false
    });
}
```

To run the readiness/liveness configuration scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario liveness
```

In a browser, visit `/health/ready` several times until 15 seconds have passed. The health check reports *Unhealthy* for the first 15 seconds. After 15 seconds, the endpoint reports *Healthy*, which reflects the completion of the long-running task by the hosted service.

This example also creates a Health Check Publisher (IHealthCheckPublisher implementation) that runs the first readiness check with a two second delay. For more information, see the Health Check Publisher section.

Kubernetes example

Using separate readiness and liveness checks is useful in an environment such as Kubernetes. In Kubernetes, an app might be required to perform time-consuming startup work before accepting requests, such as a test of the underlying database availability. Using separate checks allows the orchestrator to distinguish whether the app is functioning but not yet ready or if the app has failed to start. For more information on readiness and liveness probes in Kubernetes, see Configure Liveness and Readiness Probes in the Kubernetes documentation.

The following example demonstrates a Kubernetes readiness probe configuration:

```
spec:
  template:
  spec:
    readinessProbe:
      # an http probe
      httpGet:
        path: /health/ready
        port: 80
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
      - containerPort: 80
```

Metric-based probe with a custom response writer

The sample app demonstrates a memory health check with a custom response writer.

`MemoryHealthCheck` reports a degraded status if the app uses more than a given threshold of memory (1 GB in the sample app). The HealthCheckResult includes Garbage Collector (GC) information for the app (*MemoryHealthCheck.cs*):

```
public class MemoryHealthCheck : IHealthCheck
{
    private readonly IOptionsMonitor<MemoryCheckOptions> _options;

    public MemoryHealthCheck(IOptionsMonitor<MemoryCheckOptions> options)
    {
        _options = options;
    }

    public string Name => "memory_check";

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        var options = _options.Get(context.Registration.Name);

        // Include GC information in the reported diagnostics.
        var allocated = GC.GetTotalMemory(forceFullCollection: false);
        var data = new Dictionary<string, object>()
        {
            { "AllocatedBytes", allocated },
            { "Gen0Collections", GC.CollectionCount(0) },
            { "Gen1Collections", GC.CollectionCount(1) },
            { "Gen2Collections", GC.CollectionCount(2) },
        };
        var status = (allocated < options.Threshold) ?
            HealthStatus.Healthy : context.Registration.FailureStatus;

        return Task.FromResult(new HealthCheckResult(
            status,
            description: "Reports degraded status if allocated bytes " +
                $">= {options.Threshold} bytes.",
            exception: null,
            data: data));
    }
}
```

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. Instead of enabling the

health check by passing it to AddCheck, the `MemoryHealthCheck` is registered as a service. All IHealthCheck registered services are available to the health check services and middleware. We recommend registering health check services as Singleton services.

In *CustomWriterStartup.cs* of the sample app:

```
services.AddHealthChecks()
    .AddMemoryHealthCheck("memory");
```

A health check endpoint is created by calling `MapHealthChecks` in `Startup.Configure`. A `WriteResponse` delegate is provided to the <Microsoft.AspNetCore.Diagnostics.HealthChecks.HealthCheckOptions.ResponseWriter> property to output a custom JSON response when the health check executes:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health", new HealthCheckOptions()
    {
        ResponseWriter = WriteResponse
    });
}
```

The `WriteResponse` delegate formats the `CompositeHealthCheckResult` into a JSON object and yields JSON output for the health check response. For more information, see the Customize output section.

To run the metric-based probe with custom response writer output using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario writer
```

> **NOTE**
>
> AspNetCore.Diagnostics.HealthChecks includes metric-based health check scenarios, including disk storage and maximum value liveness checks.
>
> AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Filter by port

Call `RequireHost` on `MapHealthChecks` with a URL pattern that specifies a port to restrict health check requests to the port specified. This is typically used in a container environment to expose a port for monitoring services.

The sample app configures the port using the Environment Variable Configuration Provider. The port is set in the *launchSettings.json* file and passed to the configuration provider via an environment variable. You must also configure the server to listen to requests on the management port.

To use the sample app to demonstrate management port configuration, create the *launchSettings.json* file in a *Properties* folder.

The following *Properties/launchSettings.json* file in the sample app isn't included in the sample app's project files and must be created manually:

```
{
  "profiles": {
    "SampleApp": {
      "commandName": "Project",
      "commandLineArgs": "",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_URLS": "http://localhost:5000/;http://localhost:5001/",
        "ASPNETCORE_MANAGEMENTPORT": "5001"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. Create a health check endpoint by calling `MapHealthChecks` in `Startup.Configure`.

In the sample app, a call to `RequireHost` on the endpoint in `Startup.Configure` specifies the management port from configuration:

```
endpoints.MapHealthChecks("/health")
    .RequireHost($"*:{Configuration["ManagementPort"]}");
```

Endpoints are created in the sample app in `Startup.Configure`. In the following example code:

- The readiness check uses all registered checks with the 'ready' tag.
- The `Predicate` excludes all checks and return a 200-Ok.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health/ready", new HealthCheckOptions()
    {
        Predicate = (check) => check.Tags.Contains("ready"),
    });

    endpoints.MapHealthChecks("/health/live", new HealthCheckOptions()
    {
        Predicate = (_) => false
    });
}
```

NOTE

You can avoid creating the *launchSettings.json* file in the sample app by setting the management port explicitly in code. In *Program.cs* where the HostBuilder is created, add a call to ListenAnyIP and provide the app's management port endpoint. In `Configure` of *ManagementPortStartup.cs*, specify the management port with `RequireHost`:

Program.cs:

```
return new HostBuilder()
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseKestrel()
            .ConfigureKestrel(serverOptions =>
            {
                serverOptions.ListenAnyIP(5001);
            })
            .UseStartup(startupType);
    })
    .Build();
```

ManagementPortStartup.cs:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health").RequireHost("*:5001");
});
```

To run the management port configuration scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario port
```

Distribute a health check library

To distribute a health check as a library:

1. Write a health check that implements the IHealthCheck interface as a standalone class. The class can rely on dependency injection (DI), type activation, and named options to access configuration data.

   In the health checks logic of `CheckHealthAsync`:

   - `data1` and `data2` are used in the method to run the probe's health check logic.
   - `AccessViolationException` is handled.

   When an AccessViolationException occurs, the FailureStatus is returned with the HealthCheckResult to allow users to configure the health checks failure status.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

namespace SampleApp
{
    public class ExampleHealthCheck : IHealthCheck
    {
        private readonly string _data1;
        private readonly int? _data2;

        public ExampleHealthCheck(string data1, int? data2)
        {
            _data1 = data1 ?? throw new ArgumentNullException(nameof(data1));
            _data2 = data2 ?? throw new ArgumentNullException(nameof(data2));
        }

        public async Task<HealthCheckResult> CheckHealthAsync(
            HealthCheckContext context, CancellationToken cancellationToken)
        {
            try
            {
                return HealthCheckResult.Healthy();
            }
            catch (AccessViolationException ex)
            {
                return new HealthCheckResult(
                    context.Registration.FailureStatus,
                    description: "An access violation occurred during the check.",
                    exception: ex,
                    data: null);
            }
        }
    }
}
```

2. Write an extension method with parameters that the consuming app calls in its `Startup.Configure` method. In the following example, assume the following health check method signature:

```
ExampleHealthCheck(string, string, int )
```

The preceding signature indicates that the `ExampleHealthCheck` requires additional data to process the health check probe logic. The data is provided to the delegate used to create the health check instance when the health check is registered with an extension method. In the following example, the caller specifies optional:

- health check name (`name`). If `null` , `example_health_check` is used.
- string data point for the health check (`data1`).
- integer data point for the health check (`data2`). If `null` , `1` is used.
- failure status (HealthStatus). The default is `null` . If `null` , HealthStatus.Unhealthy is reported for a failure status.
- tags (`IEnumerable<string>`).

```
using System.Collections.Generic;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public static class ExampleHealthCheckBuilderExtensions
{
    const string DefaultName = "example_health_check";

    public static IHealthChecksBuilder AddExampleHealthCheck(
        this IHealthChecksBuilder builder,
        string name = default,
        string data1,
        int data2 = 1,
        HealthStatus? failureStatus = default,
        IEnumerable<string> tags = default)
    {
        return builder.Add(new HealthCheckRegistration(
            name ?? DefaultName,
            sp => new ExampleHealthCheck(data1, data2),
            failureStatus,
            tags));
    }
}
```

Health Check Publisher

When an IHealthCheckPublisher is added to the service container, the health check system periodically executes your health checks and calls `PublishAsync` with the result. This is useful in a push-based health monitoring system scenario that expects each process to call the monitoring system periodically in order to determine health.

The IHealthCheckPublisher interface has a single method:

```
Task PublishAsync(HealthReport report, CancellationToken cancellationToken);
```

HealthCheckPublisherOptions allow you to set:

- Delay: The initial delay applied after the app starts before executing IHealthCheckPublisher instances. The delay is applied once at startup and doesn't apply to subsequent iterations. The default value is five seconds.
- Period: The period of IHealthCheckPublisher execution. The default value is 30 seconds.
- Predicate: If Predicate is `null` (default), the health check publisher service runs all registered health checks. To run a subset of health checks, provide a function that filters the set of checks. The predicate is evaluated each period.
- Timeout: The timeout for executing the health checks for all IHealthCheckPublisher instances. Use InfiniteTimeSpan to execute without a timeout. The default value is 30 seconds.

In the sample app, `ReadinessPublisher` is an IHealthCheckPublisher implementation. The health check status is logged for each check at a log level of:

- Information (LogInformation) if the health checks status is Healthy.
- Error (LogError) if the status is either Degraded or Unhealthy.

```
public class ReadinessPublisher : IHealthCheckPublisher
{
    private readonly ILogger _logger;

    public ReadinessPublisher(ILogger<ReadinessPublisher> logger)
    {
        _logger = logger;
    }

    // The following example is for demonstration purposes only. Health Checks
    // Middleware already logs health checks results. A real-world readiness
    // check in a production app might perform a set of more expensive or
    // time-consuming checks to determine if other resources are responding
    // properly.
    public Task PublishAsync(HealthReport report,
        CancellationToken cancellationToken)
    {
        if (report.Status == HealthStatus.Healthy)
        {
            _logger.LogInformation("{Timestamp} Readiness Probe Status: {Result}",
                DateTime.UtcNow, report.Status);
        }
        else
        {
            _logger.LogError("{Timestamp} Readiness Probe Status: {Result}",
                DateTime.UtcNow, report.Status);
        }

        cancellationToken.ThrowIfCancellationRequested();

        return Task.CompletedTask;
    }
}
```

In the sample app's `LivenessProbeStartup` example, the `StartupHostedService` readiness check has a two second startup delay and runs the check every 30 seconds. To activate the IHealthCheckPublisher implementation, the sample registers `ReadinessPublisher` as a singleton service in the dependency injection (DI) container:

```
services.AddHostedService<StartupHostedService>();
services.AddSingleton<StartupHostedServiceHealthCheck>();

services.AddHealthChecks()
    .AddCheck<StartupHostedServiceHealthCheck>(
        "hosted_service_startup",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "ready" });

services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.FromSeconds(2);
    options.Predicate = (check) => check.Tags.Contains("ready");
});

services.AddSingleton<IHealthCheckPublisher, ReadinessPublisher>();
```

NOTE

AspNetCore.Diagnostics.HealthChecks includes publishers for several systems, including Application Insights.

AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Restrict health checks with MapWhen

Use MapWhen to conditionally branch the request pipeline for health check endpoints.

In the following example, `MapWhen` branches the request pipeline to activate Health Checks Middleware if a GET request is received for the `api/HealthCheck` endpoint:

```
app.MapWhen(
    context => context.Request.Method == HttpMethod.Get.Method &&
        context.Request.Path.StartsWith("/api/HealthCheck"),
    builder => builder.UseHealthChecks());

app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
});
```

For more information, see ASP.NET Core Middleware.

ASP.NET Core offers Health Checks Middleware and libraries for reporting the health of app infrastructure components.

Health checks are exposed by an app as HTTP endpoints. Health check endpoints can be configured for a variety of real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, a container orchestrator may respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

View or download sample code (how to download)

The sample app includes examples of the scenarios described in this topic. To run the sample app for a given scenario, use the dotnet run command from the project's folder in a command shell. See the sample app's *README.md* file and the scenario descriptions in this topic for details on how to use the sample app.

Prerequisites

Health checks are usually used with an external monitoring service or container orchestrator to check the status of an app. Before adding health checks to an app, decide on which monitoring system to use. The monitoring system dictates what types of health checks to create and how to configure their endpoints.

Reference the Microsoft.AspNetCore.App metapackage or add a package reference to the Microsoft.AspNetCore.Diagnostics.HealthChecks package.

The sample app provides startup code to demonstrate health checks for several scenarios. The database probe scenario checks the health of a database connection using AspNetCore.Diagnostics.HealthChecks. The DbContext probe scenario checks a database using an EF Core `DbContext`. To explore the database scenarios, the sample app:

- Creates a database and provides its connection string in the *appsettings.json* file.
- Has the following package references in its project file:
  - AspNetCore.HealthChecks.SqlServer
  - Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore

> **NOTE**
> AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Another health check scenario demonstrates how to filter health checks to a management port. The sample app requires you to create a *Properties/launchSettings.json* file that includes the management URL and management port. For more information, see the Filter by port section.

Basic health probe

For many apps, a basic health probe configuration that reports the app's availability to process requests (*liveness*) is sufficient to discover the status of the app.

The basic configuration registers health check services and calls the Health Checks Middleware to respond at a URL endpoint with a health response. By default, no specific health checks are registered to test any particular dependency or subsystem. The app is considered healthy if it's capable of responding at the health endpoint URL. The default response writer writes the status (HealthStatus) as a plaintext response back to the client, indicating either a HealthStatus.Healthy, HealthStatus.Degraded or HealthStatus.Unhealthy status.

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. Add an endpoint for Health Checks Middleware with UseHealthChecks in the request processing pipeline of `Startup.Configure`.

In the sample app, the health check endpoint is created at `/health` (*BasicStartup.cs*):

```
public class BasicStartup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHealthChecks();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHealthChecks("/health");
    }
}
```

To run the basic configuration scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario basic
```

Docker example

Docker offers a built-in `HEALTHCHECK` directive that can be used to check the status of an app that uses the basic health check configuration:

```
HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit
```

Create health checks

Health checks are created by implementing the IHealthCheck interface. The CheckHealthAsync method returns a HealthCheckResult that indicates the health as `Healthy`, `Degraded`, or `Unhealthy`. The result is written as a plaintext response with a configurable status code (configuration is described in the Health check options

section). HealthCheckResult can also return optional key-value pairs.

Example health check

The following `ExampleHealthCheck` class demonstrates the layout of a health check. The health checks logic is placed in the `CheckHealthAsync` method. The following example sets a dummy variable, `healthCheckResultHealthy`, to `true`. If the value of `healthCheckResultHealthy` is set to `false`, the HealthCheckResult.Unhealthy status is returned.

```
public class ExampleHealthCheck : IHealthCheck
{
    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        var healthCheckResultHealthy = true;

        if (healthCheckResultHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("The check indicates a healthy result."));
        }

        return Task.FromResult(
            HealthCheckResult.Unhealthy("The check indicates an unhealthy result."));
    }
}
```

Register health check services

The `ExampleHealthCheck` type is added to health check services in `Startup.ConfigureServices` with AddCheck:

```
services.AddHealthChecks()
    .AddCheck<ExampleHealthCheck>("example_health_check");
```

The AddCheck overload shown in the following example sets the failure status (HealthStatus) to report when the health check reports a failure. If the failure status is set to `null` (default), HealthStatus.Unhealthy is reported. This overload is a useful scenario for library authors, where the failure status indicated by the library is enforced by the app when a health check failure occurs if the health check implementation honors the setting.

Tags can be used to filter health checks (described further in the Filter health checks section).

```
services.AddHealthChecks()
    .AddCheck<ExampleHealthCheck>(
        "example_health_check",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "example" });
```

AddCheck can also execute a lambda function. In the following `Startup.ConfigureServices` example, the health check name is specified as `Example` and the check always returns a healthy state:

```
services.AddHealthChecks()
    .AddCheck("Example", () =>
        HealthCheckResult.Healthy("Example is OK!"), tags: new[] { "example" });
```

Use Health Checks Middleware

In `Startup.Configure`, call UseHealthChecks in the processing pipeline with the endpoint URL or relative path:

```
app.UseHealthChecks("/health");
```

If the health checks should listen on a specific port, use an overload of UseHealthChecks to set the port (described further in the Filter by port section):

```
app.UseHealthChecks("/health", port: 8000);
```

Health check options

HealthCheckOptions provide an opportunity to customize health check behavior:

- Filter health checks
- Customize the HTTP status code
- Suppress cache headers
- Customize output

Filter health checks

By default, Health Checks Middleware runs all registered health checks. To run a subset of health checks, provide a function that returns a boolean to the Predicate option. In the following example, the `Bar` health check is filtered out by its tag (`bar_tag`) in the function's conditional statement, where `true` is only returned if the health check's Tags property matches `foo_tag` or `baz_tag`:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Diagnostics.HealthChecks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddCheck("Foo", () =>
            HealthCheckResult.Healthy("Foo is OK!"), tags: new[] { "foo_tag" })
        .AddCheck("Bar", () =>
            HealthCheckResult.Unhealthy("Bar is unhealthy!"),
                tags: new[] { "bar_tag" })
        .AddCheck("Baz", () =>
            HealthCheckResult.Healthy("Baz is OK!"), tags: new[] { "baz_tag" });
}

public void Configure(IApplicationBuilder app)
{
    app.UseHealthChecks("/health", new HealthCheckOptions()
    {
        Predicate = (check) => check.Tags.Contains("foo_tag") ||
            check.Tags.Contains("baz_tag")
    });
}
```

Customize the HTTP status code

Use ResultStatusCodes to customize the mapping of health status to HTTP status codes. The following StatusCodes assignments are the default values used by the middleware. Change the status code values to meet your requirements.

In `Startup.Configure`:

```
//using Microsoft.AspNetCore.Diagnostics.HealthChecks;
//using Microsoft.Extensions.Diagnostics.HealthChecks;

app.UseHealthChecks("/health", new HealthCheckOptions()
{
    ResultStatusCodes =
    {
        [HealthStatus.Healthy] = StatusCodes.Status200OK,
        [HealthStatus.Degraded] = StatusCodes.Status200OK,
        [HealthStatus.Unhealthy] = StatusCodes.Status503ServiceUnavailable
    }
});
```

Suppress cache headers

AllowCachingResponses controls whether the Health Checks Middleware adds HTTP headers to a probe response to prevent response caching. If the value is `false` (default), the middleware sets or overrides the `Cache-Control`, `Expires`, and `Pragma` headers to prevent response caching. If the value is `true`, the middleware doesn't modify the cache headers of the response.

In `Startup.Configure`:

```
//using Microsoft.AspNetCore.Diagnostics.HealthChecks;
//using Microsoft.Extensions.Diagnostics.HealthChecks;

app.UseHealthChecks("/health", new HealthCheckOptions()
{
    AllowCachingResponses = false
});
```

Customize output

The ResponseWriter option gets or sets a delegate used to write the response. The default delegate writes a minimal plaintext response with the string value of HealthReport.Status.

In `Startup.Configure`:

```
// using Microsoft.AspNetCore.Diagnostics.HealthChecks;
// using Microsoft.Extensions.Diagnostics.HealthChecks;

app.UseHealthChecks("/health", new HealthCheckOptions()
{
    ResponseWriter = WriteResponse
});
```

The default delegate writes a minimal plaintext response with the string value of HealthReport.Status. The following custom delegate, `WriteResponse`, outputs a custom JSON response:

```
private static Task WriteResponse(HttpContext httpContext, HealthReport result)
{
    httpContext.Response.ContentType = "application/json";

    var json = new JObject(
        new JProperty("status", result.Status.ToString()),
        new JProperty("results", new JObject(result.Entries.Select(pair =>
            new JProperty(pair.Key, new JObject(
                new JProperty("status", pair.Value.Status.ToString()),
                new JProperty("description", pair.Value.Description),
                new JProperty("data", new JObject(pair.Value.Data.Select(
                    p => new JProperty(p.Key, p.Value)))))))));
    return httpContext.Response.WriteAsync(
        json.ToString(Formatting.Indented));
}
```

The health checks system doesn't provide built-in support for complex JSON return formats because the format is specific to your choice of monitoring system. Feel free to customize the `JObject` in the preceding example as necessary to meet your needs.

Database probe

A health check can specify a database query to run as a boolean test to indicate if the database is responding normally.

The sample app uses AspNetCore.Diagnostics.HealthChecks, a health check library for ASP.NET Core apps, to perform a health check on a SQL Server database. `AspNetCore.Diagnostics.HealthChecks` executes a `SELECT 1` query against the database to confirm the connection to the database is healthy.

> **WARNING**
> When checking a database connection with a query, choose a query that returns quickly. The query approach runs the risk of overloading the database and degrading its performance. In most cases, running a test query isn't necessary. Merely making a successful connection to the database is sufficient. If you find it necessary to run a query, choose a simple SELECT query, such as `SELECT 1`.

Include a package reference to AspNetCore.HealthChecks.SqlServer.

Supply a valid database connection string in the *appsettings.json* file of the sample app. The app uses a SQL Server database named `HealthCheckSample`:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\MSSQLLocalDB;Database=HealthCheckSample;Trusted_Connection=True;MultipleActiveResultSets=true;Con
nectRetryCount=0"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Debug"
    },
    "Console": {
      "IncludeScopes": "true"
    }
  }
}
```

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. The sample app calls the

`AddSqlServer` method with the database's connection string (*DbHealthStartup.cs*):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddSqlServer(Configuration["ConnectionStrings:DefaultConnection"]);
}
```

Call Health Checks Middleware in the app processing pipeline in `Startup.Configure`:

```
app.UseHealthChecks("/health");
```

To run the database probe scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario db
```

> **NOTE**
> AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Entity Framework Core DbContext probe

The `DbContext` check confirms that the app can communicate with the database configured for an EF Core `DbContext`. The `DbContext` check is supported in apps that:

- Use Entity Framework (EF) Core.
- Include a package reference to Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore.

`AddDbContextCheck<TContext>` registers a health check for a `DbContext`. The `DbContext` is supplied as the `TContext` to the method. An overload is available to configure the failure status, tags, and a custom test query.

By default:

- The `DbContextHealthCheck` calls EF Core's `CanConnectAsync` method. You can customize what operation is run when checking health using `AddDbContextCheck` method overloads.
- The name of the health check is the name of the `TContext` type.

In the sample app, `AppDbContext` is provided to `AddDbContextCheck` and registered as a service in `Startup.ConfigureServices` (*DbContextHealthStartup.cs*):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddDbContextCheck<AppDbContext>();

    services.AddDbContext<AppDbContext>(options =>
    {
        options.UseSqlServer(
            Configuration["ConnectionStrings:DefaultConnection"]);
    });
}
```

In the sample app, `UseHealthChecks` adds the Health Checks Middleware in `Startup.Configure`.

```
app.UseHealthChecks("/health");
```

To run the `DbContext` probe scenario using the sample app, confirm that the database specified by the connection string doesn't exist in the SQL Server instance. If the database exists, delete it.

Execute the following command from the project's folder in a command shell:

```
dotnet run --scenario dbcontext
```

After the app is running, check the health status by making a request to the `/health` endpoint in a browser. The database and `AppDbContext` don't exist, so app provides the following response:

```
Unhealthy
```

Trigger the sample app to create the database. Make a request to `/createdatabase`. The app responds:

```
Creating the database...
Done!
Navigate to /health to see the health status.
```

Make a request to the `/health` endpoint. The database and context exist, so app responds:

```
Healthy
```

Trigger the sample app to delete the database. Make a request to `/deletedatabase`. The app responds:

```
Deleting the database...
Done!
Navigate to /health to see the health status.
```

Make a request to the `/health` endpoint. The app provides an unhealthy response:

```
Unhealthy
```

Separate readiness and liveness probes

In some hosting scenarios, a pair of health checks are used that distinguish two app states:

- *Readiness* indicates if the app is running normally but isn't ready to receive requests.
- *Liveness* indicates if an app has crashed and must be restarted.

Consider the following example: An app must download a large configuration file before it's ready to process requests. We don't want the app to be restarted if the initial download fails because the app can retry downloading the file several times. We use a *liveness probe* to describe the liveness of the process, no additional checks are performed. We also want to prevent requests from being sent to the app before the configuration file download has succeeded. We use a *readiness probe* to indicate a "not ready" state until the download succeeds and the app is ready to receive requests.

The sample app contains a health check to report the completion of long-running startup task in a Hosted Service. The `StartupHostedServiceHealthCheck` exposes a property, `StartupTaskCompleted`, that the hosted service

can set to `true` when its long-running task is finished (*StartupHostedServiceHealthCheck.cs*):

```
public class StartupHostedServiceHealthCheck : IHealthCheck
{
    private volatile bool _startupTaskCompleted = false;

    public string Name => "slow_dependency_check";

    public bool StartupTaskCompleted
    {
        get => _startupTaskCompleted;
        set => _startupTaskCompleted = value;
    }

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        if (StartupTaskCompleted)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("The startup task is finished."));
        }

        return Task.FromResult(
            HealthCheckResult.Unhealthy("The startup task is still running."));
    }
}
```

The long-running background task is started by a Hosted Service (*Services/StartupHostedService*). At the conclusion of the task, `StartupHostedServiceHealthCheck.StartupTaskCompleted` is set to `true`:

```
public class StartupHostedService : IHostedService, IDisposable
{
    private readonly int _delaySeconds = 15;
    private readonly ILogger _logger;
    private readonly StartupHostedServiceHealthCheck _startupHostedServiceHealthCheck;

    public StartupHostedService(ILogger<StartupHostedService> logger,
        StartupHostedServiceHealthCheck startupHostedServiceHealthCheck)
    {
        _logger = logger;
        _startupHostedServiceHealthCheck = startupHostedServiceHealthCheck;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Startup Background Service is starting.");

        // Simulate the effect of a long-running startup task.
        Task.Run(async () =>
        {
            await Task.Delay(_delaySeconds * 1000);

            _startupHostedServiceHealthCheck.StartupTaskCompleted = true;

            _logger.LogInformation("Startup Background Service has started.");
        });

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Startup Background Service is stopping.");

        return Task.CompletedTask;
    }

    public void Dispose()
    {
    }
}
```

The health check is registered with AddCheck in `Startup.ConfigureServices` along with the hosted service. Because the hosted service must set the property on the health check, the health check is also registered in the service container (*LivenessProbeStartup.cs*):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHostedService<StartupHostedService>();
    services.AddSingleton<StartupHostedServiceHealthCheck>();

    services.AddHealthChecks()
        .AddCheck<StartupHostedServiceHealthCheck>(
            "hosted_service_startup",
            failureStatus: HealthStatus.Degraded,
            tags: new[] { "ready" });

    services.Configure<HealthCheckPublisherOptions>(options =>
    {
        options.Delay = TimeSpan.FromSeconds(2);
        options.Predicate = (check) => check.Tags.Contains("ready");
    });

    // The following workaround permits adding an IHealthCheckPublisher
    // instance to the service container when one or more other hosted
    // services have already been added to the app. This workaround
    // won't be required with the release of ASP.NET Core 3.0. For more
    // information, see: https://github.com/aspnet/Extensions/issues/639.
    services.TryAddEnumerable(
        ServiceDescriptor.Singleton(typeof(IHostedService),
            typeof(HealthCheckPublisherOptions).Assembly
                .GetType(HealthCheckServiceAssembly)));

    services.AddSingleton<IHealthCheckPublisher, ReadinessPublisher>();
}
```

Call Health Checks Middleware in the app processing pipeline in `Startup.Configure`. In the sample app, the health check endpoints are created at `/health/ready` for the readiness check and `/health/live` for the liveness check. The readiness check filters health checks to the health check with the `ready` tag. The liveness check filters out the `StartupHostedServiceHealthCheck` by returning `false` in the HealthCheckOptions.Predicate (for more information, see Filter health checks):

```
app.UseHealthChecks("/health/ready", new HealthCheckOptions()
{
    Predicate = (check) => check.Tags.Contains("ready"),
});

app.UseHealthChecks("/health/live", new HealthCheckOptions()
{
    Predicate = (_) => false
});
```

To run the readiness/liveness configuration scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario liveness
```

In a browser, visit `/health/ready` several times until 15 seconds have passed. The health check reports *Unhealthy* for the first 15 seconds. After 15 seconds, the endpoint reports *Healthy*, which reflects the completion of the long-running task by the hosted service.

This example also creates a Health Check Publisher (IHealthCheckPublisher implementation) that runs the first readiness check with a two second delay. For more information, see the Health Check Publisher section.

Kubernetes example

Using separate readiness and liveness checks is useful in an environment such as Kubernetes. In Kubernetes, an app might be required to perform time-consuming startup work before accepting requests, such as a test of the underlying database availability. Using separate checks allows the orchestrator to distinguish whether the app is functioning but not yet ready or if the app has failed to start. For more information on readiness and liveness probes in Kubernetes, see Configure Liveness and Readiness Probes in the Kubernetes documentation.

The following example demonstrates a Kubernetes readiness probe configuration:

```
spec:
  template:
  spec:
    readinessProbe:
      # an http probe
      httpGet:
        path: /health/ready
        port: 80
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
      - containerPort: 80
```

Metric-based probe with a custom response writer

The sample app demonstrates a memory health check with a custom response writer.

`MemoryHealthCheck` reports an unhealthy status if the app uses more than a given threshold of memory (1 GB in the sample app). The HealthCheckResult includes Garbage Collector (GC) information for the app (*MemoryHealthCheck.cs*):

```
public class MemoryHealthCheck : IHealthCheck
{
    private readonly IOptionsMonitor<MemoryCheckOptions> _options;

    public MemoryHealthCheck(IOptionsMonitor<MemoryCheckOptions> options)
    {
        _options = options;
    }

    public string Name => "memory_check";

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default(CancellationToken))
    {
        var options = _options.Get(context.Registration.Name);

        // Include GC information in the reported diagnostics.
        var allocated = GC.GetTotalMemory(forceFullCollection: false);
        var data = new Dictionary<string, object>()
        {
            { "AllocatedBytes", allocated },
            { "Gen0Collections", GC.CollectionCount(0) },
            { "Gen1Collections", GC.CollectionCount(1) },
            { "Gen2Collections", GC.CollectionCount(2) },
        };

        var status = (allocated < options.Threshold) ?
            HealthStatus.Healthy : HealthStatus.Unhealthy;

        return Task.FromResult(new HealthCheckResult(
            status,
            description: "Reports degraded status if allocated bytes " +
                $">= {options.Threshold} bytes.",
            exception: null,
            data: data));
    }
}
```

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. Instead of enabling the health check by passing it to AddCheck, the `MemoryHealthCheck` is registered as a service. All IHealthCheck registered services are available to the health check services and middleware. We recommend registering health check services as Singleton services.

In the sample app (*CustomWriterStartup.cs*):

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHealthChecks()
        .AddMemoryHealthCheck("memory");
}
```

Call Health Checks Middleware in the app processing pipeline in `Startup.Configure`. A `WriteResponse` delegate is provided to the `ResponseWriter` property to output a custom JSON response when the health check executes:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHealthChecks("/health", new HealthCheckOptions()
    {
        // This custom writer formats the detailed status as JSON.
        ResponseWriter = WriteResponse
    });
}
```

The `WriteResponse` method formats the `CompositeHealthCheckResult` into a JSON object and yields JSON output for the health check response:

```
private static Task WriteResponse(HttpContext httpContext,
    HealthReport result)
{
    httpContext.Response.ContentType = "application/json; charset=utf-8";

    var json = new JObject(
        new JProperty("status", result.Status.ToString()),
        new JProperty("results", new JObject(result.Entries.Select(pair =>
            new JProperty(pair.Key, new JObject(
                new JProperty("status", pair.Value.Status.ToString()),
                new JProperty("description", pair.Value.Description),
                new JProperty("data", new JObject(pair.Value.Data.Select(
                    p => new JProperty(p.Key, p.Value)))))))));
    return httpContext.Response.WriteAsync(
        json.ToString(Formatting.Indented));
}
```

To run the metric-based probe with custom response writer output using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario writer
```

> **NOTE**
>
> AspNetCore.Diagnostics.HealthChecks includes metric-based health check scenarios, including disk storage and maximum value liveness checks.
>
> AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Filter by port

Calling UseHealthChecks with a port restricts health check requests to the port specified. This is typically used in a container environment to expose a port for monitoring services.

The sample app configures the port using the Environment Variable Configuration Provider. The port is set in the *launchSettings.json* file and passed to the configuration provider via an environment variable. You must also configure the server to listen to requests on the management port.

To use the sample app to demonstrate management port configuration, create the *launchSettings.json* file in a *Properties* folder.

The following *Properties/launchSettings.json* file in the sample app isn't included in the sample app's project files and must be created manually:

```
{
  "profiles": {
    "SampleApp": {
      "commandName": "Project",
      "commandLineArgs": "",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_URLS": "http://localhost:5000/;http://localhost:5001/",
        "ASPNETCORE_MANAGEMENTPORT": "5001"
      },
      "applicationUrl": "http://localhost:5000/"
    }
  }
}
```

Register health check services with AddHealthChecks in `Startup.ConfigureServices`. The call to UseHealthChecks specifies the management port (*ManagementPortStartup.cs*):

```
public class ManagementPortStartup
{
    public ManagementPortStartup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHealthChecks();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseHealthChecks("/health", port: Configuration["ManagementPort"]);

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                "Navigate to " +
                $"http://localhost:{Configuration["ManagementPort"]}/health " +
                "to see the health status.");
        });
    }
}
```

NOTE

You can avoid creating the *launchSettings.json* file in the sample app by setting the URLs and management port explicitly in code. In *Program.cs* where the WebHostBuilder is created, add a call to UseUrls and provide the app's normal response endpoint and the management port endpoint. In *ManagementPortStartup.cs* where UseHealthChecks is called, specify the management port explicitly.

Program.cs:

```
return new WebHostBuilder()
    .UseConfiguration(config)
    .UseUrls("http://localhost:5000/;http://localhost:5001/")
    .ConfigureLogging(builder =>
    {
        builder.SetMinimumLevel(LogLevel.Trace);
        builder.AddConfiguration(config);
        builder.AddConsole();
    })
    .UseKestrel()
    .UseStartup(startupType)
    .Build();
```

ManagementPortStartup.cs:

```
app.UseHealthChecks("/health", port: 5001);
```

To run the management port configuration scenario using the sample app, execute the following command from the project's folder in a command shell:

```
dotnet run --scenario port
```

Distribute a health check library

To distribute a health check as a library:

1. Write a health check that implements the IHealthCheck interface as a standalone class. The class can rely on dependency injection (DI), type activation, and named options to access configuration data.

   In the health checks logic of `CheckHealthAsync`:

   - `data1` and `data2` are used in the method to run the probe's health check logic.
   - `AccessViolationException` is handled.

   When an AccessViolationException occurs, the FailureStatus is returned with the HealthCheckResult to allow users to configure the health checks failure status.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public class ExampleHealthCheck : IHealthCheck
{
    private readonly string _data1;
    private readonly int? _data2;

    public ExampleHealthCheck(string data1, int? data2)
    {
        _data1 = data1 ?? throw new ArgumentNullException(nameof(data1));
        _data2 = data2 ?? throw new ArgumentNullException(nameof(data2));
    }

    public async Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken)
    {
        try
        {
            return HealthCheckResult.Healthy();
        }
        catch (AccessViolationException ex)
        {
            return new HealthCheckResult(
                context.Registration.FailureStatus,
                description: "An access violation occurred during the check.",
                exception: ex,
                data: null);
        }
    }
}
```

2. Write an extension method with parameters that the consuming app calls in its `Startup.Configure` method. In the following example, assume the following health check method signature:

```
ExampleHealthCheck(string, string, int )
```

The preceding signature indicates that the `ExampleHealthCheck` requires additional data to process the health check probe logic. The data is provided to the delegate used to create the health check instance when the health check is registered with an extension method. In the following example, the caller specifies optional:

- health check name (`name`). If `null` , `example_health_check` is used.
- string data point for the health check (`data1`).
- integer data point for the health check (`data2`). If `null` , `1` is used.
- failure status (HealthStatus). The default is `null` . If `null` , HealthStatus.Unhealthy is reported for a failure status.
- tags (`IEnumerable<string>`).

```
using System.Collections.Generic;
using Microsoft.Extensions.Diagnostics.HealthChecks;

public static class ExampleHealthCheckBuilderExtensions
{
    const string DefaultName = "example_health_check";

    public static IHealthChecksBuilder AddExampleHealthCheck(
        this IHealthChecksBuilder builder,
        string name = default,
        string data1,
        int data2 = 1,
        HealthStatus? failureStatus = default,
        IEnumerable<string> tags = default)
    {
        return builder.Add(new HealthCheckRegistration(
            name ?? DefaultName,
            sp => new ExampleHealthCheck(data1, data2),
            failureStatus,
            tags));
    }
}
```

Health Check Publisher

When an IHealthCheckPublisher is added to the service container, the health check system periodically executes your health checks and calls `PublishAsync` with the result. This is useful in a push-based health monitoring system scenario that expects each process to call the monitoring system periodically in order to determine health.

The IHealthCheckPublisher interface has a single method:

```
Task PublishAsync(HealthReport report, CancellationToken cancellationToken);
```

HealthCheckPublisherOptions allow you to set:

- Delay: The initial delay applied after the app starts before executing IHealthCheckPublisher instances. The delay is applied once at startup and doesn't apply to subsequent iterations. The default value is five seconds.
- Period: The period of IHealthCheckPublisher execution. The default value is 30 seconds.
- Predicate: If Predicate is `null` (default), the health check publisher service runs all registered health checks. To run a subset of health checks, provide a function that filters the set of checks. The predicate is evaluated each period.
- Timeout: The timeout for executing the health checks for all IHealthCheckPublisher instances. Use InfiniteTimeSpan to execute without a timeout. The default value is 30 seconds.

> **WARNING**
> In the ASP.NET Core 2.2 release, setting Period isn't honored by the IHealthCheckPublisher implementation; it sets the value of Delay. This issue has been addressed in ASP.NET Core 3.0.

In the sample app, `ReadinessPublisher` is an IHealthCheckPublisher implementation. The health check status is logged for each check as either:

- Information (LogInformation) if the health checks status is Healthy.
- Error (LogError) if the status is either Degraded or Unhealthy.

```
public class ReadinessPublisher : IHealthCheckPublisher
{
    private readonly ILogger _logger;

    public ReadinessPublisher(ILogger<ReadinessPublisher> logger)
    {
        _logger = logger;
    }

    // The following example is for demonstration purposes only. Health Checks
    // Middleware already logs health checks results. A real-world readiness
    // check in a production app might perform a set of more expensive or
    // time-consuming checks to determine if other resources are responding
    // properly.
    public Task PublishAsync(HealthReport report,
        CancellationToken cancellationToken)
    {
        if (report.Status == HealthStatus.Healthy)
        {
            _logger.LogInformation("{Timestamp} Readiness Probe Status: {Result}",
                DateTime.UtcNow, report.Status);
        }
        else
        {
            _logger.LogError("{Timestamp} Readiness Probe Status: {Result}",
                DateTime.UtcNow, report.Status);
        }

        cancellationToken.ThrowIfCancellationRequested();

        return Task.CompletedTask;
    }
}
```

In the sample app's `LivenessProbeStartup` example, the `StartupHostedService` readiness check has a two second startup delay and runs the check every 30 seconds. To activate the IHealthCheckPublisher implementation, the sample registers `ReadinessPublisher` as a singleton service in the dependency injection (DI) container:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHostedService<StartupHostedService>();
    services.AddSingleton<StartupHostedServiceHealthCheck>();

    services.AddHealthChecks()
        .AddCheck<StartupHostedServiceHealthCheck>(
            "hosted_service_startup",
            failureStatus: HealthStatus.Degraded,
            tags: new[] { "ready" });

    services.Configure<HealthCheckPublisherOptions>(options =>
    {
        options.Delay = TimeSpan.FromSeconds(2);
        options.Predicate = (check) => check.Tags.Contains("ready");
    });

    // The following workaround permits adding an IHealthCheckPublisher
    // instance to the service container when one or more other hosted
    // services have already been added to the app. This workaround
    // won't be required with the release of ASP.NET Core 3.0. For more
    // information, see: https://github.com/aspnet/Extensions/issues/639.
    services.TryAddEnumerable(
        ServiceDescriptor.Singleton(typeof(IHostedService),
            typeof(HealthCheckPublisherOptions).Assembly
                .GetType(HealthCheckServiceAssembly)));

    services.AddSingleton<IHealthCheckPublisher, ReadinessPublisher>();
}
```

NOTE

The following workaround permits adding an IHealthCheckPublisher instance to the service container when one or more other hosted services have already been added to the app. This workaround won't isn't required in ASP.NET Core 3.0.

```
private const string HealthCheckServiceAssembly =
    "Microsoft.Extensions.Diagnostics.HealthChecks.HealthCheckPublisherHostedService";

services.TryAddEnumerable(
    ServiceDescriptor.Singleton(typeof(IHostedService),
        typeof(HealthCheckPublisherOptions).Assembly
            .GetType(HealthCheckServiceAssembly)));
```

NOTE

AspNetCore.Diagnostics.HealthChecks includes publishers for several systems, including Application Insights.

AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Restrict health checks with MapWhen

Use MapWhen to conditionally branch the request pipeline for health check endpoints.

In the following example, `MapWhen` branches the request pipeline to activate Health Checks Middleware if a GET request is received for the `api/HealthCheck` endpoint:

```
app.MapWhen(
    context => context.Request.Method == HttpMethod.Get.Method &&
        context.Request.Path.StartsWith("/api/HealthCheck"),
    builder => builder.UseHealthChecks());

app.UseMvc();
```

For more information, see ASP.NET Core Middleware.

Overview of ASP.NET Core Security

9/22/2020 • 2 minutes to read • Edit Online

ASP.NET Core enables developers to easily configure and manage security for their apps. ASP.NET Core contains features for managing authentication, authorization, data protection, HTTPS enforcement, app secrets, XSRF/CSRF prevention, and CORS management. These security features allow you to build robust yet secure ASP.NET Core apps.

ASP.NET Core security features

ASP.NET Core provides many tools and libraries to secure your apps including built-in identity providers, but you can use third-party identity services such as Facebook, Twitter, and LinkedIn. With ASP.NET Core, you can easily manage app secrets, which are a way to store and use confidential information without having to expose it in the code.

Authentication vs. Authorization

Authentication is a process in which a user provides credentials that are then compared to those stored in an operating system, database, app or resource. If they match, users authenticate successfully, and can then perform actions that they're authorized for, during an authorization process. The authorization refers to the process that determines what a user is allowed to do.

Another way to think of authentication is to consider it as a way to enter a space, such as a server, database, app or resource, while authorization is which actions the user can perform to which objects inside that space (server, database, or app).

Common Vulnerabilities in software

ASP.NET Core and EF contain features that help you secure your apps and prevent security breaches. The following list of links takes you to documentation detailing techniques to avoid the most common security vulnerabilities in web apps:

- Cross-Site Scripting (XSS) attacks
- SQL injection attacks
- Cross-Site Request Forgery (XSRF/CSRF) attacks
- Open redirect attacks

There are more vulnerabilities that you should be aware of. For more information, see the other articles in the **Security and Identity** section of the table of contents.

Overview of ASP.NET Core authentication

9/22/2020 • 4 minutes to read • Edit Online

By Mike Rousos

Authentication is the process of determining a user's identity. Authorization is the process of determining whether a user has access to a resource. In ASP.NET Core, authentication is handled by the `IAuthenticationService`, which is used by authentication middleware. The authentication service uses registered authentication handlers to complete authentication-related actions. Examples of authentication-related actions include:

- Authenticating a user.
- Responding when an unauthenticated user tries to access a restricted resource.

The registered authentication handlers and their configuration options are called "schemes".

Authentication schemes are specified by registering authentication services in `Startup.ConfigureServices`:

- By calling a scheme-specific extension method after a call to `services.AddAuthentication` (such as `AddJwtBearer` or `AddCookie`, for example). These extension methods use AuthenticationBuilder.AddScheme to register schemes with appropriate settings.
- Less commonly, by calling AuthenticationBuilder.AddScheme directly.

For example, the following code registers authentication services and handlers for cookie and JWT bearer authentication schemes:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options => Configuration.Bind("JwtSettings",
options))
    .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, options =>
Configuration.Bind("CookieSettings", options));
```

The `AddAuthentication` parameter `JwtBearerDefaults.AuthenticationScheme` is the name of the scheme to use by default when a specific scheme isn't requested.

If multiple schemes are used, authorization policies (or authorization attributes) can specify the authentication scheme (or schemes) they depend on to authenticate the user. In the example above, the cookie authentication scheme could be used by specifying its name (`CookieAuthenticationDefaults.AuthenticationScheme` by default, though a different name could be provided when calling `AddCookie`).

In some cases, the call to `AddAuthentication` is automatically made by other extension methods. For example, when using ASP.NET Core Identity, `AddAuthentication` is called internally.

The Authentication middleware is added in `Startup.Configure` by calling the UseAuthentication extension method on the app's `IApplicationBuilder`. Calling `UseAuthentication` registers the middleware which uses the previously registered authentication schemes. Call `UseAuthentication` before any middleware that depends on users being authenticated. When using endpoint routing, the call to `UseAuthentication` must go:

- After `UseRouting`, so that route information is available for authentication decisions.
- Before `UseEndpoints`, so that users are authenticated before accessing the endpoints.

Authentication Concepts

Authentication scheme

An authentication scheme is a name which corresponds to:

- An authentication handler.
- Options for configuring that specific instance of the handler.

Schemes are useful as a mechanism for referring to the authentication, challenge, and forbid behaviors of the associated handler. For example, an authorization policy can use scheme names to specify which authentication scheme (or schemes) should be used to authenticate the user. When configuring authentication, it's common to specify the default authentication scheme. The default scheme is used unless a resource requests a specific scheme. It's also possible to:

- Specify different default schemes to use for authenticate, challenge, and forbid actions.
- Combine multiple schemes into one using policy schemes.

Authentication handler

An authentication handler:

- Is a type that implements the behavior of a scheme.
- Is derived from IAuthenticationHandler or AuthenticationHandler<TOptions>.
- Has the primary responsibility to authenticate users.

Based on the authentication scheme's configuration and the incoming request context, authentication handlers:

- Construct AuthenticationTicket objects representing the user's identity if authentication is successful.
- Return 'no result' or 'failure' if authentication is unsuccessful.
- Have methods for challenge and forbid actions for when users attempt to access resources:
  - They are unauthorized to access (forbid).
  - When they are unauthenticated (challenge).

Authenticate

An authentication scheme's authenticate action is responsible for constructing the user's identity based on request context. It returns an AuthenticateResult indicating whether authentication was successful and, if so, the user's identity in an authentication ticket. See AuthenticateAsync. Authenticate examples include:

- A cookie authentication scheme constructing the user's identity from cookies.
- A JWT bearer scheme deserializing and validating a JWT bearer token to construct the user's identity.

Challenge

An authentication challenge is invoked by Authorization when an unauthenticated user requests an endpoint that requires authentication. An authentication challenge is issued, for example, when an anonymous user requests a restricted resource or clicks on a login link. Authorization invokes a challenge using the specified authentication scheme(s), or the default if none is specified. See ChallengeAsync. Authentication challenge examples include:

- A cookie authentication scheme redirecting the user to a login page.
- A JWT bearer scheme returning a 401 result with a `www-authenticate: bearer` header.

A challenge action should let the user know what authentication mechanism to use to access the requested resource.

Forbid

An authentication scheme's forbid action is called by Authorization when an authenticated user attempts to access a resource they are not permitted to access. See ForbidAsync. Authentication forbid examples include:

- A cookie authentication scheme redirecting the user to a page indicating access was forbidden.
- A JWT bearer scheme returning a 403 result.

- A custom authentication scheme redirecting to a page where the user can request access to the resource.

A forbid action can let the user know:

- They are authenticated.
- They aren't permitted to access the requested resource.

See the following links for differences between challenge and forbid:

- Challenge and forbid with an operational resource handler.
- Differences between challenge and forbid.

Authentication providers per tenant

ASP.NET Core framework does not have a built-in solution for multi-tenant authentication. While it's certainly possible for customers to write one, using the built-in features, we recommend customers to look into Orchard Core for this purpose.

Orchard Core is:

- An open-source modular and multi-tenant app framework built with ASP.NET Core.
- A content management system (CMS) built on top of that app framework.

See the Orchard Core source for an example of authentication providers per tenant.

Additional resources

- Authorize with a specific scheme in ASP.NET Core
- Policy schemes in ASP.NET Core
- Create an ASP.NET Core app with user data protected by authorization
- Globally require authenticated users

Introduction to Identity on ASP.NET Core

9/22/2020 • 17 minutes to read • Edit Online

By Rick Anderson

ASP.NET Core Identity:

- Is an API that supports user interface (UI) login functionality.
- Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.

Users can create an account with the login information stored in Identity or they can use an external login provider. Supported external login providers include Facebook, Google, Microsoft Account, and Twitter.

For information on how to globally require all users to be authenticated, see Require authenticated users.

The Identity source code is available on GitHub. Scaffold Identity and view the generated files to review the template interaction with Identity.

Identity is typically configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.

In this topic, you learn how to use Identity to register, log in, and log out a user. Note: the templates treat username and email as the same for users. For more detailed instructions about creating apps that use Identity, see Next Steps.

Microsoft identity platform is:

- An evolution of the Azure Active Directory (Azure AD) developer platform.
- Unrelated to ASP.NET Core Identity.

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Azure Active Directory
- Azure Active Directory B2C (Azure AD B2C)
- IdentityServer4

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. IdentityServer4 enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see Welcome to IdentityServer4.

View or download the sample code (how to download).

Create a Web app with authentication

Create an ASP.NET Core Web Application project with Individual User Accounts.

- Visual Studio

- .NET Core CLI

- Select **File** > **New** > **Project**.
- Select **ASP.NET Core Web Application**. Name the project **WebApp1** to have the same namespace as the project download. Click **OK**.
- Select an ASP.NET Core **Web Application**, then select **Change Authentication**.
- Select **Individual User Accounts** and click **OK**.

The generated project provides ASP.NET Core Identity as a Razor Class Library. The Identity Razor Class Library exposes endpoints with the `Identity` area. For example:

- /Identity/Account/Login
- /Identity/Account/Logout
- /Identity/Account/Manage

Apply migrations

Apply the migrations to initialize the database.

- Visual Studio
- .NET Core CLI

Run the following command in the Package Manager Console (PMC):

```
PM> Update-Database
```

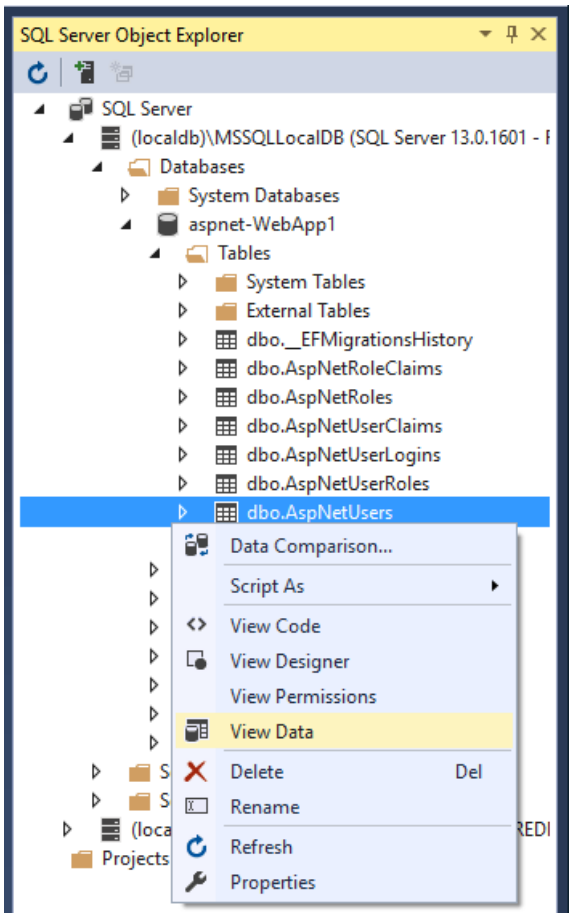
Test Register and Login

Run the app and register a user. Depending on your screen size, you might need to select the navigation toggle button to see the **Register** and **Login** links.

View the Identity database

- Visual Studio
- .NET Core CLI

- From the **View** menu, select **SQL Server Object Explorer** (SSOX).
- Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



Configure Identity services

Services are added in `ConfigureServices`. The typical pattern is to call all the `Add{Service}` methods, and then call all the `services.Configure{Service}` methods.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
     // options.UseSqlite(
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount =
true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings.
        options.Password.RequireDigit = true;
        options.Password.RequireLowercase = true;
        options.Password.RequireNonAlphanumeric = true;
        options.Password.RequireUppercase = true;
        options.Password.RequiredLength = 6;
        options.Password.RequiredUniqueChars = 1;

        // Lockout settings.
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
        options.Lockout.MaxFailedAccessAttempts = 5;
        options.Lockout.AllowedForNewUsers = true;

        // User settings.
        options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
        options.User.RequireUniqueEmail = false;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

        options.LoginPath = "/Identity/Account/Login";
        options.AccessDeniedPath = "/Identity/Account/AccessDenied";
        options.SlidingExpiration = true;
    });
}
```

The preceding highlighted code configures Identity with default option values. Services are made available to the app through dependency injection.

Identity is enabled by calling UseAuthentication. `UseAuthentication` adds authentication middleware to the request pipeline.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

The template-generated app doesn't use authorization. `app.UseAuthorization` is included to ensure it's added in the correct order should the app add authorization. `UseRouting`, `UseAuthentication`, `UseAuthorization`, and `UseEndpoints` must be called in the order shown in the preceding code.

For more information on `IdentityOptions` and `Startup`, see IdentityOptions and Application Startup.

Scaffold Register, Login, LogOut, and RegisterConfirmation

- Visual Studio
- .NET Core CLI

Add the `Register`, `Login`, `LogOut`, and `RegisterConfirmation` files. Follow the Scaffold identity into a Razor project with authorization instructions to generate the code shown in this section.

Examine Register

When a user clicks the **Register** button on the `Register` page, the `RegisterModel.OnPostAsync` action is invoked. The user is created by CreateAsync on the `_userManager` object:

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync())
                                          .ToList();
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = user.Id, code = code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your email",
                $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

            if (_userManager.Options.SignIn.RequireConfirmedAccount)
            {
                return RedirectToPage("RegisterConfirmation",
                                      new { email = Input.Email });
            }
            else
            {
                await _signInManager.SignInAsync(user, isPersistent: false);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

Disable default account verification

With the default templates, the user is redirected to the `Account.RegisterConfirmation` where they can select a link to have the account confirmed. The default `Account.RegisterConfirmation` is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set `DisplayConfirmAccountLink = false` in */Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs*:

```
[AllowAnonymous]
public class RegisterConfirmationModel : PageModel
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IEmailSender _sender;

    public RegisterConfirmationModel(UserManager<IdentityUser> userManager, IEmailSender sender)
    {
        _userManager = userManager;
        _sender = sender;
    }

    public string Email { get; set; }

    public bool DisplayConfirmAccountLink { get; set; }

    public string EmailConfirmationUrl { get; set; }

    public async Task<IActionResult> OnGetAsync(string email, string returnUrl = null)
    {
        if (email == null)
        {
            return RedirectToPage("/Index");
        }

        var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
        {
            return NotFound($"Unable to load user with email '{email}'.");
        }

        Email = email;
        // Once you add a real email sender, you should remove this code that lets you confirm the
account
        DisplayConfirmAccountLink = false;
        if (DisplayConfirmAccountLink)
        {
            var userId = await _userManager.GetUserIdAsync(user);
            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            EmailConfirmationUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = userId, code = code, returnUrl = returnUrl
},
                protocol: Request.Scheme);
        }

        return Page();
    }
}
```

Log in

The Login form is displayed when:

- The **Log in** link is selected.
- A user attempts to access a restricted page that they aren't authorized to access **or** when they haven't been authenticated by the system.

When the form on the Login page is submitted, the `OnPostAsync` action is called. `PasswordSignInAsync` is called on the `_signInManager` object.

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
                          Input.Password, Input.RememberMe, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new
            {
                ReturnUrl = returnUrl,
                RememberMe = Input.RememberMe
            });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

For information on how to make authorization decisions, see Introduction to authorization in ASP.NET Core.

Log out

The **Log out** link invokes the `LogoutModel.OnPost` action.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace WebApp1.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class LogoutModel : PageModel
    {
        private readonly SignInManager<IdentityUser> _signInManager;
        private readonly ILogger<LogoutModel> _logger;

        public LogoutModel(SignInManager<IdentityUser> signInManager, ILogger<LogoutModel> logger)
        {
            _signInManager = signInManager;
            _logger = logger;
        }

        public void OnGet()
        {
        }

        public async Task<IActionResult> OnPost(string returnUrl = null)
        {
            await _signInManager.SignOutAsync();
            _logger.LogInformation("User logged out.");
            if (returnUrl != null)
            {
                return LocalRedirect(returnUrl);
            }
            else
            {
                return RedirectToPage();
            }
        }
    }
}
```

In the preceding code, the code `return RedirectToPage();` needs to be a redirect so that the browser performs a new request and the identity for the user gets updated.

SignOutAsync clears the user's claims stored in a cookie.

Post is specified in the *Pages/Shared/_LoginPartial.cshtml*:

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Manage/Index"
                                            title="Manage">Hello @User.Identity.Name!</a>
    </li>
    <li class="nav-item">
        <form class="form-inline" asp-area="Identity" asp-page="/Account/Logout"
                                  asp-route-returnUrl="@Url.Page("/", new { area = "" })"
                                  method="post" >
            <button  type="submit" class="nav-link btn btn-link text-dark">Logout</button>
        </form>
    </li>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Register">Register</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>
    </li>
}
</ul>
```

Test Identity

The default web project templates allow anonymous access to the home pages. To test Identity, add `[Authorize]`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace WebApp1.Pages
{
    [Authorize]
    public class PrivacyModel : PageModel
    {
        private readonly ILogger<PrivacyModel> _logger;

        public PrivacyModel(ILogger<PrivacyModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

If you are signed in, sign out. Run the app and select the **Privacy** link. You are redirected to the login page.

Explore Identity

To explore Identity in more detail:

- Create full identity UI source
- Examine the source of each page and step through the debugger.

Identity Components

All the Identity-dependent NuGet packages are included in the ASP.NET Core shared framework.

The primary package for Identity is Microsoft.AspNetCore.Identity. This package contains the core set of interfaces for ASP.NET Core Identity, and is included by `Microsoft.AspNetCore.Identity.EntityFrameworkCore`.

Migrating to ASP.NET Core Identity

For more information and guidance on migrating your existing Identity store, see Migrate Authentication and Identity.

Setting password strength

See Configuration for a sample that sets the minimum password requirements.

AddDefaultIdentity and AddIdentity

AddDefaultIdentity was introduced in ASP.NET Core 2.1. Calling `AddDefaultIdentity` is similar to calling the following:

- AddIdentity
- AddDefaultUI
- AddDefaultTokenProviders

See AddDefaultIdentity source for more information.

Prevent publish of static Identity assets

To prevent publishing static Identity assets (stylesheets and JavaScript files for Identity UI) to the web root, add the following `ResolveStaticWebAssetsInputsDependsOn` property and `RemoveIdentityAssets` target to the app's project file:

```
<PropertyGroup>
  <ResolveStaticWebAssetsInputsDependsOn>RemoveIdentityAssets</ResolveStaticWebAssetsInputsDependsOn>
</PropertyGroup>

<Target Name="RemoveIdentityAssets">
  <ItemGroup>
    <StaticWebAsset Remove="@(StaticWebAsset)" Condition="%(SourceId) ==
'Microsoft.AspNetCore.Identity.UI'" />
  </ItemGroup>
</Target>
```

Next Steps

- ASP.NET Core Identity source code
- See this GitHub issue for information on configuring Identity using SQLite.
- Configure Identity
- Create an ASP.NET Core app with user data protected by authorization
- Add, download, and delete user data to Identity in an ASP.NET Core project

- Enable QR Code generation for TOTP authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core
- Account confirmation and password recovery in ASP.NET Core
- Two-factor authentication with SMS in ASP.NET Core
- Host ASP.NET Core in a web farm

By Rick Anderson

ASP.NET Core Identity is a membership system that adds login functionality to ASP.NET Core apps. Users can create an account with the login information stored in Identity or they can use an external login provider. Supported external login providers include Facebook, Google, Microsoft Account, and Twitter.

Identity can be configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.

View or download the sample code (how to download).

In this topic, you learn how to use Identity to register, log in, and log out a user. For more detailed instructions about creating apps that use Identity, see the Next Steps section at the end of this article.

AddDefaultIdentity and AddIdentity

AddDefaultIdentity was introduced in ASP.NET Core 2.1. Calling `AddDefaultIdentity` is similar to calling the following:

- AddIdentity
- AddDefaultUI
- AddDefaultTokenProviders

See AddDefaultIdentity source for more information.

Create a Web app with authentication

Create an ASP.NET Core Web Application project with Individual User Accounts.

- Visual Studio
- .NET Core CLI

- Select **File** > **New** > **Project**.
- Select **ASP.NET Core Web Application**. Name the project **WebApp1** to have the same namespace as the project download. Click **OK**.
- Select an ASP.NET Core **Web Application**, then select **Change Authentication**.
- Select **Individual User Accounts** and click **OK**.

The generated project provides ASP.NET Core Identity as a Razor Class Library. The Identity Razor Class Library exposes endpoints with the `Identity` area. For example:

- /Identity/Account/Login
- /Identity/Account/Logout
- /Identity/Account/Manage

Apply migrations

Apply the migrations to initialize the database.

- Visual Studio
- .NET Core CLI

Run the following command in the Package Manager Console (PMC):

```
Update-Database
```

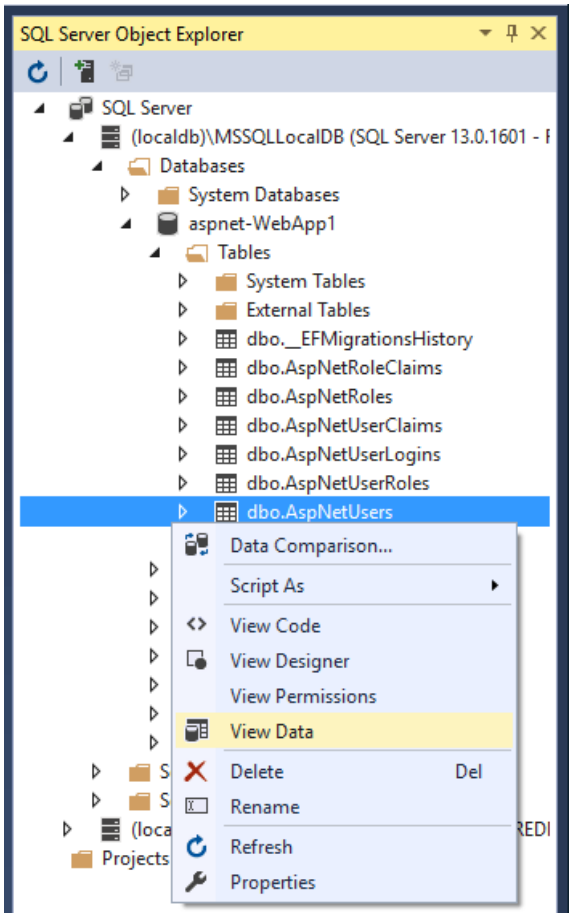
Test Register and Login

Run the app and register a user. Depending on your screen size, you might need to select the navigation toggle button to see the **Register** and **Login** links.

View the Identity database

- Visual Studio
- .NET Core CLI

- From the **View** menu, select **SQL Server Object Explorer** (SSOX).
- Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



Configure Identity services

Services are added in `ConfigureServices`. The typical pattern is to call all the `Add{Service}` methods, and then call all the `services.Configure{Service}` methods.

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.Configure<IdentityOptions>(options =>
    {
        // Password settings.
        options.Password.RequireDigit = true;
        options.Password.RequireLowercase = true;
        options.Password.RequireNonAlphanumeric = true;
        options.Password.RequireUppercase = true;
        options.Password.RequiredLength = 6;
        options.Password.RequiredUniqueChars = 1;

        // Lockout settings.
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
        options.Lockout.MaxFailedAccessAttempts = 5;
        options.Lockout.AllowedForNewUsers = true;

        // User settings.
        options.User.AllowedUserNameCharacters =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
        options.User.RequireUniqueEmail = false;
    });

    services.ConfigureApplicationCookie(options =>
    {
        // Cookie settings
        options.Cookie.HttpOnly = true;
        options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

        options.LoginPath = "/Identity/Account/Login";
        options.AccessDeniedPath = "/Identity/Account/AccessDenied";
        options.SlidingExpiration = true;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

The preceding code configures Identity with default option values. Services are made available to the app through dependency injection.

Identity is enabled by calling UseAuthentication. `UseAuthentication` adds authentication middleware to the request pipeline.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseAuthentication();

    app.UseMvc();
}
```

For more information, see the IdentityOptions Class and Application Startup.

Scaffold Register, Login, and LogOut

Follow the Scaffold identity into a Razor project with authorization instructions to generate the code shown in this section.

- Visual Studio
- .NET Core CLI

Add the Register, Login, and LogOut files.

Examine Register

When a user clicks the **Register** link, the `RegisterModel.OnPostAsync` action is invoked. The user is created by CreateAsync on the `_userManager` object:

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { userId = user.Id, code = code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your email",
                $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

            await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

If the user was created successfully, the user is logged in by the call to `_signInManager.SignInAsync`.

Note: See account confirmation for steps to prevent immediate login at registration.

Log in

The Login form is displayed when:

- The **Log in** link is selected.
- A user attempts to access a restricted page that they aren't authorized to access **or** when they haven't been authenticated by the system.

When the form on the Login page is submitted, the `OnPostAsync` action is called. `PasswordSignInAsync` is called on the `_signInManager` object.

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
            Input.Password, Input.RememberMe, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl, RememberMe =
Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

For information on how to make authorization decisions, see Introduction to authorization in ASP.NET Core.

Log out

The **Log out** link invokes the `LogoutModel.OnPost` action.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace WebApp1.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class LogoutModel : PageModel
    {
        private readonly SignInManager<IdentityUser> _signInManager;
        private readonly ILogger<LogoutModel> _logger;

        public LogoutModel(SignInManager<IdentityUser> signInManager, ILogger<LogoutModel> logger)
        {
            _signInManager = signInManager;
            _logger = logger;
        }

        public void OnGet()
        {
        }

        public async Task<IActionResult> OnPost(string returnUrl = null)
        {
            await _signInManager.SignOutAsync();
            _logger.LogInformation("User logged out.");
            if (returnUrl != null)
            {
                return LocalRedirect(returnUrl);
            }
            else
            {
                // This needs to be a redirect so that the browser performs a new
                // request and the identity for the user gets updated.
                return RedirectToPage();
            }
        }
    }
}
```

SignOutAsync clears the user's claims stored in a cookie.

Post is specified in the *Pages/Shared/_LoginPartial.cshtml*:

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
    @if (SignInManager.IsSignedIn(User))
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity"
               asp-page="/Account/Manage/Index"
               title="Manage">Hello@User.Identity.Name!</a>
        </li>
        <li class="nav-item">
            <form class="form-inline" asp-area="Identity" asp-page="/Account/Logout"
                  asp-route-returnUrl="@Url.Page("/", new { area = "" })"
                  method="post">
                <button type="submit" class="nav-link btn btn-link text-dark">Logout</button>
            </form>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Register">Register</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>
        </li>
    }
</ul>
```

Test Identity

The default web project templates allow anonymous access to the home pages. To test Identity, add `[Authorize]` to the Privacy page.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace WebApp1.Pages
{
    [Authorize]
    public class PrivacyModel : PageModel
    {
        public void OnGet()
        {
        }
    }
}
```

If you are signed in, sign out. Run the app and select the **Privacy** link. You are redirected to the login page.

Explore Identity

To explore Identity in more detail:

- Create full identity UI source
- Examine the source of each page and step through the debugger.

Identity Components

All the Identity dependent NuGet packages are included in the Microsoft.AspNetCore.App metapackage.

The primary package for Identity is Microsoft.AspNetCore.Identity. This package contains the core set of interfaces for ASP.NET Core Identity, and is included by `Microsoft.AspNetCore.Identity.EntityFrameworkCore`.

Migrating to ASP.NET Core Identity

For more information and guidance on migrating your existing Identity store, see Migrate Authentication and Identity.

Setting password strength

See Configuration for a sample that sets the minimum password requirements.

Next Steps

- See this GitHub issue for information on configuring Identity using SQLite.
- Configure Identity
- Create an ASP.NET Core app with user data protected by authorization
- Add, download, and delete user data to Identity in an ASP.NET Core project
- Enable QR Code generation for TOTP authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core
- Account confirmation and password recovery in ASP.NET Core
- Two-factor authentication with SMS in ASP.NET Core
- Host ASP.NET Core in a web farm

Authentication and authorization for SPAs

9/22/2020 • 11 minutes to read • Edit Online

ASP.NET Core 3.0 or later offers authentication in Single Page Apps (SPAs) using the support for API authorization. ASP.NET Core Identity for authenticating and storing users is combined with IdentityServer for implementing OpenID Connect.

An authentication parameter was added to the **Angular** and **React** project templates that is similar to the authentication parameter in the **Web Application (Model-View-Controller)** (MVC) and **Web Application** (Razor Pages) project templates. The allowed parameter values are **None** and **Individual**. The **React.js and Redux** project template doesn't support the authentication parameter at this time.

Create an app with API authorization support

User authentication and authorization can be used with both Angular and React SPAs. Open a command shell, and run the following command:

Angular:

```
dotnet new angular -o <output_directory_name> -au Individual
```

React:

```
dotnet new react -o <output_directory_name> -au Individual
```

The preceding command creates an ASP.NET Core app with a *ClientApp* directory containing the SPA.

General description of the ASP.NET Core components of the app

The following sections describe additions to the project when authentication support is included:

Startup class

The following code examples rely on the Microsoft.AspNetCore.ApiAuthorization.IdentityServer NuGet package. The examples configure API authentication and authorization using the AddApiAuthorization and AddIdentityServerJwt extension methods. Projects using the React or Angular SPA project templates with authentication include a reference to this package.

The `Startup` class has the following additions:

- Inside the `Startup.ConfigureServices` method:
  - Identity with the default UI:

    ```
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlite(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>();
    ```

  - IdentityServer with an additional `AddApiAuthorization` helper method that sets up some default ASP.NET Core conventions on top of IdentityServer:

```
services.AddIdentityServer()
    .AddApiAuthorization<ApplicationUser, ApplicationDbContext>();
```

- Authentication with an additional `AddIdentityServerJwt` helper method that configures the app to validate JWT tokens produced by IdentityServer:

```
services.AddAuthentication()
    .AddIdentityServerJwt();
```

- Inside the `Startup.Configure` method:
  - The authentication middleware that is responsible for validating the request credentials and setting the user on the request context:

```
app.UseAuthentication();
```

  - The IdentityServer middleware that exposes the OpenID Connect endpoints:

```
app.UseIdentityServer();
```

AddApiAuthorization

This helper method configures IdentityServer to use our supported configuration. IdentityServer is a powerful and extensible framework for handling app security concerns. At the same time, that exposes unnecessary complexity for the most common scenarios. Consequently, a set of conventions and configuration options is provided to you that are considered a good starting point. Once your authentication needs change, the full power of IdentityServer is still available to customize authentication to suit your needs.

AddIdentityServerJwt

This helper method configures a policy scheme for the app as the default authentication handler. The policy is configured to let Identity handle all requests routed to any subpath in the Identity URL space "/Identity". The `JwtBearerHandler` handles all other requests. Additionally, this method registers an `<<ApplicationName>>API` API resource with IdentityServer with a default scope of `<<ApplicationName>>API` and configures the JWT Bearer token middleware to validate tokens issued by IdentityServer for the app.

WeatherForecastController

In the *Controllers\WeatherForecastController.cs* file, notice the `[Authorize]` attribute applied to the class that indicates that the user needs to be authorized based on the default policy to access the resource. The default authorization policy happens to be configured to use the default authentication scheme, which is set up by `AddIdentityServerJwt` to the policy scheme that was mentioned above, making the `JwtBearerHandler` configured by such helper method the default handler for requests to the app.

ApplicationDbContext

In the *Data\ApplicationDbContext.cs* file, notice the same `DbContext` is used in Identity with the exception that it extends `ApiAuthorizationDbContext` (a more derived class from `IdentityDbContext`) to include the schema for IdentityServer.

To gain full control of the database schema, inherit from one of the available Identity `DbContext` classes and configure the context to include the Identity schema by calling `builder.ConfigurePersistedGrantContext(_operationalStoreOptions.Value)` on the `OnModelCreating` method.

OidcConfigurationController

In the *Controllers\OidcConfigurationController.cs* file, notice the endpoint that's provisioned to serve the OIDC

parameters that the client needs to use.

appsettings.json

In the *appsettings.json* file of the project root, there's a new `IdentityServer` section that describes the list of configured clients. In the following example, there's a single client. The client name corresponds to the app name and is mapped by convention to the OAuth `ClientId` parameter. The profile indicates the app type being configured. It's used internally to drive conventions that simplify the configuration process for the server. There are several profiles available, as explained in the Application profiles section.

```
"IdentityServer": {
  "Clients": {
    "angularindividualpreview3final": {
      "Profile": "IdentityServerSPA"
    }
  }
}
```

appsettings.Development.json

In the *appsettings.Development.json* file of the project root, there's an `IdentityServer` section that describes the key used to sign tokens. When deploying to production, a key needs to be provisioned and deployed alongside the app, as explained in the Deploy to production section.

```
"IdentityServer": {
  "Key": {
    "Type": "Development"
  }
}
```

General description of the Angular app

The authentication and API authorization support in the Angular template resides in its own Angular module in the *ClientApp\src\api-authorization* directory. The module is composed of the following elements:

- 3 components:
  - *login.component.ts*: Handles the app's login flow.
  - *logout.component.ts*: Handles the app's logout flow.
  - *login-menu.component.ts*: A widget that displays one of the following sets of links:
    - User profile management and log out links when the user is authenticated.
    - Registration and log in links when the user isn't authenticated.
- A route guard `AuthorizeGuard` that can be added to routes and requires a user to be authenticated before visiting the route.
- An HTTP interceptor `AuthorizeInterceptor` that attaches the access token to outgoing HTTP requests targeting the API when the user is authenticated.
- A service `AuthorizeService` that handles the lower-level details of the authentication process and exposes information about the authenticated user to the rest of the app for consumption.
- An Angular module that defines routes associated with the authentication parts of the app. It exposes the login menu component, the interceptor, the guard, and the service for consumption from the rest of the app.

General description of the React app

The support for authentication and API authorization in the React template resides in the *ClientApp\src\components\api-authorization* directory. It's composed of the following elements:

- 4 components:
  - *Login.js*: Handles the app's login flow.
  - *Logout.js*: Handles the app's logout flow.
  - *LoginMenu.js*: A widget that displays one of the following sets of links:
    - User profile management and log out links when the user is authenticated.
    - Registration and log in links when the user isn't authenticated.
  - *AuthorizeRoute.js*: A route component that requires a user to be authenticated before rendering the component indicated in the `Component` parameter.
- An exported `authService` instance of class `AuthorizeService` that handles the lower-level details of the authentication process and exposes information about the authenticated user to the rest of the app for consumption.

Now that you've seen the main components of the solution, you can take a deeper look at individual scenarios for the app.

Require authorization on a new API

By default, the system is configured to easily require authorization for new APIs. To do so, create a new controller and add the `[Authorize]` attribute to the controller class or to any action within the controller.

Customize the API authentication handler

To customize the configuration of the API's JWT handler, configure its JwtBearerOptions instance:

```
services.AddAuthentication()
    .AddIdentityServerJwt();

services.Configure<JwtBearerOptions>(
    IdentityServerJwtConstants.IdentityServerJwtBearerScheme,
    options =>
    {
        ...
    });
```

The API's JWT handler raises events that enable control over the authentication process using `JwtBearerEvents`. To provide support for API authorization, `AddIdentityServerJwt` registers its own event handlers.

To customize the handling of an event, wrap the existing event handler with additional logic as required. For example:

```
services.Configure<JwtBearerOptions>(
    IdentityServerJwtConstants.IdentityServerJwtBearerScheme,
    options =>
    {
        var onTokenValidated = options.Events.OnTokenValidated;

        options.Events.OnTokenValidated = async context =>
        {
            await onTokenValidated(context);
            ...
        }
    });
```

In the preceding code, the `OnTokenValidated` event handler is replaced with a custom implementation. This implementation:

1. Calls the original implementation provided by the API authorization support.
2. Run its own custom logic.

Protect a client-side route (Angular)

Protecting a client-side route is done by adding the authorize guard to the list of guards to run when configuring a route. As an example, you can see how the `fetch-data` route is configured within the main app Angular module:

```
RouterModule.forRoot([
  // ...
  { path: 'fetch-data', component: FetchDataComponent, canActivate: [AuthorizeGuard] },
])
```

It's important to mention that protecting a route doesn't protect the actual endpoint (which still requires an `[Authorize]` attribute applied to it) but that it only prevents the user from navigating to the given client-side route when it isn't authenticated.

Authenticate API requests (Angular)

Authenticating requests to APIs hosted alongside the app is done automatically through the use of the HTTP client interceptor defined by the app.

Protect a client-side route (React)

Protect a client-side route by using the `AuthorizeRoute` component instead of the plain `Route` component. For example, notice how the `fetch-data` route is configured within the `App` component:

```
<AuthorizeRoute path='/fetch-data' component={FetchData} />
```

Protecting a route:

- Doesn't protect the actual endpoint (which still requires an `[Authorize]` attribute applied to it).
- Only prevents the user from navigating to the given client-side route when it isn't authenticated.

Authenticate API requests (React)

Authenticating requests with React is done by first importing the `authService` instance from the `AuthorizeService` . The access token is retrieved from the `authService` and is attached to the request as shown below. In React components, this work is typically done in the `componentDidMount` lifecycle method or as the result from some user interaction.

Import the authService into your component

```
import authService from './api-authorization/AuthorizeService'
```

Retrieve and attach the access token to the response

```
async populateWeatherData() {
  const token = await authService.getAccessToken();
  const response = await fetch('api/SampleData/WeatherForecasts', {
    headers: !token ? {} : { 'Authorization': `Bearer ${token}` }
  });
  const data = await response.json();
  this.setState({ forecasts: data, loading: false });
}
```

Deploy to production

To deploy the app to production, the following resources need to be provisioned:

- A database to store the Identity user accounts and the IdentityServer grants.
- A production certificate to use for signing tokens.
  - There are no specific requirements for this certificate; it can be a self-signed certificate or a certificate provisioned through a CA authority.
  - It can be generated through standard tools like PowerShell or OpenSSL.
  - It can be installed into the certificate store on the target machines or deployed as a *.pfx* file with a strong password.

Example: Deploy to Azure App Service

This section describes deploying the app to Azure App Service using a certificate stored in the certificate store. To modify the app to load a certificate from the certificate store, a Standard tier service plan or better is required when you configure the app in the Azure portal in a later step.

In the app's *appsettings.json* file, modify the `IdentityServer` section to include the key details:

```
"IdentityServer": {
  "Key": {
    "Type": "Store",
    "StoreName": "My",
    "StoreLocation": "CurrentUser",
    "Name": "CN=MyApplication"
  }
}
```

- The store name represents the name of the certificate store where the certificate is stored. In this case, it points to the personal user store.
- The store location represents where to load the certificate from (`CurrentUser` or `LocalMachine`).
- The name property on certificate corresponds with the distinguished subject for the certificate.

To deploy to Azure App Service, follow the steps in Deploy the app to Azure, which explains how to create the necessary Azure resources and deploy the app to production.

After following the preceding instructions, the app is deployed to Azure but isn't yet functional. The certificate used by the app must be configured in the Azure portal. Locate the thumbprint for the certificate and follow the steps described in Load your certificates.

While these steps mention SSL, there's a **Private certificates** section in the Azure portal where you can upload the provisioned certificate to use with the app.

After configuring the app and the app's settings in the Azure portal, restart the app in the portal.

Other configuration options

The support for API authorization builds on top of IdentityServer with a set of conventions, default values, and enhancements to simplify the experience for SPAs. Needless to say, the full power of IdentityServer is available behind the scenes if the ASP.NET Core integrations don't cover your scenario. The ASP.NET Core support is focused on "first-party" apps, where all the apps are created and deployed by our organization. As such, support isn't offered for things like consent or federation. For those scenarios, use IdentityServer and follow their documentation.

Application profiles

Application profiles are predefined configurations for apps that further define their parameters. At this time, the following profiles are supported:

- `IdentityServerSPA`: Represents a SPA hosted alongside IdentityServer as a single unit.
  - The `redirect_uri` defaults to `/authentication/login-callback`.
  - The `post_logout_redirect_uri` defaults to `/authentication/logout-callback`.
  - The set of scopes includes the `openid`, `profile`, and every scope defined for the APIs in the app.
  - The set of allowed OIDC response types is `id_token token` or each of them individually (`id_token`, `token`).
  - The allowed response mode is `fragment`.
- `SPA`: Represents a SPA that isn't hosted with IdentityServer.
  - The set of scopes includes the `openid`, `profile`, and every scope defined for the APIs in the app.
  - The set of allowed OIDC response types is `id_token token` or each of them individually (`id_token`, `token`).
  - The allowed response mode is `fragment`.
- `IdentityServerJwt`: Represents an API that is hosted alongside with IdentityServer.
  - The app is configured to have a single scope that defaults to the app name.
- `API`: Represents an API that isn't hosted with IdentityServer.
  - The app is configured to have a single scope that defaults to the app name.

Configuration through AppSettings

Configure the apps through the configuration system by adding them to the list of `Clients` or `Resources`.

Configure each client's `redirect_uri` and `post_logout_redirect_uri` property, as shown in the following example:

```
"IdentityServer": {
  "Clients": {
    "MySPA": {
      "Profile": "SPA",
      "RedirectUri": "https://www.example.com/authentication/login-callback",
      "LogoutUri": "https://www.example.com/authentication/logout-callback"
    }
  }
}
```

When configuring resources, you can configure the scopes for the resource as shown below:

```
"IdentityServer": {
  "Resources": {
    "MyExternalApi": {
      "Profile": "API",
      "Scopes": "a b c"
    }
  }
}
```

Configuration through code

You can also configure the clients and resources through code using an overload of `AddApiAuthorization` that takes an action to configure options.

```
AddApiAuthorization<ApplicationUser, ApplicationDbContext>(options =>
{
    options.Clients.AddSPA(
        "My SPA", spa =>
        spa.WithRedirectUri("http://www.example.com/authentication/login-callback")
           .WithLogoutRedirectUri(
               "http://www.example.com/authentication/logout-callback"));

    options.ApiResources.AddApiResource("MyExternalApi", resource =>
        resource.WithScopes("a", "b", "c"));
});
```

Additional resources

- Use the Angular project template with ASP.NET Core
- Use the React project template with ASP.NET Core
- Scaffold Identity in ASP.NET Core projects

Scaffold Identity in ASP.NET Core projects

9/22/2020 • 51 minutes to read • Edit Online

By Rick Anderson

ASP.NET Core provides ASP.NET Core Identity as a Razor Class Library. Applications that include Identity can apply the scaffolder to selectively add the source code contained in the Identity Razor Class Library (RCL). You might want to generate source code so you can modify the code and change the behavior. For example, you could instruct the scaffolder to generate the code used in registration. Generated code takes precedence over the same code in the Identity RCL. To gain full control of the UI and not use the default RCL, see the section Create full Identity UI source.

Applications that do **not** include authentication can apply the scaffolder to add the RCL Identity package. You have the option of selecting Identity code to be generated.

Although the scaffolder generates most of the necessary code, you need to update your project to complete the process. This document explains the steps needed to complete an Identity scaffolding update.

We recommend using a source control system that shows file differences and allows you to back out of changes. Inspect the changes after running the Identity scaffolder.

Services are required when using Two Factor Authentication, Account confirmation and password recovery, and other security features with Identity. Services or service stubs aren't generated when scaffolding Identity. Services to enable these features must be added manually. For example, see Require Email Confirmation.

When scaffolding Identity with a new data context into a project with existing individual accounts:

- In `Startup.ConfigureServices`, remove the calls to:
  - `AddDbContext`
  - `AddDefaultIdentity`

For example, `AddDbContext` and `AddDefaultIdentity` are commented out in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    //services.AddDbContext<ApplicationDbContext>(options =>
    //    options.UseSqlServer(
    //        Configuration.GetConnectionString("DefaultConnection")));
    //services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
    //    .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

The preceeding code comments out the code that is duplicated in *Areas/Identity/IdentityHostingStartup.cs*

Typically, apps that were created with individual accounts should ***not*** create a new data context.

Scaffold Identity into an empty project

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
    - Blazor Server apps created from the Blazor Server template (`blazorserver`) aren't configured for Razor Pages or MVC by default. Leave the layout page entry blank.
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
- Select **Add**.

Update the `Startup` class with code similar to the following:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
            endpoints.MapRazorPages();
        });
    }
}
```

`UseHsts` is recommended but not required. For more information, see HTTP Strict Transport Security Protocol.

The generated Identity database code requires Entity Framework Core Migrations. Create a migration and update the database. For example, run the following commands:

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Add-Migration CreateIdentitySchema
Update-Database
```

The "CreateIdentitySchema" name parameter for the `Add-Migration` command is arbitrary. `"CreateIdentitySchema"` describes the migration.

Scaffold Identity into a Razor project without existing authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
    - Blazor Server apps created from the Blazor Server template (`blazorserver`) aren't configured for Razor Pages or MVC by default. Leave the layout page entry blank.
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
- Select **Add**.

Identity is configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

Migrations, UseAuthentication, and layout

The generated Identity database code requires Entity Framework Core Migrations. Create a migration and update the database. For example, run the following commands:

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Add-Migration CreateIdentitySchema
Update-Database
```

The "CreateIdentitySchema" name parameter for the `Add-Migration` command is arbitrary. `"CreateIdentitySchema"` describes the migration.

Enable authentication

Update the `Startup` class with code similar to the following:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

`UseHsts` is recommended but not required. For more information, see HTTP Strict Transport Security Protocol.

Layout changes

Optional: Add the login partial (`_LoginPartial`) to the layout file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebRP</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">WebRP</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <partial name="_LoginPartial" />
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - WebRP - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

Scaffold Identity into a Razor project with authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.

    - Select your existing layout page so your layout file isn't overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
        - `~/Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure
        - `~/Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
    - Select your data context class.
    - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
    - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
    - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
    - Select your existing layout page, or your layout file will be overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
        - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
        - `~/Views/Shared/_Layout.cshtml` for MVC projects
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
    - Select your data context class.
    - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
    - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
    - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Some Identity options are configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

Scaffold Identity into an MVC project without existing authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity** > **Add**.

- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
    - Blazor Server apps created from the Blazor Server template (`blazorserver`) aren't configured for Razor Pages or MVC by default. Leave the layout page entry blank.
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
- Select **Add**.

Optional: Add the login partial (`_LoginPartial`) to the *Views/Shared/_Layout.cshtml* file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebRP</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-
shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-page="/Index">WebRP</a>
                <button class="navbar-toggler" type="button" data-toggle="collapse" data-target=".navbar-
collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
                    <partial name="_LoginPartial" />
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-page="/Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - WebRP - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

- Move the *Pages/Shared/_LoginPartial.cshtml* file to *Views/Shared/_LoginPartial.cshtml*

Identity is configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

The generated Identity database code requires Entity Framework Core Migrations. Create a migration and update the database. For example, run the following commands:

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Add-Migration CreateIdentitySchema
Update-Database
```

The "CreateIdentitySchema" name parameter for the `Add-Migration` command is arbitrary. `"CreateIdentitySchema"` describes the migration.

Update the `Startup` class with code similar to the following:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
            endpoints.MapRazorPages();
        });
    }
}
```

`UseHsts` is recommended but not required. For more information, see HTTP Strict Transport Security Protocol.

Scaffold Identity into an MVC project with authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page so your layout file isn't overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure
    - `~/Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
  - Select your data context class.
  - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
  - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
  - Select your data context class.
  - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
  - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Scaffold Identity into a Blazor Server project without existing authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
    - Blazor Server apps created from the Blazor Server template (`blazorserver`) aren't configured for Razor Pages or MVC by default. Leave the layout page entry blank.
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
- Select **Add**.

Identity is configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

Migrations

The generated Identity database code requires Entity Framework Core Migrations. Create a migration and update the database. For example, run the following commands:

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Add-Migration CreateIdentitySchema
Update-Database
```

The "CreateIdentitySchema" name parameter for the `Add-Migration` command is arbitrary. `"CreateIdentitySchema"` describes the migration.

Pass an XSRF token to the app

Tokens can be passed to components:

- When authentication tokens are provisioned and saved to the authentication cookie, they can be passed to components.
- Razor components can't use `HttpContext` directly, so there's no way to obtain an anti-request forgery (XSRF) token to POST to Identity's logout endpoint at `/Identity/Account/Logout`. An XSRF token can be passed to components.

For more information, see ASP.NET Core Blazor Server additional security scenarios.

In the *Pages/_Host.cshtml* file, establish the token after adding it to the `InitialApplicationState` and `TokenProvider` classes:

```
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Xsrf

...

var tokens = new InitialApplicationState
{
    ...

    XsrfToken = Xsrf.GetAndStoreTokens(HttpContext).RequestToken
};
```

Update the `App` component (*App.razor*) to assign the `InitialState.XsrfToken`:

```
@inject TokenProvider TokenProvider

...

TokenProvider.XsrfToken = InitialState.XsrfToken;
```

The `TokenProvider` service demonstrated in the topic is used in the `LoginDisplay` component in the following Layout and authentication flow changes section.

Enable authentication

In the `Startup` class:

- Confirm that Razor Pages services are added in `Startup.ConfigureServices`.
- If using the TokenProvider, register the service.
- Call `UseDatabaseErrorPage` on the application builder in `Startup.Configure` for the Development environment.
- Call `UseAuthentication` and `UseAuthorization` after `UseRouting`.
- Add an endpoint for Razor Pages.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddServerSideBlazor();
    services.AddSingleton<WeatherForecastService>();
    services.AddScoped<TokenProvider>();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
    });
}
```

`UseHsts` is recommended but not required. For more information, see HTTP Strict Transport Security Protocol.

Layout and authentication flow changes

Add a `RedirectToLogin` component (*RedirectToLogin.razor*) to the app's *Shared* folder in the project root:

```
@inject NavigationManager Navigation
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo("Identity/Account/Login?returnUrl=" +
            Uri.EscapeDataString(Navigation.Uri), true);
    }
}
```

Add a `LoginDisplay` component (*LoginDisplay.razor*) to the app's *Shared* folder. The TokenProvider service provides the XSRF token for the HTML form that POSTs to Identity's logout endpoint:

```
@using Microsoft.AspNetCore.Components.Authorization
@inject NavigationManager Navigation
@inject TokenProvider TokenProvider

<AuthorizeView>
    <Authorized>
        <a href="Identity/Account/Manage/Index">
            Hello, @context.User.Identity.Name!
        </a>
        <form action="/Identity/Account/Logout?returnUrl=%2F" method="post">
            <button class="nav-link btn btn-link" type="submit">Logout</button>
            <input name="__RequestVerificationToken" type="hidden"
                value="@TokenProvider.XsrfToken">
        </form>
    </Authorized>
    <NotAuthorized>
        <a href="Identity/Account/Register">Register</a>
        <a href="Identity/Account/Login">Login</a>
    </NotAuthorized>
</AuthorizeView>
```

In the `MainLayout` component (*Shared/MainLayout.razor*), add the `LoginDisplay` component to the top-row `<div>` element's content:

```
<div class="top-row px-4 auth">
    <LoginDisplay />
    <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
</div>
```

Style authentication endpoints

Because Blazor Server uses Razor Pages Identity pages, the styling of the UI changes when a visitor navigates between Identity pages and components. You have two options to address the incongruous styles:

Build Identity components

An approach to using components for Identity instead of pages is to build Identity components. Because `SignInManager` and `UserManager` aren't supported in Razor components, use API endpoints in the Blazor Server app to process user account actions.

Use a custom layout with Blazor app styles

The Identity pages layout and styles can be modified to produce pages that use the default Blazor theme.

> **NOTE**
>
> The example in this section is merely a starting point for customization. Additional work is likely required for the best user experience.

Create a new `NavMenu_IdentityLayout` component (*Shared/NavMenu_IdentityLayout.razor*). For the markup and code of the component, use the same content of the app's `NavMenu` component (*Shared/NavMenu.razor*). Strip out any `NavLink` s to components that can't be reached anonymously because automatic redirects in the `RedirectToLogin` component fail for components requiring authentication or authorization.

In the *Pages/Shared/Layout.cshtml* file, make the following changes:

- Add Razor directives to the top of the file to use Tag Helpers and the app's components in the *Shared* folder:

  ```
  @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
  @using {APPLICATION ASSEMBLY}.Shared
  ```

  Replace `{APPLICATION ASSEMBLY}` with the app's assembly name.

- Add a `<base>` tag and Blazor stylesheet `<link>` to the `<head>` content:

  ```
  <base href="~/" />
  <link rel="stylesheet" href="~/css/site.css" />
  ```

- Change the content of the `<body>` tag to the following:

```
<div class="sidebar" style="float:left">
    <component type="typeof(NavMenu_IdentityLayout)"
        render-mode="ServerPrerendered" />
</div>

<div class="main" style="padding-left:250px">
    <div class="top-row px-4">
        @{
            var result = Engine.FindView(ViewContext, "_LoginPartial",
                isMainPage: false);
        }
        @if (result.Success)
        {
            await Html.RenderPartialAsync("_LoginPartial");
        }
        else
        {
            throw new InvalidOperationException("The default Identity UI " +
                "layout requires a partial view '_LoginPartial'.");
        }
        <a href="https://docs.microsoft.com/aspnet/" target="_blank">About</a>
    </div>

    <div class="content px-4">
        @RenderBody()
    </div>
</div>

<script src="~/Identity/lib/jquery/dist/jquery.min.js"></script>
<script src="~/Identity/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/Identity/js/site.js" asp-append-version="true"></script>
@RenderSection("Scripts", required: false)
<script src="_framework/blazor.server.js"></script>
```

Scaffold Identity into a Blazor Server project with authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page so your layout file isn't overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure
    - `~/Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
  - Select your data context class.
  - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
  - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
  - Select your data context class.
  - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
  - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Some Identity options are configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

Create full Identity UI source

To maintain full control of the Identity UI, run the Identity scaffolder and select **Override all files**.

The following highlighted code shows the changes to replace the default Identity UI with Identity in an ASP.NET Core 2.1 web app. You might want to do this to have full control of the Identity UI.

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<IdentityUser, IdentityRole>()
        // services.AddDefaultIdentity<IdentityUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
        .AddRazorPagesOptions(options =>
        {
            options.AllowAreas = true;
            options.Conventions.AuthorizeAreaFolder("Identity", "/Account/Manage");
            options.Conventions.AuthorizeAreaPage("Identity", "/Account/Logout");
        });

    services.ConfigureApplicationCookie(options =>
    {
        options.LoginPath = $"/Identity/Account/Login";
        options.LogoutPath = $"/Identity/Account/Logout";
        options.AccessDeniedPath = $"/Identity/Account/AccessDenied";
    });

    // using Microsoft.AspNetCore.Identity.UI.Services;
    services.AddSingleton<IEmailSender, EmailSender>();
}
```

The default Identity is replaced in the following code:

```
services.AddIdentity<IdentityUser, IdentityRole>()
    // services.AddDefaultIdentity<IdentityUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

The following code sets the LoginPath, LogoutPath, and AccessDeniedPath:

```
services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = $"/Identity/Account/Login";
    options.LogoutPath = $"/Identity/Account/Logout";
    options.AccessDeniedPath = $"/Identity/Account/AccessDenied";
});
```

Register an `IEmailSender` implementation, for example:

```
// using Microsoft.AspNetCore.Identity.UI.Services;
services.AddSingleton<IEmailSender, EmailSender>();
```

```
public class EmailSender : IEmailSender
{
    public Task SendEmailAsync(string email, string subject, string message)
    {
        return Task.CompletedTask;
    }
}
```

Password configuration

If PasswordOptions are configured in `Startup.ConfigureServices`, `[StringLength]` attribute configuration might be required for the `Password` property in scaffolded Identity pages. `InputModel` `Password` properties are found in the following files:

- `Areas/Identity/Pages/Account/Register.cshtml.cs`
- `Areas/Identity/Pages/Account/ResetPassword.cshtml.cs`

Disable a page

This sections show how to disable the register page but the approach can be used to disable any page.

To disable user registration:

- Scaffold Identity. Include Account.Register, Account.Login, and Account.RegisterConfirmation. For example:

  ```
  dotnet aspnet-codegenerator identity -dc RPauth.Data.ApplicationDbContext --files
  "Account.Register;Account.Login;Account.RegisterConfirmation"
  ```

- Update *Areas/Identity/Pages/Account/Register.cshtml.cs* so users can't register from this endpoint:

  ```
  public class RegisterModel : PageModel
  {
      public IActionResult OnGet()
      {
          return RedirectToPage("Login");
      }

      public IActionResult OnPost()
      {
          return RedirectToPage("Login");
      }
  }
  ```

- Update *Areas/Identity/Pages/Account/Register.cshtml* to be consistent with the preceding changes:

  ```
  @page
  @model RegisterModel
  @{
      ViewData["Title"] = "Go to Login";
  }

  <h1>@ViewData["Title"]</h1>

  <li class="nav-item">
      <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>
  </li>
  ```

- Comment out or remove the registration link from *Areas/Identity/Pages/Account/Login.cshtml*

```
@*
<p>
    <a asp-page="./Register" asp-route-returnUrl="@Model.ReturnUrl">Register as a new user</a>
</p>
*@
```

- Update the *Areas/Identity/Pages/Account/RegisterConfirmation* page.
  - Remove the code and links from the cshtml file.
  - Remove the confirmation code from the `PageModel`:

```
[AllowAnonymous]
  public class RegisterConfirmationModel : PageModel
  {
      public IActionResult OnGet()
      {
          return Page();
      }
  }
```

Use another app to add users

Provide a mechanism to add users outside the web app. Options to add users include:

- A dedicated admin web app.
- A console app.

The following code outlines one approach to adding users:

- A list of users is read into memory.
- A strong unique password is generated for each user.
- The user is added to the Identity database.
- The user is notified and told to change the password.

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<AppDbCntx>();
                context.Database.Migrate();

                var config = host.Services.GetRequiredService<IConfiguration>();
                var userList = config.GetSection("userList").Get<List<string>>();

                SeedData.Initialize(services, userList).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred adding users.");
            }
        }

        host.Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code outlines adding a user:

```
public static async Task Initialize(IServiceProvider serviceProvider,
                                    List<string> userList)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    foreach (var userName in userList)
    {
        var userPassword = GenerateSecurePassword();
        var userId = await EnsureUser(userManager, userName, userPassword);

        NotifyUser(userName, userPassword);
    }
}

private static async Task<string> EnsureUser(UserManager<IdentityUser> userManager,
                                             string userName, string userPassword)
{
    var user = await userManager.FindByNameAsync(userName);

    if (user == null)
    {
        user = new IdentityUser(userName)
        {
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, userPassword);
    }

    return user.Id;
}
```

A similar approach can be followed for production scenarios.

Prevent publish of static Identity assets

To prevent publishing static Identity assets to the web root, see Introduction to Identity on ASP.NET Core.

Additional resources

- Changes to authentication code to ASP.NET Core 2.1 and later

ASP.NET Core 2.1 and later provides ASP.NET Core Identity as a Razor Class Library. Applications that include Identity can apply the scaffolder to selectively add the source code contained in the Identity Razor Class Library (RCL). You might want to generate source code so you can modify the code and change the behavior. For example, you could instruct the scaffolder to generate the code used in registration. Generated code takes precedence over the same code in the Identity RCL. To gain full control of the UI and not use the default RCL, see the section Create full identity UI source.

Applications that do **not** include authentication can apply the scaffolder to add the RCL Identity package. You have the option of selecting Identity code to be generated.

Although the scaffolder generates most of the necessary code, you'll have to update your project to complete the process. This document explains the steps needed to complete an Identity scaffolding update.

When the Identity scaffolder is run, a *ScaffoldingReadme.txt* file is created in the project directory. The *ScaffoldingReadme.txt* file contains general instructions on what's needed to complete the Identity scaffolding update. This document contains more complete instructions than the *ScaffoldingReadme.txt* file.

We recommend using a source control system that shows file differences and allows you to back out of changes. Inspect the changes after running the Identity scaffolder.

> **NOTE**
>
> Services are required when using Two Factor Authentication, Account confirmation and password recovery, and other security features with Identity. Services or service stubs aren't generated when scaffolding Identity. Services to enable these features must be added manually. For example, see Require Email Confirmation.

Scaffold Identity into an empty project

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
    - Blazor Server apps created from the Blazor Server template (`blazorserver`) aren't configured for Razor Pages or MVC by default. Leave the layout page entry blank.
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
- Select **Add**.

Add the following highlighted calls to the `Startup` class:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAuthentication();
        app.UseMvc();
    }
}
```

`UseHsts` is recommended but not required. For more information, see HTTP Strict Transport Security Protocol.

The generated Identity database code requires Entity Framework Core Migrations. Create a migration and update the database. For example, run the following commands:

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Add-Migration CreateIdentitySchema
Update-Database
```

The "CreateIdentitySchema" name parameter for the `Add-Migration` command is arbitrary. `"CreateIdentitySchema"` describes the migration.

Scaffold Identity into a Razor project without existing authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
    - Blazor Server apps created from the Blazor Server template (`blazorserver`) aren't configured for Razor Pages or MVC by default. Leave the layout page entry blank.
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
- Select **Add**.

Identity is configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

Migrations, UseAuthentication, and layout

The generated Identity database code requires Entity Framework Core Migrations. Create a migration and update the database. For example, run the following commands:

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Add-Migration CreateIdentitySchema
Update-Database
```

The "CreateIdentitySchema" name parameter for the `Add-Migration` command is arbitrary. `"CreateIdentitySchema"` describes the migration.

Enable authentication

In the `Configure` method of the `Startup` class, call UseAuthentication after `UseStaticFiles`:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAuthentication();

        app.UseMvc();
    }
}
```

`UseHsts` is recommended but not required. For more information, see HTTP Strict Transport Security Protocol.

Layout changes

Optional: Add the login partial (`_LoginPartial`) to the layout file:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorNoAuth8</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
```

```
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-brand">RazorNoAuth8</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Contact">Contact</a></li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - RazorNoAuth8</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

Scaffold Identity into a Razor project with authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page so your layout file isn't overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:

    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure
    - `~/Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
    - Select your data context class.
    - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
    - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
    - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
    - Select your existing layout page, or your layout file will be overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
        - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
        - `~/Views/Shared/_Layout.cshtml` for MVC projects
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
    - Select your data context class.
    - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
    - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
    - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Some Identity options are configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

Scaffold Identity into an MVC project without existing authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
    - Select your existing layout page, or your layout file will be overwritten with incorrect markup:

    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
    - Blazor Server apps created from the Blazor Server template (`blazorserver`) aren't configured for Razor Pages or MVC by default. Leave the layout page entry blank.
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
- Select **Add**.

Optional: Add the login partial (`_LoginPartial`) to the *Views/Shared/_Layout.cshtml* file:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - MvcNoAuth3</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet" href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-
value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index" class="navbar-brand">MvcNoAuth3</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-action="Contact">Contact</a></li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - MvcNoAuth3</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
```

```
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
        </script>
        <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

- Move the *Pages/Shared/_LoginPartial.cshtml* file to *Views/Shared/_LoginPartial.cshtml*

Identity is configured in *Areas/Identity/IdentityHostingStartup.cs*. For more information, see IHostingStartup.

The generated Identity database code requires Entity Framework Core Migrations. Create a migration and update the database. For example, run the following commands:

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Add-Migration CreateIdentitySchema
Update-Database
```

The "CreateIdentitySchema" name parameter for the `Add-Migration` command is arbitrary. `"CreateIdentitySchema"` describes the migration.

Call UseAuthentication after `UseStaticFiles`:

```
public class Startup
{

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAuthentication();
        app.UseMvcWithDefaultRoute();
    }
}
```

`UseHsts` is recommended but not required. For more information, see HTTP Strict Transport Security Protocol.

Scaffold Identity into an MVC project with authorization

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page so your layout file isn't overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure
    - `~/Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
  - Select your data context class.
  - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
  - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Run the Identity scaffolder:

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, select the options you want.
  - Select your existing layout page, or your layout file will be overwritten with incorrect markup. When an existing *_Layout.cshtml* file is selected, it is **not** overwritten. For example:
    - `~/Pages/Shared/_Layout.cshtml` for Razor Pages
    - `~/Views/Shared/_Layout.cshtml` for MVC projects
- To use your existing data context, select at least one file to override. You must select at least one file to add your data context.
  - Select your data context class.
  - Select **Add**.
- To create a new user context and possibly create a custom user class for Identity:
  - Select the **+** button to create a new **Data context class**. Accept the default value or specify a class (for example, `MyApplication.Data.ApplicationDbContext`).
  - Select **Add**.

Note: If you're creating a new user context, you don't have to select a file to override.

Delete the *Pages/Shared* folder and the files in that folder.

Create full Identity UI source

To maintain full control of the Identity UI, run the Identity scaffolder and select **Override all files**.

The following highlighted code shows the changes to replace the default Identity UI with Identity in an ASP.NET Core 2.1 web app. You might want to do this to have full control of the Identity UI.

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<IdentityUser, IdentityRole>()
        // services.AddDefaultIdentity<IdentityUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
        .AddRazorPagesOptions(options =>
        {
            options.AllowAreas = true;
            options.Conventions.AuthorizeAreaFolder("Identity", "/Account/Manage");
            options.Conventions.AuthorizeAreaPage("Identity", "/Account/Logout");
        });

    services.ConfigureApplicationCookie(options =>
    {
        options.LoginPath = $"/Identity/Account/Login";
        options.LogoutPath = $"/Identity/Account/Logout";
        options.AccessDeniedPath = $"/Identity/Account/AccessDenied";
    });

    // using Microsoft.AspNetCore.Identity.UI.Services;
    services.AddSingleton<IEmailSender, EmailSender>();
}
```

The default Identity is replaced in the following code:

```
services.AddIdentity<IdentityUser, IdentityRole>()
    // services.AddDefaultIdentity<IdentityUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

The following code sets the LoginPath, LogoutPath, and AccessDeniedPath:

```
services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = $"/Identity/Account/Login";
    options.LogoutPath = $"/Identity/Account/Logout";
    options.AccessDeniedPath = $"/Identity/Account/AccessDenied";
});
```

Register an `IEmailSender` implementation, for example:

```
// using Microsoft.AspNetCore.Identity.UI.Services;
services.AddSingleton<IEmailSender, EmailSender>();
```

```
public class EmailSender : IEmailSender
{
    public Task SendEmailAsync(string email, string subject, string message)
    {
        return Task.CompletedTask;
    }
}
```

Password configuration

If PasswordOptions are configured in `Startup.ConfigureServices`, `[StringLength]` attribute configuration might be required for the `Password` property in scaffolded Identity pages. `InputModel` `Password` properties are found in the following files:

- `Areas/Identity/Pages/Account/Register.cshtml.cs`
- `Areas/Identity/Pages/Account/ResetPassword.cshtml.cs`

Disable register page

To disable user registration:

- Scaffold Identity. Include Account.Register, Account.Login, and Account.RegisterConfirmation. For example:

```
dotnet aspnet-codegenerator identity -dc RPauth.Data.ApplicationDbContext --files
"Account.Register;Account.Login;Account.RegisterConfirmation"
```

- Update *Areas/Identity/Pages/Account/Register.cshtml.cs* so users can't register from this endpoint:

```
public class RegisterModel : PageModel
{
    public IActionResult OnGet()
    {
        return RedirectToPage("Login");
    }

    public IActionResult OnPost()
    {
        return RedirectToPage("Login");
    }
}
```

- Update *Areas/Identity/Pages/Account/Register.cshtml* to be consistent with the preceding changes:

```
@page
@model RegisterModel
@{
    ViewData["Title"] = "Go to Login";
}

<h1>@ViewData["Title"]</h1>

<li class="nav-item">
    <a class="nav-link text-dark" asp-area="Identity" asp-page="/Account/Login">Login</a>
</li>
```

- Comment out or remove the registration link from *Areas/Identity/Pages/Account/Login.cshtml*

```
@*
<p>
    <a asp-page="./Register" asp-route-returnUrl="@Model.ReturnUrl">Register as a new user</a>
</p>
*@
```

- Update the *Areas/Identity/Pages/Account/RegisterConfirmation* page.
  - Remove the code and links from the cshtml file.
  - Remove the confirmation code from the `PageModel`:

```
[AllowAnonymous]
  public class RegisterConfirmationModel : PageModel
  {
      public IActionResult OnGet()
      {
          return Page();
      }
  }
```

Use another app to add users

Provide a mechanism to add users outside the web app. Options to add users include:

- A dedicated admin web app.
- A console app.

The following code outlines one approach to adding users:

- A list of users is read into memory.
- A strong unique password is generated for each user.
- The user is added to the Identity database.
- The user is notified and told to change the password.

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<AppDbCntx>();
                context.Database.Migrate();

                var config = host.Services.GetRequiredService<IConfiguration>();
                var userList = config.GetSection("userList").Get<List<string>>();

                SeedData.Initialize(services, userList).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred adding users.");
            }
        }

        host.Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code outlines adding a user:

```
public static async Task Initialize(IServiceProvider serviceProvider,
                                    List<string> userList)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    foreach (var userName in userList)
    {
        var userPassword = GenerateSecurePassword();
        var userId = await EnsureUser(userManager, userName, userPassword);

        NotifyUser(userName, userPassword);
    }
}

private static async Task<string> EnsureUser(UserManager<IdentityUser> userManager,
                                             string userName, string userPassword)
{
    var user = await userManager.FindByNameAsync(userName);

    if (user == null)
    {
        user = new IdentityUser(userName)
        {
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, userPassword);
    }

    return user.Id;
}
```

A similar approach can be followed for production scenarios.

Additional resources

- Changes to authentication code to ASP.NET Core 2.1 and later

Add, download, and delete custom user data to Identity in an ASP.NET Core project

9/22/2020 • 13 minutes to read • Edit Online

By Rick Anderson

This article shows how to:

- Add custom user data to an ASP.NET Core web app.
- Mark the custom user data model with the PersonalDataAttribute attribute so it's automatically available for download and deletion. Making the data able to be downloaded and deleted helps meet GDPR requirements.

The project sample is created from a Razor Pages web app, but the instructions are similar for a ASP.NET Core MVC web app.

View or download sample code (how to download)

Prerequisites

.NET Core 3.0 SDK or later

.NET Core 2.2 SDK or later

Create a Razor web app

- Visual Studio
- .NET Core CLI

- From the Visual Studio **File** menu, select **New** > **Project**. Name the project **WebApp1** if you want to it match the namespace of the download sample code.
- Select **ASP.NET Core Web Application** > **OK**
- Select **ASP.NET Core 3.0** in the dropdown
- Select **Web Application** > **OK**
- Build and run the project.

- From the Visual Studio **File** menu, select **New** > **Project**. Name the project **WebApp1** if you want to it match the namespace of the download sample code.
- Select **ASP.NET Core Web Application** > **OK**
- Select **ASP.NET Core 2.2** in the dropdown
- Select **Web Application** > **OK**
- Build and run the project.

Run the Identity scaffolder

- Visual Studio
- .NET Core CLI

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, the following options:

    - Select the existing layout file *~/Pages/Shared/_Layout.cshtml*
    - Select the following files to override:
        - **Account/Register**
        - **Account/Manage/Index**
    - Select the **+** button to create a new **Data context class**. Accept the type (**WebApp1.Models.WebApp1Context** if the project is named **WebApp1**).
    - Select the **+** button to create a new **User class**. Accept the type (**WebApp1User** if the project is named **WebApp1**) > **Add**.
- Select **Add**.

Follow the instruction in Migrations, UseAuthentication, and layout to perform the following steps:

- Create a migration and update the database.
- Add `UseAuthentication` to `Startup.Configure`.
- Add `<partial name="_LoginPartial" />` to the layout file.
- Test the app:
    - Register a user
    - Select the new user name (next to the **Logout** link). You might need to expand the window or select the navigation bar icon to show the user name and other links.
    - Select the **Personal Data** tab.
    - Select the **Download** button and examined the *PersonalData.json* file.
    - Test the **Delete** button, which deletes the logged on user.

Add custom user data to the Identity DB

Update the `IdentityUser` derived class with custom properties. If you named the project WebApp1, the file is named *Areas/Identity/Data/WebApp1User.cs*. Update the file with the following code:

```
using System;
using Microsoft.AspNetCore.Identity;

namespace WebApp1.Areas.Identity.Data
{
    public class WebApp1User : IdentityUser
    {
        [PersonalData]
        public string Name { get; set; }
        [PersonalData]
        public DateTime DOB { get; set; }
    }
}
```

```
using Microsoft.AspNetCore.Identity;
using System;

namespace WebApp1.Areas.Identity.Data
{
    public class WebApp1User : IdentityUser
    {
        [PersonalData]
        public string Name { get; set; }
        [PersonalData]
        public DateTime DOB { get; set; }
    }
}
```

Properties with the PersonalData attribute are:

- Deleted when the *Areas/Identity/Pages/Account/Manage/DeletePersonalData.cshtml* Razor Page calls `UserManager.Delete`.
- Included in the downloaded data by the *Areas/Identity/Pages/Account/Manage/DownloadPersonalData.cshtml* Razor Page.

Update the Account/Manage/Index.cshtml page

Update the `InputModel` in *Areas/Identity/Pages/Account/Manage/Index.cshtml.cs* with the following highlighted code:

```
public partial class IndexModel : PageModel
{
    private readonly UserManager<WebApp1User> _userManager;
    private readonly SignInManager<WebApp1User> _signInManager;

    public IndexModel(
        UserManager<WebApp1User> userManager,
        SignInManager<WebApp1User> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }

    public string Username { get; set; }

    [TempData]
    public string StatusMessage { get; set; }

    [BindProperty]
    public InputModel Input { get; set; }

    public class InputModel
    {
        [Required]
        [DataType(DataType.Text)]
        [Display(Name = "Full name")]
        public string Name { get; set; }

        [Required]
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Phone]
        [Display(Name = "Phone number")]
        public string PhoneNumber { get; set; }
    }

    private async Task LoadAsync(WebApp1User user)
    {
        var userName = await _userManager.GetUserNameAsync(user);
        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);

        Username = userName;

        Input = new InputModel
        {
            Name = user.Name,
            DOB = user.DOB,
            PhoneNumber = phoneNumber
        };
    }

    public async Task<IActionResult> OnGetAsync()
    {
```

```
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound(
                $"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }

        await LoadAsync(user);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound(
                $"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }

        if (!ModelState.IsValid)
        {
            await LoadAsync(user);
            return Page();
        }

        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);
        if (Input.PhoneNumber != phoneNumber)
        {
            var setPhoneResult = await _userManager.SetPhoneNumberAsync(user,
                Input.PhoneNumber);

            if (!setPhoneResult.Succeeded)
            {
                var userId = await _userManager.GetUserIdAsync(user);
                throw new InvalidOperationException(
                    $"Unexpected error occurred setting phone number for user with ID '{userId}'.");
            }
        }

        if (Input.Name != user.Name)
        {
            user.Name = Input.Name;
        }

        if (Input.DOB != user.DOB)
        {
            user.DOB = Input.DOB;
        }

        await _userManager.UpdateAsync(user);

        await _signInManager.RefreshSignInAsync(user);
        StatusMessage = "Your profile has been updated";
        return RedirectToPage();
    }
}
```

Update the *Areas/Identity/Pages/Account/Manage/Index.cshtml* with the following highlighted markup:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Profile";
    ViewData["ActivePage"] = ManageNavPages.Index;
}

<h4>@ViewData["Title"]</h4>
<partial name="_StatusMessage" model="Model.StatusMessage" />
<div class="row">
    <div class="col-md-6">
        <form id="profile-form" method="post">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Username"></label>
                <input asp-for="Username" class="form-control" disabled />
            </div>
            <div class="form-group">
                <label asp-for="Input.Name"></label>
                <input asp-for="Input.Name" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="Input.DOB"></label>
                <input asp-for="Input.DOB" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="Input.PhoneNumber"></label>
                <input asp-for="Input.PhoneNumber" class="form-control" />
                <span asp-validation-for="Input.PhoneNumber"
                    class="text-danger"></span>
            </div>
            <button id="update-profile-button" type="submit"
                class="btn btn-primary">Save</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

```
public partial class IndexModel : PageModel
{
    private readonly UserManager<WebApp1User> _userManager;
    private readonly SignInManager<WebApp1User> _signInManager;
    private readonly IEmailSender _emailSender;

    public IndexModel(
        UserManager<WebApp1User> userManager,
        SignInManager<WebApp1User> signInManager,
        IEmailSender emailSender)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _emailSender = emailSender;
    }

    public string Username { get; set; }
    public bool IsEmailConfirmed { get; set; }

    [TempData]
    public string StatusMessage { get; set; }

    [BindProperty]
    public InputModel Input { get; set; }
```

```
    public class InputModel
    {
        [Required]
        [DataType(DataType.Text)]
        [Display(Name = "Full name")]
        public string Name { get; set; }

        [Required]
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Phone]
        [Display(Name = "Phone number")]
        public string PhoneNumber { get; set; }
    }

    public async Task<IActionResult> OnGetAsync()
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }

        var userName = await _userManager.GetUserNameAsync(user);
        var email = await _userManager.GetEmailAsync(user);
        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);

        Username = userName;

        Input = new InputModel
        {
            Name = user.Name,
            DOB = user.DOB,
            Email = email,
            PhoneNumber = phoneNumber
        };

        IsEmailConfirmed = await _userManager.IsEmailConfirmedAsync(user);

        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }

        var email = await _userManager.GetEmailAsync(user);
        if (Input.Email != email)
        {
            var setEmailResult = await _userManager.SetEmailAsync(user, Input.Email);
            if (!setEmailResult.Succeeded)
            {
                var userId = await _userManager.GetUserIdAsync(user);
                throw new InvalidOperationException($"Unexpected error occurred setting email for user with ID
```

```
'{userId}'.");
            }
        }

        if (Input.Name != user.Name)
        {
            user.Name = Input.Name;
        }

        if (Input.DOB != user.DOB)
        {
            user.DOB = Input.DOB;
        }

        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);
        if (Input.PhoneNumber != phoneNumber)
        {
            var setPhoneResult = await _userManager.SetPhoneNumberAsync(user, Input.PhoneNumber);
            if (!setPhoneResult.Succeeded)
            {
                var userId = await _userManager.GetUserIdAsync(user);
                throw new InvalidOperationException($"Unexpected error occurred setting phone number for user
with ID '{userId}'.");
            }
        }

        await _userManager.UpdateAsync(user);

        await _signInManager.RefreshSignInAsync(user);
        StatusMessage = "Your profile has been updated";
        return RedirectToPage();
    }

    public async Task<IActionResult> OnPostSendVerificationEmailAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID '{_userManager.GetUserId(User)}'.");
        }


        var userId = await _userManager.GetUserIdAsync(user);
        var email = await _userManager.GetEmailAsync(user);
        var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
        var callbackUrl = Url.Page(
            "/Account/ConfirmEmail",
            pageHandler: null,
            values: new { userId = userId, code = code },
            protocol: Request.Scheme);
        await _emailSender.SendEmailAsync(
            email,
            "Confirm your email",
            $"Please confirm your account by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking
here</a>.");

        StatusMessage = "Verification email sent. Please check your email.";
        return RedirectToPage();
    }
}
```

Update the *Areas/Identity/Pages/Account/Manage/Index.cshtml* with the following highlighted markup:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Profile";
    ViewData["ActivePage"] = ManageNavPages.Index;
}

<h4>@ViewData["Title"]</h4>
<partial name="_StatusMessage" for="StatusMessage" />
<div class="row">
    <div class="col-md-6">
        <form id="profile-form" method="post">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Username"></label>
                <input asp-for="Username" class="form-control" disabled />
            </div>
            <div class="form-group">
                <label asp-for="Input.Email"></label>
                @if (Model.IsEmailConfirmed)
                {
                    <div class="input-group">
                        <input asp-for="Input.Email" class="form-control" />
                        <span class="input-group-addon" aria-hidden="true"><span class="glyphicon glyphicon-ok
text-success"></span></span>
                    </div>
                }
                else
                {
                    <input asp-for="Input.Email" class="form-control" />
                    <button id="email-verification" type="submit" asp-page-handler="SendVerificationEmail"
class="btn btn-link">Send verification email</button>
                }
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.Name"></label>
                <input asp-for="Input.Name" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="Input.DOB"></label>
                <input asp-for="Input.DOB" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="Input.PhoneNumber"></label>
                <input asp-for="Input.PhoneNumber" class="form-control" />
                <span asp-validation-for="Input.PhoneNumber" class="text-danger"></span>
            </div>
            <button id="update-profile-button" type="submit" class="btn btn-primary">Save</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Update the Account/Register.cshtml page

Update the `InputModel` in *Areas/Identity/Pages/Account/Register.cshtml.cs* with the following highlighted code:

```
[AllowAnonymous]
public class RegisterModel : PageModel
{
    private readonly SignInManager<WebApp1User> _signInManager;
    private readonly UserManager<WebApp1User> _userManager;
```

```
    private readonly ILogger<RegisterModel> _logger;
    private readonly IEmailSender _emailSender;

    public RegisterModel(
        UserManager<WebApp1User> userManager,
        SignInManager<WebApp1User> signInManager,
        ILogger<RegisterModel> logger,
        IEmailSender emailSender)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _logger = logger;
        _emailSender = emailSender;
    }

    [BindProperty]
    public InputModel Input { get; set; }

    public string ReturnUrl { get; set; }

    public IList<AuthenticationScheme> ExternalLogins { get; set; }

    public class InputModel
    {
        [Required]
        [DataType(DataType.Text)]
        [Display(Name = "Full name")]
        public string Name { get; set; }

        [Required]
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.",
MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
        public string ConfirmPassword { get; set; }
    }

    public async Task OnGetAsync(string returnUrl = null)
    {
        ReturnUrl = returnUrl;
        ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
    }

    public async Task<IActionResult> OnPostAsync(string returnUrl = null)
    {
        returnUrl = returnUrl ?? Url.Content("~/");
        ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
        if (ModelState.IsValid)
        {
            var user = new WebApp1User {
                Name = Input.Name,
                DOB = Input.DOB,
                UserName = Input.Email,
                Email = Input.Email
```

```
            };
            var result = await _userManager.CreateAsync(user, Input.Password);
            if (result.Succeeded)
            {
                _logger.LogInformation("User created a new account with password.");

                var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
                code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                var callbackUrl = Url.Page(
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { area = "Identity", userId = user.Id, code = code },
                    protocol: Request.Scheme);

                await _emailSender.SendEmailAsync(Input.Email,
                    "Confirm your email",
                    $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                if (_userManager.Options.SignIn.RequireConfirmedAccount)
                {
                    return RedirectToPage("RegisterConfirmation", new { email = Input.Email });
                }
                else
                {
                    await _signInManager.SignInAsync(user, isPersistent: false);
                    return LocalRedirect(returnUrl);
                }
            }
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
        }

        // If we got this far, something failed, redisplay form
        return Page();
    }
}
```

Update the *Areas/Identity/Pages/Account/Register.cshtml* with the following highlighted markup:

```
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h1>@ViewData["Title"]</h1>

<div class="row">
    <div class="col-md-4">
        <form asp-route-returnUrl="@Model.ReturnUrl" method="post">
            <h4>Create a new account.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>

            <div class="form-group">
                <label asp-for="Input.Name"></label>
                <input asp-for="Input.Name" class="form-control" />
                <span asp-validation-for="Input.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.DOB"></label>
                <input asp-for="Input.DOB" class="form-control" />
                <span asp-validation-for="Input.DOB" class="text-danger"></span>
            </div>
```

```
            <div class="form-group">
                <label asp-for="Input.Email"></label>
                <input asp-for="Input.Email" class="form-control" />
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.Password"></label>
                <input asp-for="Input.Password" class="form-control" />
                <span asp-validation-for="Input.Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.ConfirmPassword"></label>
                <input asp-for="Input.ConfirmPassword" class="form-control" />
                <span asp-validation-for="Input.ConfirmPassword" class="text-danger"></span>
            </div>
            <button type="submit" class="btn btn-primary">Register</button>
        </form>
    </div>
    <div class="col-md-6 col-md-offset-2">
        <section>
            <h4>Use another service to register.</h4>
            <hr />
            @{
                if ((Model.ExternalLogins?.Count ?? 0) == 0)
                {
                    <div>
                        <p>
                            There are no external authentication services configured. See
                             <a href="https://go.microsoft.com/fwlink/?LinkID=532715">this article</a>
                            for details on setting up this ASP.NET application to support
                            logging in via external services.
                        </p>
                    </div>
                }
                else
                {
                    <form id="external-account" asp-page="./ExternalLogin"
                        asp-route-returnUrl="@Model.ReturnUrl" method="post"
                        class="form-horizontal">
                        <div>
                            <p>
                                @foreach (var provider in Model.ExternalLogins)
                                {
                                    <button type="submit" class="btn btn-primary" name="provider"
                                        value="@provider.Name"
                                        title="Log in using your @provider.DisplayName account">
                                            @provider.DisplayName</button>
                                }
                            </p>
                        </div>
                    </form>
                }
            }
        </section>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

```
[AllowAnonymous]
public class RegisterModel : PageModel
{
    private readonly SignInManager<WebApp1User> _signInManager;
    private readonly UserManager<WebApp1User> _userManager;
```

```
    private readonly ILogger<RegisterModel> _logger;
    private readonly IEmailSender _emailSender;

    public RegisterModel(
        UserManager<WebApp1User> userManager,
        SignInManager<WebApp1User> signInManager,
        ILogger<RegisterModel> logger,
        IEmailSender emailSender)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _logger = logger;
        _emailSender = emailSender;
    }

    [BindProperty]
    public InputModel Input { get; set; }

    public string ReturnUrl { get; set; }

    public class InputModel
    {
        [Required]
        [DataType(DataType.Text)]
        [Display(Name = "Full name")]
        public string Name { get; set; }

        [Required]
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; }

        [Required]
        [StringLength(100, ErrorMessage = "The {0} must be at least {2} and at max {1} characters long.",
MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
        public string ConfirmPassword { get; set; }
    }

    public void OnGet(string returnUrl = null)
    {
        ReturnUrl = returnUrl;
    }

    public async Task<IActionResult> OnPostAsync(string returnUrl = null)
    {
        returnUrl = returnUrl ?? Url.Content("~/");
        if (ModelState.IsValid)
        {
            var user = new WebApp1User {
                Name = Input.Name,
                DOB = Input.DOB,
                UserName = Input.Email,
                Email = Input.Email
            };
            var result = await _userManager.CreateAsync(user, Input.Password);
            if (result.Succeeded)
            {
```

```
                _logger.LogInformation("User created a new account with password.");

                var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
                var callbackUrl = Url.Page(
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { userId = user.Id, code = code },
                    protocol: Request.Scheme);

                await _emailSender.SendEmailAsync(Input.Email, "Confirm your email",
                    $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                await _signInManager.SignInAsync(user, isPersistent: false);
                return LocalRedirect(returnUrl);
            }
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
        }

        // If we got this far, something failed, redisplay form
        return Page();
    }
}
```

Update the *Areas/Identity/Pages/Account/Register.cshtml* with the following highlighted markup:

```
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h1>@ViewData["Title"]</h1>

<div class="row">
    <div class="col-md-4">
        <form asp-route-returnUrl="@Model.ReturnUrl" method="post">
            <h4>Create a new account.</h4>
            <hr />
            <div asp-validation-summary="All" class="text-danger"></div>

            <div class="form-group">
                <label asp-for="Input.Name"></label>
                <input asp-for="Input.Name" class="form-control" />
                <span asp-validation-for="Input.Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.DOB"></label>
                <input asp-for="Input.DOB" class="form-control" />
                <span asp-validation-for="Input.DOB" class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="Input.Email"></label>
                <input asp-for="Input.Email" class="form-control" />
                <span asp-validation-for="Input.Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.Password"></label>
                <input asp-for="Input.Password" class="form-control" />
                <span asp-validation-for="Input.Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Input.ConfirmPassword"></label>
                <input asp-for="Input.ConfirmPassword" class="form-control" />
                <span asp-validation-for="Input.ConfirmPassword" class="text-danger"></span>
            </div>
            <button type="submit" class="btn btn-primary">Register</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Build the project.

Add a migration for the custom user data

- Visual Studio
- .NET Core CLI

In the Visual Studio **Package Manager Console**:

```
Add-Migration CustomUserData
Update-Database
```

Test create, view, download, delete custom user data

Test the app:

- Register a new user.
- View the custom user data on the `/Identity/Account/Manage` page.
- Download and view the users personal data from the `/Identity/Account/Manage/PersonalData` page.

Add claims to Identity using IUserClaimsPrincipalFactory

> **NOTE**
>
> This section isn't an extension of the previous tutorial. To apply the following steps to the app built using the tutorial, see this GitHub issue.

Additional claims can be added to ASP.NET Core Identity by using the `IUserClaimsPrincipalFactory<T>` interface. This class can be added to the app in the `Startup.ConfigureServices` method. Add the custom implementation of the class as follows:

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddIdentity<ApplicationUser, IdentityRole>()
  .AddEntityFrameworkStores<ApplicationDbContext>()
  .AddDefaultTokenProviders();

 services.AddScoped<IUserClaimsPrincipalFactory<ApplicationUser>,
  AdditionalUserClaimsPrincipalFactory>();
```

The demo code uses the `ApplicationUser` class. This class adds an `IsAdmin` property which is used to add the additional claim.

```
public class ApplicationUser : IdentityUser
{
 public bool IsAdmin { get; set; }
}
```

The `AdditionalUserClaimsPrincipalFactory` implements the `UserClaimsPrincipalFactory` interface. A new role claim is added to the `ClaimsPrincipal`.

```
public class AdditionalUserClaimsPrincipalFactory
  : UserClaimsPrincipalFactory<ApplicationUser, IdentityRole>
{
 public AdditionalUserClaimsPrincipalFactory(
  UserManager<ApplicationUser> userManager,
  RoleManager<IdentityRole> roleManager,
  IOptions<IdentityOptions> optionsAccessor)
  : base(userManager, roleManager, optionsAccessor)
 {}

 public async override Task<ClaimsPrincipal> CreateAsync(ApplicationUser user)
 {
  var principal = await base.CreateAsync(user);
  var identity = (ClaimsIdentity)principal.Identity;

  var claims = new List<Claim>();
  if (user.IsAdmin)
  {
   claims.Add(new Claim(JwtClaimTypes.Role, "admin"));
  }
  else
  {
   claims.Add(new Claim(JwtClaimTypes.Role, "user"));
  }

  identity.AddClaims(claims);
  return principal;
 }
}
```

The additional claim can then be used in the app. In a Razor Page, the `IAuthorizationService` instance can be used to access the claim value.

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

@if ((await AuthorizationService.AuthorizeAsync(User, "IsAdmin")).Succeeded)
{
 <ul class="mr-auto navbar-nav">
  <li class="nav-item">
   <a class="nav-link" asp-controller="Admin" asp-action="Index">ADMIN</a>
  </li>
 </ul>
}
```

Authentication samples for ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Rick Anderson

The ASP.NET Core repository contains the following authentication samples in the *AspNetCore/src/Security/samples* folder:

- Claims transformation
- Cookie authentication
- Custom policy provider - IAuthorizationPolicyProvider
- Dynamic authentication schemes and options
- External claims
- Selecting between cookie and another authentication scheme based on the request
- Restricts access to static files

Run the samples

- Select a branch. For example, `release/3.1`
- Clone or download the ASP.NET Core repository.
- Verify you have installed the .NET Core SDK version matching the clone of the ASP.NET Core repository.
- Navigate to a sample in *AspNetCore/src/Security/samples* and run the sample with `dotnet run`.

The ASP.NET Core repository contains the following authentication samples in the *AspNetCore/src/Security/samples* folder:

- Claims transformation
- Cookie authentication
- Custom policy provider - IAuthorizationPolicyProvider
- Dynamic authentication schemes and options
- External claims
- Selecting between cookie and another authentication scheme based on the request
- Restricts access to static files

Run the samples

- Select a branch. For example, `release/2.1`
- Clone or download the ASP.NET Core repository.
- Verify you have installed the .NET Core SDK version matching the clone of the ASP.NET Core repository.
- Navigate to a sample in *AspNetCore/src/Security/samples* and run the sample with `dotnet run`.

Identity model customization in ASP.NET Core

9/22/2020 • 19 minutes to read • Edit Online

By Arthur Vickers

ASP.NET Core Identity provides a framework for managing and storing user accounts in ASP.NET Core apps. Identity is added to your project when **Individual User Accounts** is selected as the authentication mechanism. By default, Identity makes use of an Entity Framework (EF) Core data model. This article describes how to customize the Identity model.

Identity and EF Core Migrations

Before examining the model, it's useful to understand how Identity works with EF Core Migrations to create and update a database. At the top level, the process is:

1. Define or update a data model in code.
2. Add a Migration to translate this model into changes that can be applied to the database.
3. Check that the Migration correctly represents your intentions.
4. Apply the Migration to update the database to be in sync with the model.
5. Repeat steps 1 through 4 to further refine the model and keep the database in sync.

Use one of the following approaches to add and apply Migrations:

- The **Package Manager Console** (PMC) window if using Visual Studio. For more information, see EF Core PMC tools.
- The .NET Core CLI if using the command line. For more information, see EF Core .NET command line tools.
- Clicking the **Apply Migrations** button on the error page when the app is run.

ASP.NET Core has a development-time error page handler. The handler can apply migrations when the app is run. Production apps typically generate SQL scripts from the migrations and deploy database changes as part of a controlled app and database deployment.

When a new app using Identity is created, steps 1 and 2 above have already been completed. That is, the initial data model already exists, and the initial migration has been added to the project. The initial migration still needs to be applied to the database. The initial migration can be applied via one of the following approaches:

- Run `Update-Database` in PMC.
- Run `dotnet ef database update` in a command shell.
- Click the **Apply Migrations** button on the error page when the app is run.

Repeat the preceding steps as changes are made to the model.

The Identity model

Entity types

The Identity model consists of the following entity types.

ENTITY TYPE	DESCRIPTION
`User`	Represents the user.

ENTITY TYPE	DESCRIPTION
`Role`	Represents a role.
`UserClaim`	Represents a claim that a user possesses.
`UserToken`	Represents an authentication token for a user.
`UserLogin`	Associates a user with a login.
`RoleClaim`	Represents a claim that's granted to all users within a role.
`UserRole`	A join entity that associates users and roles.

Entity type relationships

The entity types are related to each other in the following ways:

- Each `User` can have many `UserClaims`.
- Each `User` can have many `UserLogins`.
- Each `User` can have many `UserTokens`.
- Each `Role` can have many associated `RoleClaims`.
- Each `User` can have many associated `Roles`, and each `Role` can be associated with many `Users`. This is a many-to-many relationship that requires a join table in the database. The join table is represented by the `UserRole` entity.

Default model configuration

Identity defines many *context classes* that inherit from DbContext to configure and use the model. This configuration is done using the EF Core Code First Fluent API in the OnModelCreating method of the context class. The default configuration is:

```
builder.Entity<TUser>(b =>
{
    // Primary key
    b.HasKey(u => u.Id);

    // Indexes for "normalized" username and email, to allow efficient lookups
    b.HasIndex(u => u.NormalizedUserName).HasName("UserNameIndex").IsUnique();
    b.HasIndex(u => u.NormalizedEmail).HasName("EmailIndex");

    // Maps to the AspNetUsers table
    b.ToTable("AspNetUsers");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(u => u.ConcurrencyStamp).IsConcurrencyToken();

    // Limit the size of columns to use efficient database types
    b.Property(u => u.UserName).HasMaxLength(256);
    b.Property(u => u.NormalizedUserName).HasMaxLength(256);
    b.Property(u => u.Email).HasMaxLength(256);
    b.Property(u => u.NormalizedEmail).HasMaxLength(256);

    // The relationships between User and other entity types
    // Note that these relationships are configured with no navigation properties

    // Each User can have many UserClaims
    b.HasMany<TUserClaim>().WithOne().HasForeignKey(uc => uc.UserId).IsRequired();

    // Each User can have many UserLogins
```

```
    b.HasMany<TUserLogin>().WithOne().HasForeignKey(ul => ul.UserId).IsRequired();

    // Each User can have many UserTokens
    b.HasMany<TUserToken>().WithOne().HasForeignKey(ut => ut.UserId).IsRequired();

    // Each User can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur => ur.UserId).IsRequired();
});

builder.Entity<TUserClaim>(b =>
{
    // Primary key
    b.HasKey(uc => uc.Id);

    // Maps to the AspNetUserClaims table
    b.ToTable("AspNetUserClaims");
});

builder.Entity<TUserLogin>(b =>
{
    // Composite primary key consisting of the LoginProvider and the key to use
    // with that provider
    b.HasKey(l => new { l.LoginProvider, l.ProviderKey });

    // Limit the size of the composite key columns due to common DB restrictions
    b.Property(l => l.LoginProvider).HasMaxLength(128);
    b.Property(l => l.ProviderKey).HasMaxLength(128);

    // Maps to the AspNetUserLogins table
    b.ToTable("AspNetUserLogins");
});

builder.Entity<TUserToken>(b =>
{
    // Composite primary key consisting of the UserId, LoginProvider and Name
    b.HasKey(t => new { t.UserId, t.LoginProvider, t.Name });

    // Limit the size of the composite key columns due to common DB restrictions
    b.Property(t => t.LoginProvider).HasMaxLength(maxKeyLength);
    b.Property(t => t.Name).HasMaxLength(maxKeyLength);

    // Maps to the AspNetUserTokens table
    b.ToTable("AspNetUserTokens");
});

builder.Entity<TRole>(b =>
{
    // Primary key
    b.HasKey(r => r.Id);

    // Index for "normalized" role name to allow efficient lookups
    b.HasIndex(r => r.NormalizedName).HasName("RoleNameIndex").IsUnique();

    // Maps to the AspNetRoles table
    b.ToTable("AspNetRoles");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(r => r.ConcurrencyStamp).IsConcurrencyToken();

    // Limit the size of columns to use efficient database types
    b.Property(u => u.Name).HasMaxLength(256);
    b.Property(u => u.NormalizedName).HasMaxLength(256);

    // The relationships between Role and other entity types
    // Note that these relationships are configured with no navigation properties

    // Each Role can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur => ur.RoleId).IsRequired();
```

```
    // Each Role can have many associated RoleClaims
    b.HasMany<TRoleClaim>().WithOne().HasForeignKey(rc => rc.RoleId).IsRequired();
});

builder.Entity<TRoleClaim>(b =>
{
    // Primary key
    b.HasKey(rc => rc.Id);

    // Maps to the AspNetRoleClaims table
    b.ToTable("AspNetRoleClaims");
});

builder.Entity<TUserRole>(b =>
{
    // Primary key
    b.HasKey(r => new { r.UserId, r.RoleId });

    // Maps to the AspNetUserRoles table
    b.ToTable("AspNetUserRoles");
});
```

Model generic types

Identity defines default Common Language Runtime (CLR) types for each of the entity types listed above. These types are all prefixed with *Identity*:

- `IdentityUser`
- `IdentityRole`
- `IdentityUserClaim`
- `IdentityUserToken`
- `IdentityUserLogin`
- `IdentityRoleClaim`
- `IdentityUserRole`

Rather than using these types directly, the types can be used as base classes for the app's own types. The `DbContext` classes defined by Identity are generic, such that different CLR types can be used for one or more of the entity types in the model. These generic types also allow the `User` primary key (PK) data type to be changed.

When using Identity with support for roles, an IdentityDbContext class should be used. For example:

```
// Uses all the built-in Identity types
// Uses `string` as the key type
public class IdentityDbContext
    : IdentityDbContext<IdentityUser, IdentityRole, string>
{
}

// Uses the built-in Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityDbContext<TUser>
    : IdentityDbContext<TUser, IdentityRole, string>
        where TUser : IdentityUser
{
}

// Uses the built-in Identity types except with custom User and Role types
// The key type is defined by TKey
public class IdentityDbContext<TUser, TRole, TKey> : IdentityDbContext<
    TUser, TRole, TKey, IdentityUserClaim<TKey>, IdentityUserRole<TKey>,
    IdentityUserLogin<TKey>, IdentityRoleClaim<TKey>, IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TRole : IdentityRole<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic arguments
// The key type is defined by TKey
public abstract class IdentityDbContext<
    TUser, TRole, TKey, TUserClaim, TUserRole, TUserLogin, TRoleClaim, TUserToken>
    : IdentityUserContext<TUser, TKey, TUserClaim, TUserLogin, TUserToken>
         where TUser : IdentityUser<TKey>
         where TRole : IdentityRole<TKey>
         where TKey : IEquatable<TKey>
         where TUserClaim : IdentityUserClaim<TKey>
         where TUserRole : IdentityUserRole<TKey>
         where TUserLogin : IdentityUserLogin<TKey>
         where TRoleClaim : IdentityRoleClaim<TKey>
         where TUserToken : IdentityUserToken<TKey>
```

It's also possible to use Identity without roles (only claims), in which case an IdentityUserContext<TUser> class should be used:

```
// Uses the built-in non-role Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityUserContext<TUser>
    : IdentityUserContext<TUser, string>
        where TUser : IdentityUser
{
}

// Uses the built-in non-role Identity types except with a custom User type
// The key type is defined by TKey
public class IdentityUserContext<TUser, TKey> : IdentityUserContext<
    TUser, TKey, IdentityUserClaim<TKey>, IdentityUserLogin<TKey>,
    IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic arguments, with no roles
// The key type is defined by TKey
public abstract class IdentityUserContext<
    TUser, TKey, TUserClaim, TUserLogin, TUserToken> : DbContext
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
        where TUserClaim : IdentityUserClaim<TKey>
        where TUserLogin : IdentityUserLogin<TKey>
        where TUserToken : IdentityUserToken<TKey>
{
}
```

Customize the model

The starting point for model customization is to derive from the appropriate context type. See the Model generic types section. This context type is customarily called `ApplicationDbContext` and is created by the ASP.NET Core templates.

The context is used to configure the model in two ways:

- Supplying entity and key types for the generic type parameters.
- Overriding `OnModelCreating` to modify the mapping of these types.

When overriding `OnModelCreating`, `base.OnModelCreating` should be called first; the overriding configuration should be called next. EF Core generally has a last-one-wins policy for configuration. For example, if the `ToTable` method for an entity type is called first with one table name and then again later with a different table name, the table name in the second call is used.

Custom user data

Custom user data is supported by inheriting from `IdentityUser`. It's customary to name this type `ApplicationUser`:

```
public class ApplicationUser : IdentityUser
{
    public string CustomTag { get; set; }
}
```

Use the `ApplicationUser` type as a generic argument for the context:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
    }
}
```

There's no need to override `OnModelCreating` in the `ApplicationDbContext` class. EF Core maps the `CustomTag` property by convention. However, the database needs to be updated to create a new `CustomTag` column. To create the column, add a migration, and then update the database as described in Identity and EF Core Migrations.

Update *Pages/Shared/_LoginPartial.cshtml* and replace `IdentityUser` with `ApplicationUser`:

```
@using Microsoft.AspNetCore.Identity
@using WebApp1.Areas.Identity.Data
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager
```

Update *Areas/Identity/IdentityHostingStartup.cs* or `Startup.ConfigureServices` and replace `IdentityUser` with `ApplicationUser`.

```
services.AddIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultUI();
```

In ASP.NET Core 2.1 or later, Identity is provided as a Razor Class Library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`. For more information, see:

- Scaffold Identity
- Add, download, and delete custom user data to Identity

Change the primary key type

A change to the PK column's data type after the database has been created is problematic on many database systems. Changing the PK typically involves dropping and re-creating the table. Therefore, key types should be specified in the initial migration when the database is created.

Follow these steps to change the PK type:

1. If the database was created before the PK change, run `Drop-Database` (PMC) or `dotnet ef database drop` (.NET Core CLI) to delete it.
2. After confirming deletion of the database, remove the initial migration with `Remove-Migration` (PMC) or `dotnet ef migrations remove` (.NET Core CLI).
3. Update the `ApplicationDbContext` class to derive from IdentityDbContext<TUser,TRole,TKey>. Specify the new key type for `TKey`. For example, to use a `Guid` key type:

```
public class ApplicationDbContext
    : IdentityDbContext<IdentityUser<Guid>, IdentityRole<Guid>, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

In the preceding code, the generic classes IdentityUser<TKey> and IdentityRole<TKey> must be specified to use the new key type.

In the preceding code, the generic classes IdentityUser<TKey> and IdentityRole<TKey> must be specified to use the new key type.

`Startup.ConfigureServices` must be updated to use the generic user:

```
services.AddDefaultIdentity<IdentityUser<Guid>>()
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

```
services.AddIdentity<IdentityUser<Guid>, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

```
services.AddIdentity<IdentityUser<Guid>, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
        .AddDefaultTokenProviders();
```

4. If a custom `ApplicationUser` class is being used, update the class to inherit from `IdentityUser`. For example:

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

public class ApplicationUser : IdentityUser<Guid>
{
    public string CustomTag { get; set; }
}
```

```
using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationUser : IdentityUser<Guid>
{
    public string CustomTag { get; set; }
}
```

Update `ApplicationDbContext` to reference the custom `ApplicationUser` class:

```
public class ApplicationDbContext
    : IdentityDbContext<ApplicationUser, IdentityRole<Guid>, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

```
services.AddIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultUI()
        .AddDefaultTokenProviders();
```

The primary key's data type is inferred by analyzing the DbContext object.

In ASP.NET Core 2.1 or later, Identity is provided as a Razor Class Library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`.

```
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```

The primary key's data type is inferred by analyzing the DbContext object.

```
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
        .AddDefaultTokenProviders();
```

The AddEntityFrameworkStores method accepts a `TKey` type indicating the primary key's data type.

5. If a custom `ApplicationRole` class is being used, update the class to inherit from `IdentityRole<TKey>`. For example:

```
using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationRole : IdentityRole<Guid>
{
    public string Description { get; set; }
}
```

Update `ApplicationDbContext` to reference the custom `ApplicationRole` class. For example, the following class references a custom `ApplicationUser` and a custom `ApplicationRole`:

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultUI()
            .AddDefaultTokenProviders();

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

The primary key's data type is inferred by analyzing the DbContext object.

In ASP.NET Core 2.1 or later, Identity is provided as a Razor Class Library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`.

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();

    services.AddMvc()
        .AddRazorPagesOptions(options =>
        {
            options.Conventions.AuthorizeFolder("/Account/Manage");
            options.Conventions.AuthorizePage("/Account/Logout");
        });

    services.AddSingleton<IEmailSender, EmailSender>();
}
```

The primary key's data type is inferred by analyzing the DbContext object.

```
using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext, Guid>()
            .AddDefaultTokenProviders();

    services.AddMvc();

    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
}
```

The AddEntityFrameworkStores method accepts a `TKey` type indicating the primary key's data type.

Add navigation properties

Changing the model configuration for relationships can be more difficult than making other changes. Care must be taken to replace the existing relationships rather than create new, additional relationships. In particular, the changed relationship must specify the same foreign key (FK) property as the existing relationship. For example, the relationship between `Users` and `UserClaims` is, by default, specified as follows:

```
builder.Entity<TUser>(b =>
{
    // Each User can have many UserClaims
    b.HasMany<TUserClaim>()
     .WithOne()
     .HasForeignKey(uc => uc.UserId)
     .IsRequired();
});
```

The FK for this relationship is specified as the `UserClaim.UserId` property. `HasMany` and `WithOne` are called without arguments to create the relationship without navigation properties.

Add a navigation property to `ApplicationUser` that allows associated `UserClaims` to be referenced from the user:

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; }
}
```

The `TKey` for `IdentityUserClaim<TKey>` is the type specified for the PK of users. In this case, `TKey` is `string` because the defaults are being used. It's **not** the PK type for the `UserClaim` entity type.

Now that the navigation property exists, it must be configured in `OnModelCreating`:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();
        });
    }
}
```

Notice that relationship is configured exactly as it was before, only with a navigation property specified in the call to `HasMany`.

The navigation properties only exist in the EF model, not the database. Because the FK for the relationship hasn't changed, this kind of model change doesn't require the database to be updated. This can be checked by adding a migration after making the change. The `Up` and `Down` methods are empty.

Add all User navigation properties

Using the section above as guidance, the following example configures unidirectional navigation properties for all relationships on User:

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; }
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set; }
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set; }
    public virtual ICollection<IdentityUserRole<string>> UserRoles { get; set; }
}
```

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne()
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });
    }
}
```

Add User and Role navigation properties

Using the section above as guidance, the following example configures navigation properties for all relationships on User and Role:

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set; }
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set; }
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set; }
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}
```

```
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        IdentityUserClaim<string>, ApplicationUserRole, IdentityUserLogin<string>,
        IdentityRoleClaim<string>, IdentityUserToken<string>>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();
        });

    }
}
```

Notes:

- This example also includes the `UserRole` join entity, which is needed to navigate the many-to-many relationship from Users to Roles.
- Remember to change the types of the navigation properties to reflect that `Application{...}` types are now being used instead of `Identity{...}` types.
- Remember to use the `Application{...}` in the generic `ApplicationContext` definition.

Add all navigation properties

Using the section above as guidance, the following example configures navigation properties for all relationships

on all entity types:

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<ApplicationUserClaim> Claims { get; set; }
    public virtual ICollection<ApplicationUserLogin> Logins { get; set; }
    public virtual ICollection<ApplicationUserToken> Tokens { get; set; }
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
    public virtual ICollection<ApplicationRoleClaim> RoleClaims { get; set; }
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserClaim : IdentityUserClaim<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationUserLogin : IdentityUserLogin<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationRoleClaim : IdentityRoleClaim<string>
{
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserToken : IdentityUserToken<string>
{
    public virtual ApplicationUser User { get; set; }
}
```

```
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        ApplicationUserClaim, ApplicationUserRole, ApplicationUserLogin,
        ApplicationRoleClaim, ApplicationUserToken>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne(e => e.User)
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne(e => e.User)
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne(e => e.User)
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();

            // Each Role can have many associated RoleClaims
            b.HasMany(e => e.RoleClaims)
                .WithOne(e => e.Role)
                .HasForeignKey(rc => rc.RoleId)
                .IsRequired();
        });
    }
}
```

Use composite keys

The preceding sections demonstrated changing the type of key used in the Identity model. Changing the Identity key model to use composite keys isn't supported or recommended. Using a composite key with Identity involves changing how the Identity manager code interacts with the model. This customization is beyond the scope of this document.

Change table/column names and facets

To change the names of tables and columns, call `base.OnModelCreating`. Then, add configuration to override any of the defaults. For example, to change the name of all the Identity tables:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.ToTable("MyUsers");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.ToTable("MyUserClaims");
    });

    modelBuilder.Entity<IdentityUserLogin<string>>(b =>
    {
        b.ToTable("MyUserLogins");
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.ToTable("MyUserTokens");
    });

    modelBuilder.Entity<IdentityRole>(b =>
    {
        b.ToTable("MyRoles");
    });

    modelBuilder.Entity<IdentityRoleClaim<string>>(b =>
    {
        b.ToTable("MyRoleClaims");
    });

    modelBuilder.Entity<IdentityUserRole<string>>(b =>
    {
        b.ToTable("MyUserRoles");
    });
}
```

These examples use the default Identity types. If using an app type such as `ApplicationUser`, configure that type instead of the default type.

The following example changes some column names:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.Property(e => e.Email).HasColumnName("EMail");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.Property(e => e.ClaimType).HasColumnName("CType");
        b.Property(e => e.ClaimValue).HasColumnName("CValue");
    });
}
```

Some types of database columns can be configured with certain *facets* (for example, the maximum `string` length allowed). The following example sets column maximum lengths for several `string` properties in the model:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.Property(u => u.UserName).HasMaxLength(128);
        b.Property(u => u.NormalizedUserName).HasMaxLength(128);
        b.Property(u => u.Email).HasMaxLength(128);
        b.Property(u => u.NormalizedEmail).HasMaxLength(128);
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.Property(t => t.LoginProvider).HasMaxLength(128);
        b.Property(t => t.Name).HasMaxLength(128);
    });
}
```

Map to a different schema

Schemas can behave differently across database providers. For SQL Server, the default is to create all tables in the *dbo* schema. The tables can be created in a different schema. For example:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.HasDefaultSchema("notdbo");
}
```

Lazy loading

In this section, support for lazy-loading proxies in the Identity model is added. Lazy-loading is useful since it allows navigation properties to be used without first ensuring they're loaded.

Entity types can be made suitable for lazy-loading in several ways, as described in the EF Core documentation. For simplicity, use lazy-loading proxies, which requires:

- Installation of the Microsoft.EntityFrameworkCore.Proxies package.
- A call to UseLazyLoadingProxies inside AddDbContext<TContext>.
- Public entity types with `public virtual` navigation properties.

The following example demonstrates calling `UseLazyLoadingProxies` in `Startup.ConfigureServices`:

```
services
    .AddDbContext<ApplicationDbContext>(
        b => b.UseSqlServer(connectionString)
              .UseLazyLoadingProxies())
    .AddDefaultIdentity<ApplicationUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Refer to the preceding examples for guidance on adding navigation properties to the entity types.

Additional resources

- Scaffold Identity in ASP.NET Core projects

Community OSS authentication options for ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

This page contains community-provided, open source authentication options for ASP.NET Core. This page is periodically updated as new providers become available.

OSS authentication providers

The list below is sorted alphabetically.

NAME	DESCRIPTION
AspNet.Security.OpenIdConnect.Server (ASOS)	ASOS is a low-level, protocol-first OpenID Connect server framework for ASP.NET Core and OWIN/Katana.
Gluu Server	Enterprise ready, open source software for identity, access management (IAM), and single sign-on (SSO). For more information, see the Gluu Product Documentation.
IdentityServer	IdentityServer is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core, officially certified by the OpenID Foundation and under governance of the .NET Foundation. For more information, see Welcome to IdentityServer4 (Documentation).
OpenIddict	OpenIddict is an easy-to-use OpenID Connect server for ASP.NET Core.

To add a provider, edit this page.

Configure ASP.NET Core Identity

9/22/2020 • 4 minutes to read • Edit Online

ASP.NET Core Identity uses default values for settings such as password policy, lockout, and cookie configuration. These settings can be overridden in the `Startup` class.

Identity options

The IdentityOptions class represents the options that can be used to configure the Identity system. `IdentityOptions` must be set **after** calling `AddIdentity` or `AddDefaultIdentity`.

Claims Identity

IdentityOptions.ClaimsIdentity specifies the ClaimsIdentityOptions with the properties shown in the following table.

PROPERTY	DESCRIPTION	DEFAULT
RoleClaimType	Gets or sets the claim type used for a role claim.	ClaimTypes.Role
SecurityStampClaimType	Gets or sets the claim type used for the security stamp claim.	`AspNet.Identity.SecurityStamp`
UserIdClaimType	Gets or sets the claim type used for the user identifier claim.	ClaimTypes.NameIdentifier
UserNameClaimType	Gets or sets the claim type used for the user name claim.	ClaimTypes.Name

Lockout

Lockout is set in the PasswordSignInAsync method:

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
            Input.Password, Input.RememberMe,
            lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl,
                Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

The preceding code is based on the `Login` Identity template.

Lockout options are set in `StartUp.ConfigureServices`:

```
services.Configure<IdentityOptions>(options =>
{
    // Default Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;
});
```

The preceding code sets the IdentityOptions LockoutOptions with default values.

A successful authentication resets the failed access attempts count and resets the clock.

IdentityOptions.Lockout specifies the LockoutOptions with the properties shown in the table.

PROPERTY	DESCRIPTION	DEFAULT
AllowedForNewUsers	Determines if a new user can be locked out.	`true`
DefaultLockoutTimeSpan	The amount of time a user is locked out when a lockout occurs.	5 minutes

PROPERTY	DESCRIPTION	DEFAULT
MaxFailedAccessAttempts	The number of failed access attempts until a user is locked out, if lockout is enabled.	5

Password

By default, Identity requires that passwords contain an uppercase character, lowercase character, a digit, and a non-alphanumeric character. Passwords must be at least six characters long.

Passwords are configured with:

- PasswordOptions in `Startup.ConfigureServices`.
- `[StringLength]` attributes of `Password` properties if Identity is scaffolded into the app. `InputModel` `Password` properties are found in the following files:
  - `Areas/Identity/Pages/Account/Register.cshtml.cs`
  - `Areas/Identity/Pages/Account/ResetPassword.cshtml.cs`

```
services.Configure<IdentityOptions>(options =>
{
    // Default Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;
});
```

IdentityOptions.Password specifies the PasswordOptions with the properties shown in the table.

PROPERTY	DESCRIPTION	DEFAULT
RequireDigit	Requires a number between 0-9 in the password.	`true`
RequiredLength	The minimum length of the password.	6
RequireLowercase	Requires a lowercase character in the password.	`true`
RequireNonAlphanumeric	Requires a non-alphanumeric character in the password.	`true`
RequiredUniqueChars	Only applies to ASP.NET Core 2.0 or later. Requires the number of distinct characters in the password.	1
RequireUppercase	Requires an uppercase character in the password.	`true`

Sign-in

The following code sets `SignIn` settings (to default values):

```
services.Configure<IdentityOptions>(options =>
{
    // Default SignIn settings.
    options.SignIn.RequireConfirmedEmail = false;
    options.SignIn.RequireConfirmedPhoneNumber = false;
});
```

IdentityOptions.SignIn specifies the SignInOptions with the properties shown in the table.

PROPERTY	DESCRIPTION	DEFAULT
RequireConfirmedEmail	Requires a confirmed email to sign in.	`false`
RequireConfirmedPhoneNumber	Requires a confirmed phone number to sign in.	`false`

Tokens

IdentityOptions.Tokens specifies the TokenOptions with the properties shown in the table.

PROPERTY	DESCRIPTION
AuthenticatorTokenProvider	Gets or sets the `AuthenticatorTokenProvider` used to validate two-factor sign-ins with an authenticator.
ChangeEmailTokenProvider	Gets or sets the `ChangeEmailTokenProvider` used to generate tokens used in email change confirmation emails.
ChangePhoneNumberTokenProvider	Gets or sets the `ChangePhoneNumberTokenProvider` used to generate tokens used when changing phone numbers.
EmailConfirmationTokenProvider	Gets or sets the token provider used to generate tokens used in account confirmation emails.
PasswordResetTokenProvider	Gets or sets the IUserTwoFactorTokenProvider<TUser> used to generate tokens used in password reset emails.
ProviderMap	Used to construct a User Token Provider with the key used as the provider's name.

User

```
services.Configure<IdentityOptions>(options =>
{
    // Default User settings.
    options.User.AllowedUserNameCharacters =
            "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;

});
```

IdentityOptions.User specifies the UserOptions with the properties shown in the table.

PROPERTY	DESCRIPTION	DEFAULT

PROPERTY	DESCRIPTION	DEFAULT
AllowedUserNameCharacters	Allowed characters in the username.	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 -._@+
RequireUniqueEmail	Requires each user to have a unique email.	`false`

Cookie settings

Configure the app's cookie in `Startup.ConfigureServices`. ConfigureApplicationCookie must be called **after** calling `AddIdentity` or `AddDefaultIdentity`.

```
services.ConfigureApplicationCookie(options =>
{
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Identity/Account/Login";
    // ReturnUrlParameter requires
    //using Microsoft.AspNetCore.Authentication.Cookies;
    options.ReturnUrlParameter = CookieAuthenticationDefaults.ReturnUrlParameter;
    options.SlidingExpiration = true;
});
```

For more information, see CookieAuthenticationOptions.

Password Hasher options

PasswordHasherOptions gets and sets options for password hashing.

OPTION	DESCRIPTION
CompatibilityMode	The compatibility mode used when hashing new passwords. Defaults to IdentityV3. The first byte of a hashed password, called a *format marker*, specifies the version of the hashing algorithm used to hash the password. When verifying a password against a hash, the VerifyHashedPassword method selects the correct algorithm based on the first byte. A client is able to authenticate regardless of which version of the algorithm was used to hash the password. Setting the compatibility mode affects the hashing of *new passwords*.
IterationCount	The number of iterations used when hashing passwords using PBKDF2. This value is only used when the CompatibilityMode is set to IdentityV3. The value must be a positive integer and defaults to `10000`.

In the following example, the IterationCount is set to `12000` in `Startup.ConfigureServices`:

```
// using Microsoft.AspNetCore.Identity;

services.Configure<PasswordHasherOptions>(option =>
{
    option.IterationCount = 12000;
});
```

Globally require all users to be authenticated

For information on how to globally require all users to be authenticated, see Require authenticated users.

Configure Windows Authentication in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Scott Addie

Windows Authentication (also known as Negotiate, Kerberos, or NTLM authentication) can be configured for ASP.NET Core apps hosted with IIS, Kestrel, or HTTP.sys.

Windows Authentication (also known as Negotiate, Kerberos, or NTLM authentication) can be configured for ASP.NET Core apps hosted with IIS or HTTP.sys.

Windows Authentication relies on the operating system to authenticate users of ASP.NET Core apps. You can use Windows Authentication when your server runs on a corporate network using Active Directory domain identities or Windows accounts to identify users. Windows Authentication is best suited to intranet environments where users, client apps, and web servers belong to the same Windows domain.

> **NOTE**
> Windows Authentication isn't supported with HTTP/2. Authentication challenges can be sent on HTTP/2 responses, but the client must downgrade to HTTP/1.1 before authenticating.

Proxy and load balancer scenarios

Windows Authentication is a stateful scenario primarily used in an intranet, where a proxy or load balancer doesn't usually handle traffic between clients and servers. If a proxy or load balancer is used, Windows Authentication only works if the proxy or load balancer:

- Handles the authentication.
- Passes the user authentication information to the app (for example, in a request header), which acts on the authentication information.

An alternative to Windows Authentication in environments where proxies and load balancers are used is Active Directory Federated Services (ADFS) with OpenID Connect (OIDC).

IIS/IIS Express

Add authentication services by invoking AddAuthentication (Microsoft.AspNetCore.Server.IISIntegration namespace) in `Startup.ConfigureServices`:

```
services.AddAuthentication(IISDefaults.AuthenticationScheme);
```

Launch settings (debugger)

Configuration for launch settings only affects the *Properties/launchSettings.json* file for IIS Express and doesn't configure IIS for Windows Authentication. Server configuration is explained in the IIS section.

The **Web Application** template available via Visual Studio or the .NET Core CLI can be configured to support Windows Authentication, which updates the *Properties/launchSettings.json* file automatically.

- Visual Studio

- .NET Core CLI

New project

1. Create a new project.
2. Select **ASP.NET Core Web Application**. Select **Next**.
3. Provide a name in the **Project name** field. Confirm the **Location** entry is correct or provide a location for the project. Select **Create**.
4. Select **Change** under **Authentication**.
5. In the **Change Authentication** window, select **Windows Authentication**. Select **OK**.
6. Select **Web Application**.
7. Select **Create**.

Run the app. The username appears in the rendered app's user interface.

Existing project

The project's properties enable Windows Authentication and disable Anonymous Authentication:

1. Right-click the project in **Solution Explorer** and select **Properties**.
2. Select the **Debug** tab.
3. Clear the check box for **Enable Anonymous Authentication**.
4. Select the check box for **Enable Windows Authentication**.
5. Save and close the property page.

Alternatively, the properties can be configured in the `iisSettings` node of the *launchSettings.json* file:

```
"iisSettings": {
    "windowsAuthentication": true,
    "anonymousAuthentication": false,
    "iisExpress": {
        "applicationUrl": "http://localhost:52171/",
        "sslPort": 44308
    }
}
```

When modifying an existing project, confirm that the project file includes a package reference for the Microsoft.AspNetCore.App metapackage **or** the Microsoft.AspNetCore.Authentication NuGet package.

IIS

IIS uses the ASP.NET Core Module to host ASP.NET Core apps. Windows Authentication is configured for IIS via the *web.config* file. The following sections show how to:

- Provide a local *web.config* file that activates Windows Authentication on the server when the app is deployed.
- Use the IIS Manager to configure the *web.config* file of an ASP.NET Core app that has already been deployed to the server.

If you haven't already done so, enable IIS to host ASP.NET Core apps. For more information, see Host ASP.NET Core on Windows with IIS.

Enable the IIS Role Service for Windows Authentication. For more information, see Enable Windows Authentication in IIS Role Services (see Step 2).

IIS Integration Middleware is configured to automatically authenticate requests by default. For more information, see Host ASP.NET Core on Windows with IIS: IIS options (AutomaticAuthentication).

The ASP.NET Core Module is configured to forward the Windows Authentication token to the app by default. For more information, see ASP.NET Core Module configuration reference: Attributes of the aspNetCore element.

Use **either** of the following approaches:

- **Before publishing and deploying the project,** add the following *web.config* file to the project root:

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <location path="." inheritInChildApplications="false">
      <system.webServer>
        <security>
          <authentication>
            <anonymousAuthentication enabled="false" />
            <windowsAuthentication enabled="true" />
          </authentication>
        </security>
      </system.webServer>
    </location>
  </configuration>
  ```

  When the project is published by the .NET Core SDK (without the `<IsTransformWebConfigDisabled>` property set to `true` in the project file), the published *web.config* file includes the `<location><system.webServer><security><authentication>` section. For more information on the `<IsTransformWebConfigDisabled>` property, see Host ASP.NET Core on Windows with IIS.

- **After publishing and deploying the project,** perform server-side configuration with the IIS Manager:

  1. In IIS Manager, select the IIS site under the **Sites** node of the **Connections** sidebar.
  2. Double-click **Authentication** in the **IIS** area.
  3. Select **Anonymous Authentication**. Select **Disable** in the **Actions** sidebar.
  4. Select **Windows Authentication**. Select **Enable** in the **Actions** sidebar.

  When these actions are taken, IIS Manager modifies the app's *web.config* file. A `<system.webServer><security><authentication>` node is added with updated settings for `anonymousAuthentication` and `windowsAuthentication`:

  ```
  <system.webServer>
    <security>
      <authentication>
        <anonymousAuthentication enabled="false" />
        <windowsAuthentication enabled="true" />
      </authentication>
    </security>
  </system.webServer>
  ```

  The `<system.webServer>` section added to the *web.config* file by IIS Manager is outside of the app's `<location>` section added by the .NET Core SDK when the app is published. Because the section is added outside of the `<location>` node, the settings are inherited by any sub-apps to the current app. To prevent inheritance, move the added `<security>` section inside of the `<location><system.webServer>` section that the .NET Core SDK provided.

  When IIS Manager is used to add the IIS configuration, it only affects the app's *web.config* file on the server. A subsequent deployment of the app may overwrite the settings on the server if the server's copy of *web.config* is replaced by the project's *web.config* file. Use **either** of the following

approaches to manage the settings:

- Use IIS Manager to reset the settings in the *web.config* file after the file is overwritten on deployment.
- Add a *web.config file* to the app locally with the settings.

Kestrel

The Microsoft.AspNetCore.Authentication.Negotiate NuGet package can be used with Kestrel to support Windows Authentication using Negotiate and Kerberos on Windows, Linux, and macOS.

> **WARNING**
>
> Credentials can be persisted across requests on a connection. *Negotiate authentication must not be used with proxies unless the proxy maintains a 1:1 connection affinity (a persistent connection) with Kestrel.*

> **NOTE**
>
> The Negotiate handler detects if the underlying server supports Windows Authentication natively and if it's enabled. If the server supports Windows Authentication but it's disabled, an error is thrown asking you to enable the server implementation. When Windows Authentication is enabled in the server, the Negotiate handler transparently forwards to it.

Add authentication services by invoking AddAuthentication and AddNegotiate in `Startup.ConfigureServices`:

```
// using Microsoft.AspNetCore.Authentication.Negotiate;
// using Microsoft.Extensions.DependencyInjection;

services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
   .AddNegotiate();
```

Add Authentication Middleware by calling UseAuthentication in `Startup.Configure`:

```
app.UseAuthentication();
```

For more information on middleware, see ASP.NET Core Middleware.

Anonymous requests are allowed. Use ASP.NET Core Authorization to challenge anonymous requests for authentication.

Windows environment configuration

The Microsoft.AspNetCore.Authentication.Negotiate component performs User Mode authentication. Service Principal Names (SPNs) must be added to the user account running the service, not the machine account. Execute `setspn -S HTTP/myservername.mydomain.com myuser` in an administrative command shell.

Linux and macOS environment configuration

Instructions for joining a Linux or macOS machine to a Windows domain are available in the Connect Azure Data Studio to your SQL Server using Windows authentication - Kerberos article. The instructions create a machine account for the Linux machine on the domain. SPNs must be added to that machine account.

> **NOTE**
>
> When following the guidance in the Connect Azure Data Studio to your SQL Server using Windows authentication - Kerberos article, replace `python-software-properties` with `python3-software-properties` if needed.

Once the Linux or macOS machine is joined to the domain, additional steps are required to provide a keytab file with the SPNs:

- On the domain controller, add new web service SPNs to the machine account:
  - `setspn -S HTTP/mywebservice.mydomain.com mymachine`
  - `setspn -S HTTP/mywebservice@MYDOMAIN.COM mymachine`
- Use ktpass to generate a keytab file:
  - ```
    ktpass -princ HTTP/mywebservice.mydomain.com@MYDOMAIN.COM -pass myKeyTabFilePassword -mapuser
    MYDOMAIN\mymachine$ -pType KRB5_NT_PRINCIPAL -out c:\temp\mymachine.HTTP.keytab -crypto
    AES256-SHA1
    ```
  - Some fields must be specified in uppercase as indicated.
- Copy the keytab file to the Linux or macOS machine.
- Select the keytab file via an environment variable: `export KRB5_KTNAME=/tmp/mymachine.HTTP.keytab`
- Invoke `klist` to show the SPNs currently available for use.

> **NOTE**
>
> A keytab file contains domain access credentials and must be protected accordingly.

HTTP.sys

HTTP.sys supports Kernel Mode Windows Authentication using Negotiate, NTLM, or Basic authentication.

Add authentication services by invoking AddAuthentication (Microsoft.AspNetCore.Server.HttpSys namespace) in `Startup.ConfigureServices`:

```
services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
```

Configure the app's web host to use HTTP.sys with Windows Authentication (*Program.cs*). UseHttpSys is in the Microsoft.AspNetCore.Server.HttpSys namespace.
```

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>()
                    .UseHttpSys(options =>
                    {
                        options.Authentication.Schemes =
                            AuthenticationSchemes.NTLM |
                            AuthenticationSchemes.Negotiate;
                        options.Authentication.AllowAnonymous = false;
                    });
            });
}
```

```
public class Program
{
    public static void Main(string[] args) =>
        BuildWebHost(args).Run();

    public static IWebHost BuildWebHost(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .UseHttpSys(options =>
            {
                options.Authentication.Schemes =
                    AuthenticationSchemes.NTLM |
                    AuthenticationSchemes.Negotiate;
                options.Authentication.AllowAnonymous = false;
            })
            .Build();
}
```

NOTE

HTTP.sys delegates to Kernel Mode authentication with the Kerberos authentication protocol. User Mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

NOTE

HTTP.sys isn't supported on Nano Server version 1709 or later. To use Windows Authentication and HTTP.sys with Nano Server, use a Server Core (microsoft/windowsservercore) container. For more information on Server Core, see What is the Server Core installation option in Windows Server?.

Authorize users

The configuration state of anonymous access determines the way in which the `[Authorize]` and `[AllowAnonymous]` attributes are used in the app. The following two sections explain how to handle the disallowed and allowed configuration states of anonymous access.

Disallow anonymous access

When Windows Authentication is enabled and anonymous access is disabled, the `[Authorize]` and `[AllowAnonymous]` attributes have no effect. If an IIS site is configured to disallow anonymous access, the request never reaches the app. For this reason, the `[AllowAnonymous]` attribute isn't applicable.

Allow anonymous access

When both Windows Authentication and anonymous access are enabled, use the `[Authorize]` and `[AllowAnonymous]` attributes. The `[Authorize]` attribute allows you to secure endpoints of the app which require authentication. The `[AllowAnonymous]` attribute overrides the `[Authorize]` attribute in apps that allow anonymous access. For attribute usage details, see Simple authorization in ASP.NET Core.

> **NOTE**
>
> By default, users who lack authorization to access a page are presented with an empty HTTP 403 response. The StatusCodePages Middleware can be configured to provide users with a better "Access Denied" experience.

Impersonation

ASP.NET Core doesn't implement impersonation. Apps run with the app's identity for all requests, using app pool or process identity. If the app should perform an action on behalf of a user, use WindowsIdentity.RunImpersonated in a terminal inline middleware in `Startup.Configure`. Run a single action in this context and then close the context.

```
app.Run(async (context) =>
{
    try
    {
        var user = (WindowsIdentity)context.User.Identity;

        await context.Response
            .WriteAsync($"User: {user.Name}\tState: {user.ImpersonationLevel}\n");

        WindowsIdentity.RunImpersonated(user.AccessToken, () =>
        {
            var impersonatedUser = WindowsIdentity.GetCurrent();
            var message =
                $"User: {impersonatedUser.Name}\t" +
                $"State: {impersonatedUser.ImpersonationLevel}";

            var bytes = Encoding.UTF8.GetBytes(message);
            context.Response.Body.Write(bytes, 0, bytes.Length);
        });
    }
    catch (Exception e)
    {
        await context.Response.WriteAsync(e.ToString());
    }
});
```

`RunImpersonated` doesn't support asynchronous operations and shouldn't be used for complex scenarios. For example, wrapping entire requests or middleware chains isn't supported or recommended.

While the Microsoft.AspNetCore.Authentication.Negotiate package enables authentication on Windows, Linux, and macOS, impersonation is only supported on Windows.

Claims transformations

When hosting with IIS, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. For more information and a code example that activates claims transformations, see ASP.NET Core Module.

When hosting with IIS in-process mode, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. For more information and a code example that activates claims transformations when hosting in-process, see ASP.NET Core Module.

Additional resources

- dotnet publish
- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

Custom storage providers for ASP.NET Core Identity

9/22/2020 • 9 minutes to read • Edit Online

By Steve Smith

ASP.NET Core Identity is an extensible system which enables you to create a custom storage provider and connect it to your app. This topic describes how to create a customized storage provider for ASP.NET Core Identity. It covers the important concepts for creating your own storage provider, but isn't a step-by-step walkthrough.

View or download sample from GitHub.

Introduction

By default, the ASP.NET Core Identity system stores user information in a SQL Server database using Entity Framework Core. For many apps, this approach works well. However, you may prefer to use a different persistence mechanism or data schema. For example:

- You use Azure Table Storage or another data store.
- Your database tables have a different structure.
- You may wish to use a different data access approach, such as Dapper.

In each of these cases, you can write a customized provider for your storage mechanism and plug that provider into your app.

ASP.NET Core Identity is included in project templates in Visual Studio with the "Individual User Accounts" option.

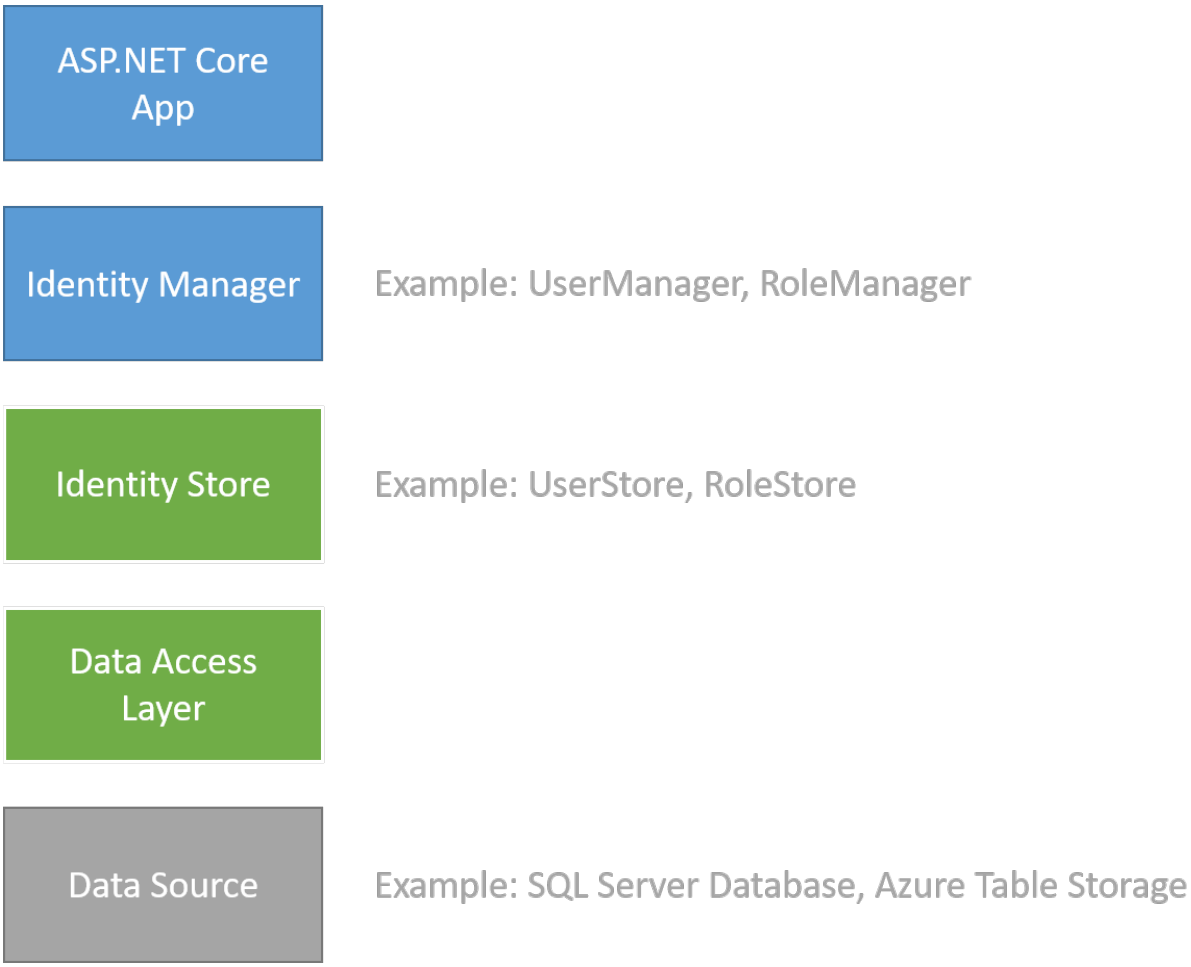
When using the .NET Core CLI, add `-au Individual`:

```
dotnet new mvc -au Individual
```

The ASP.NET Core Identity architecture

ASP.NET Core Identity consists of classes called managers and stores. *Managers* are high-level classes which an app developer uses to perform operations, such as creating an Identity user. *Stores* are lower-level classes that specify how entities, such as users and roles, are persisted. Stores follow the repository pattern and are closely coupled with the persistence mechanism. Managers are decoupled from stores, which means you can replace the persistence mechanism without changing your application code (except for configuration).

The following diagram shows how a web app interacts with the managers, while stores interact with the data access layer.



To create a custom storage provider, create the data source, the data access layer, and the store classes that interact with this data access layer (the green and grey boxes in the diagram above). You don't need to customize the managers or your app code that interacts with them (the blue boxes above).

When creating a new instance of `UserManager` or `RoleManager` you provide the type of the user class and pass an instance of the store class as an argument. This approach enables you to plug your customized classes into ASP.NET Core.

Reconfigure app to use new storage provider shows how to instantiate `UserManager` and `RoleManager` with a customized store.

ASP.NET Core Identity stores data types

ASP.NET Core Identity data types are detailed in the following sections:

Users

Registered users of your web site. The IdentityUser type may be extended or used as an example for your own custom type. You don't need to inherit from a particular type to implement your own custom identity storage solution.

User Claims

A set of statements (or Claims) about the user that represent the user's identity. Can enable greater expression of the user's identity than can be achieved through roles.

User Logins

Information about the external authentication provider (like Facebook or a Microsoft account) to use when logging in a user. Example

Roles

Authorization groups for your site. Includes the role Id and role name (like "Admin" or "Employee"). Example

The data access layer

This topic assumes you are familiar with the persistence mechanism that you are going to use and how to create entities for that mechanism. This topic doesn't provide details about how to create the repositories or data access classes; it provides some suggestions about design decisions when working with ASP.NET Core Identity.

You have a lot of freedom when designing the data access layer for a customized store provider. You only need to create persistence mechanisms for features that you intend to use in your app. For example, if you are not using roles in your app, you don't need to create storage for roles or user role associations. Your technology and existing infrastructure may require a structure that's very different from the default implementation of ASP.NET Core Identity. In your data access layer, you provide the logic to work with the structure of your storage implementation.

The data access layer provides the logic to save the data from ASP.NET Core Identity to a data source. The data access layer for your customized storage provider might include the following classes to store user and role information.

Context class

Encapsulates the information to connect to your persistence mechanism and execute queries. Several data classes require an instance of this class, typically provided through dependency injection. Example.

User Storage

Stores and retrieves user information (such as user name and password hash). Example

Role Storage

Stores and retrieves role information (such as the role name). Example

UserClaims Storage

Stores and retrieves user claim information (such as the claim type and value). Example

UserLogins Storage

Stores and retrieves user login information (such as an external authentication provider). Example

UserRole Storage

Stores and retrieves which roles are assigned to which users. Example

TIP: Only implement the classes you intend to use in your app.

In the data access classes, provide code to perform data operations for your persistence mechanism. For example, within a custom provider, you might have the following code to create a new user in the *store* class:

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user,
    CancellationToken cancellationToken = default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(user));

    return await _usersTable.CreateAsync(user);
}
```

The implementation logic for creating the user is in the `_usersTable.CreateAsync` method, shown below.

Customize the user class

When implementing a storage provider, create a user class which is equivalent to the IdentityUser class.

At a minimum, your user class must include an `Id` and a `UserName` property.

The `IdentityUser` class defines the properties that the `UserManager` calls when performing requested operations. The default type of the `Id` property is a string, but you can inherit from `IdentityUser<TKey, TUserClaim, TUserRole, TUserLogin, TUserToken>` and specify a different type. The framework expects the storage implementation to handle data type conversions.

Customize the user store

Create a `UserStore` class that provides the methods for all data operations on the user. This class is equivalent to the UserStore<TUser> class. In your `UserStore` class, implement `IUserStore<TUser>` and the optional interfaces required. You select which optional interfaces to implement based on the functionality provided in your app.

Optional interfaces

- IUserRoleStore
- IUserClaimStore
- IUserPasswordStore
- IUserSecurityStampStore
- IUserEmailStore
- IUserPhoneNumberStore
- IQueryableUserStore
- IUserLoginStore
- IUserTwoFactorStore
- IUserLockoutStore

The optional interfaces inherit from `IUserStore<TUser>`. You can see a partially implemented sample user store in the sample app.

Within the `UserStore` class, you use the data access classes that you created to perform operations. These are passed in using dependency injection. For example, in the SQL Server with Dapper implementation, the `UserStore` class has the `CreateAsync` method which uses an instance of `DapperUsersTable` to insert a new record:

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user)
{
    string sql = "INSERT INTO dbo.CustomUser " +
        "VALUES (@id, @Email, @EmailConfirmed, @PasswordHash, @UserName)";

    int rows = await _connection.ExecuteAsync(sql, new { user.Id, user.Email, user.EmailConfirmed,
user.PasswordHash, user.UserName });

    if(rows > 0)
    {
        return IdentityResult.Success;
    }
    return IdentityResult.Failed(new IdentityError { Description = $"Could not insert user {user.Email}." });
}
```

Interfaces to implement when customizing user store

- **IUserStore**
  The IUserStore<TUser> interface is the only interface you must implement in the user store. It defines methods for creating, updating, deleting, and retrieving users.
- **IUserClaimStore**
  The IUserClaimStore<TUser> interface defines the methods you implement to enable user claims. It contains methods for adding, removing and retrieving user claims.

- **IUserLoginStore**
  The IUserLoginStore<TUser> defines the methods you implement to enable external authentication providers. It contains methods for adding, removing and retrieving user logins, and a method for retrieving a user based on the login information.
- **IUserRoleStore**
  The IUserRoleStore<TUser> interface defines the methods you implement to map a user to a role. It contains methods to add, remove, and retrieve a user's roles, and a method to check if a user is assigned to a role.
- **IUserPasswordStore**
  The IUserPasswordStore<TUser> interface defines the methods you implement to persist hashed passwords. It contains methods for getting and setting the hashed password, and a method that indicates whether the user has set a password.
- **IUserSecurityStampStore**
  The IUserSecurityStampStore<TUser> interface defines the methods you implement to use a security stamp for indicating whether the user's account information has changed. This stamp is updated when a user changes the password, or adds or removes logins. It contains methods for getting and setting the security stamp.
- **IUserTwoFactorStore**
  The IUserTwoFactorStore<TUser> interface defines the methods you implement to support two factor authentication. It contains methods for getting and setting whether two factor authentication is enabled for a user.
- **IUserPhoneNumberStore**
  The IUserPhoneNumberStore<TUser> interface defines the methods you implement to store user phone numbers. It contains methods for getting and setting the phone number and whether the phone number is confirmed.
- **IUserEmailStore**
  The IUserEmailStore<TUser> interface defines the methods you implement to store user email addresses. It contains methods for getting and setting the email address and whether the email is confirmed.
- **IUserLockoutStore**
  The IUserLockoutStore<TUser> interface defines the methods you implement to store information about locking an account. It contains methods for tracking failed access attempts and lockouts.
- **IQueryableUserStore**
  The IQueryableUserStore<TUser> interface defines the members you implement to provide a queryable user store.

You implement only the interfaces that are needed in your app. For example:

```
public class UserStore : IUserStore<IdentityUser>,
                         IUserClaimStore<IdentityUser>,
                         IUserLoginStore<IdentityUser>,
                         IUserRoleStore<IdentityUser>,
                         IUserPasswordStore<IdentityUser>,
                         IUserSecurityStampStore<IdentityUser>
{
    // interface implementations not shown
}
```

IdentityUserClaim, IdentityUserLogin, and IdentityUserRole

The `Microsoft.AspNet.Identity.EntityFramework` namespace contains implementations of the IdentityUserClaim, IdentityUserLogin, and IdentityUserRole classes. If you are using these features, you may want to create your own versions of these classes and define the properties for your app. However, sometimes it's more efficient to not load these entities into memory when performing basic operations (such as adding or removing a user's claim). Instead, the backend store classes can execute these operations directly on the data source. For example, the `UserStore.GetClaimsAsync` method can call the `userClaimTable.FindByUserId(user.Id)` method to execute a query

on that table directly and return a list of claims.

Customize the role class

When implementing a role storage provider, you can create a custom role type. It need not implement a particular interface, but it must have an `Id` and typically it will have a `Name` property.

The following is an example role class:

```
using System;

namespace CustomIdentityProviderSample.CustomProvider
{
    public class ApplicationRole
    {
        public Guid Id { get; set; } = Guid.NewGuid();
        public string Name { get; set; }
    }
}
```

Customize the role store

You can create a `RoleStore` class that provides the methods for all data operations on roles. This class is equivalent to the RoleStore<TRole> class. In the `RoleStore` class, you implement the `IRoleStore<TRole>` and optionally the `IQueryableRoleStore<TRole>` interface.

- **IRoleStore<TRole>**
  The IRoleStore<TRole> interface defines the methods to implement in the role store class. It contains methods for creating, updating, deleting, and retrieving roles.
- **RoleStore<TRole>**
  To customize `RoleStore`, create a class that implements the `IRoleStore<TRole>` interface.

Reconfigure app to use a new storage provider

Once you have implemented a storage provider, you configure your app to use it. If your app used the default provider, replace it with your custom provider.

1. Remove the `Microsoft.AspNetCore.EntityFramework.Identity` NuGet package.
2. If the storage provider resides in a separate project or package, add a reference to it.
3. Replace all references to `Microsoft.AspNetCore.EntityFramework.Identity` with a using statement for the namespace of your storage provider.
4. In the `ConfigureServices` method, change the `AddIdentity` method to use your custom types. You can create your own extension methods for this purpose. See IdentityServiceCollectionExtensions for an example.
5. If you are using Roles, update the `RoleManager` to use your `RoleStore` class.
6. Update the connection string and credentials to your app's configuration.

Example:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add identity types
    services.AddIdentity<ApplicationUser, ApplicationRole>()
        .AddDefaultTokenProviders();

    // Identity Services
    services.AddTransient<IUserStore<ApplicationUser>, CustomUserStore>();
    services.AddTransient<IRoleStore<ApplicationRole>, CustomRoleStore>();
    string connectionString = Configuration.GetConnectionString("DefaultConnection");
    services.AddTransient<SqlConnection>(e => new SqlConnection(connectionString));
    services.AddTransient<DapperUsersTable>();

    // additional configuration
}
```

References

- Custom Storage Providers for ASP.NET 4.x Identity
- ASP.NET Core Identity: This repository includes links to community maintained store providers.

Facebook, Google, and external provider authentication in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By Valeriy Novytskyy and Rick Anderson

This tutorial demonstrates how to build an ASP.NET Core 3.0 app that enables users to sign in using OAuth 2.0 with credentials from external authentication providers.

Facebook, Twitter, Google, and Microsoft providers are covered in the following sections and use the starter project created in this article. Other providers are available in third-party packages such as AspNet.Security.OAuth.Providers and AspNet.Security.OpenId.Providers.

Enabling users to sign in with their existing credentials:

- Is convenient for the users.
- Shifts many of the complexities of managing the sign-in process onto a third party.

For examples of how social logins can drive traffic and customer conversions, see case studies by Facebook and Twitter.

Create a New ASP.NET Core Project

- Visual Studio
- Visual Studio Code / Visual Studio for Mac

- Create a new project.
- Select **ASP.NET Core Web Application** and **Next**.
- Provide a **Project name** and confirm or change the **Location**. Select **Create**.
- Select the latest version of ASP.NET Core in the drop-down (**ASP.NET Core {X.Y}**), and then select **Web Application**.
- Under **Authentication**, select **Change** and set the authentication to **Individual User Accounts**. Select **OK**.
- In the **Create a new ASP.NET Core Web Application** window, select **Create**.

Apply migrations

- Run the app and select the **Register** link.
- Enter the email and password for the new account, and then select **Register**.
- Follow the instructions to apply migrations.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Use SecretManager to store tokens assigned by login providers

Social login providers assign **Application Id** and **Application Secret** tokens during the registration process. The exact token names vary by provider. These tokens represent the credentials your app uses to access their API. The tokens constitute the "secrets" that can be linked to your app configuration with the help of Secret Manager. Secret Manager is a more secure alternative to storing the tokens in a configuration file, such as *appsettings.json*.

> **IMPORTANT**
> Secret Manager is for development purposes only. You can store and protect Azure test and production secrets with the Azure Key Vault configuration provider.

Follow the steps in Safe storage of app secrets in development in ASP.NET Core topic to store tokens assigned by each login provider below.

Setup login providers required by your application

Use the following topics to configure your application to use the respective providers:

- Facebook instructions
- Twitter instructions
- Google instructions
- Microsoft instructions
- Other provider instructions

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

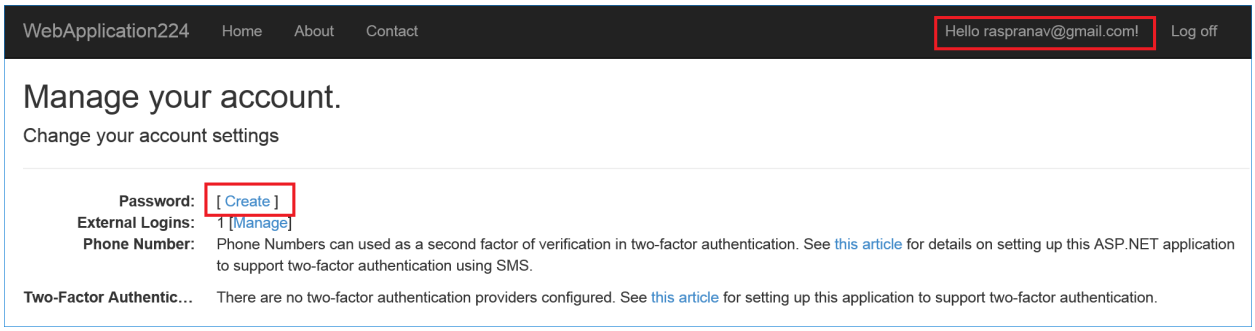
```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

Optionally set password

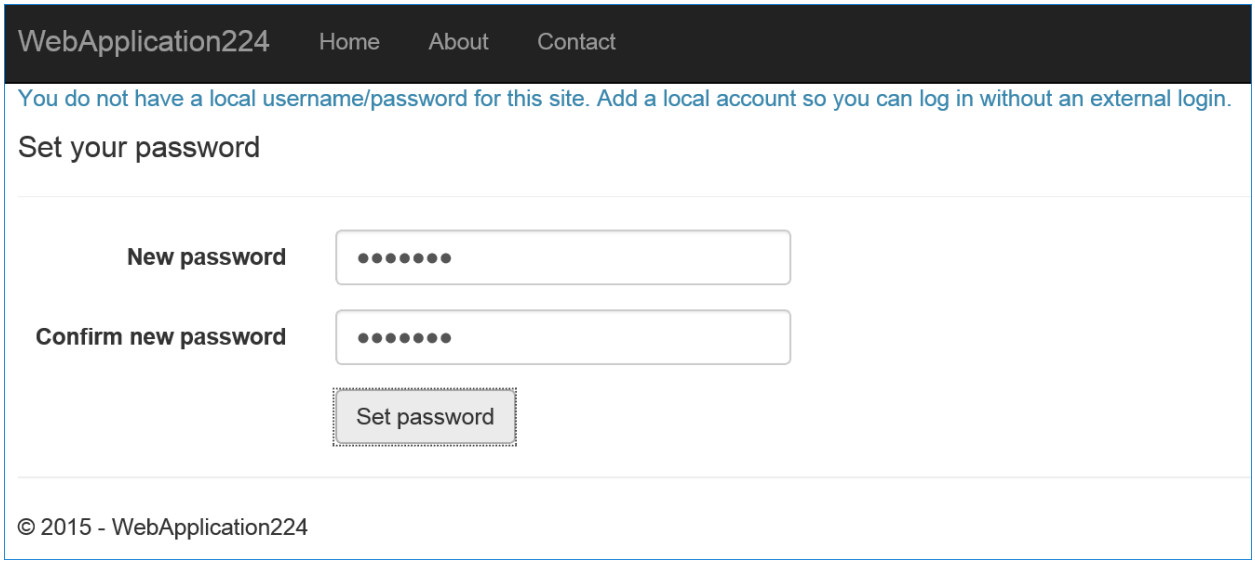
When you register with an external login provider, you don't have a password registered with the app. This alleviates you from creating and remembering a password for the site, but it also makes you dependent on the external login provider. If the external login provider is unavailable, you won't be able to sign in to the web site.

To create a password and sign in using your email that you set during the sign in process with external providers:

- Select the **Hello <email alias>** link at the top-right corner to navigate to the **Manage** view.



- Select **Create**



- Set a valid password and you can use this to sign in with your email.

Next steps

- See this GitHub issue for information on how to customize the login buttons.
- This article introduced external authentication and explained the prerequisites required to add external logins to your ASP.NET Core app.
- Reference provider-specific pages to configure logins for the providers required by your app.
- You may want to persist additional data about the user and their access and refresh tokens. For more information, see Persist additional claims and tokens from external providers in ASP.NET Core.

Google external login setup in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Valeriy Novytskyy and Rick Anderson

This tutorial shows you how to enable users to sign in with their Google account using the ASP.NET Core 3.0 project created on the previous page.

Create a Google API Console project and client ID

- Install Microsoft.AspNetCore.Authentication.Google.
- Navigate to Integrating Google Sign-In into your web app and select **Configure a project**.
- In the **Configure your OAuth client** dialog, select **Web server**.
- In the **Authorized redirect URIs** text entry box, set the redirect URI. For example, `https://localhost:44312/signin-google`
- Save the **Client ID** and **Client Secret**.
- When deploying the site, register the new public url from the **Google Console**.

Store the Google client ID and secret

Store sensitive settings such as the Google client ID and secret values with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret`:

```
dotnet user-secrets set "Authentication:Google:ClientId" "<client-id>"
dotnet user-secrets set "Authentication:Google:ClientSecret" "<client-secret>"
```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

You can manage your API credentials and usage in the API Console.

Configure Google authentication

Add the Google service to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>
        options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();

    services.AddAuthentication()
        .AddGoogle(options =>
        {
            IConfigurationSection googleAuthNSection =
                Configuration.GetSection("Authentication:Google");

            options.ClientId = googleAuthNSection["ClientId"];
            options.ClientSecret = googleAuthNSection["ClientSecret"];
        });
}
```

The call to AddIdentity configures the default scheme settings. The AddAuthentication(String) overload sets the DefaultScheme property. The AddAuthentication(Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Sign in with Google

- Run the app and click **Log in**. An option to sign in with Google appears.
- Click the **Google** button, which redirects to Google for authentication.
- After entering your Google credentials, you are redirected back to the web site.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

See the GoogleOptions API reference for more information on configuration options supported by Google authentication. This can be used to request different information about the user.

Change the default callback URI

The URI segment `/signin-google` is set as the default callback of the Google authentication provider. You can change the default callback URI while configuring the Google authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the GoogleOptions class.

Troubleshooting

- If the sign-in doesn't work and you aren't getting any errors, switch to development mode to make the issue easier to debug.
- If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate results in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you get *A database operation failed while processing the request* error. Select **Apply Migrations** to create the database, and refresh the page to continue past the error.

Next steps

- This article showed how you can authenticate with Google. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish the app to Azure, reset the `ClientSecret` in the Google API Console.
- Set the `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Facebook external login setup in ASP.NET Core

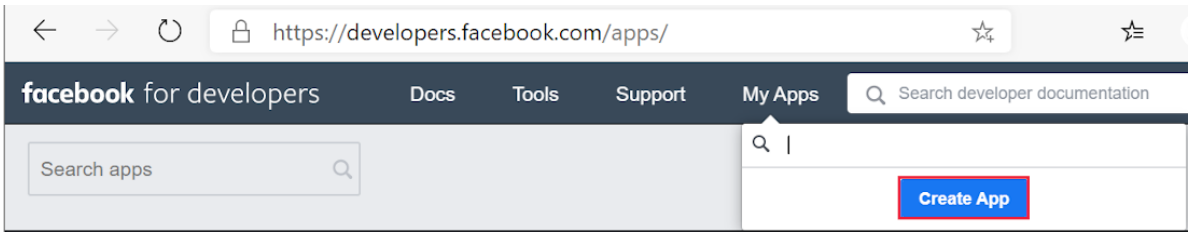
9/22/2020 • 5 minutes to read • Edit Online

By Valeriy Novytskyy and Rick Anderson

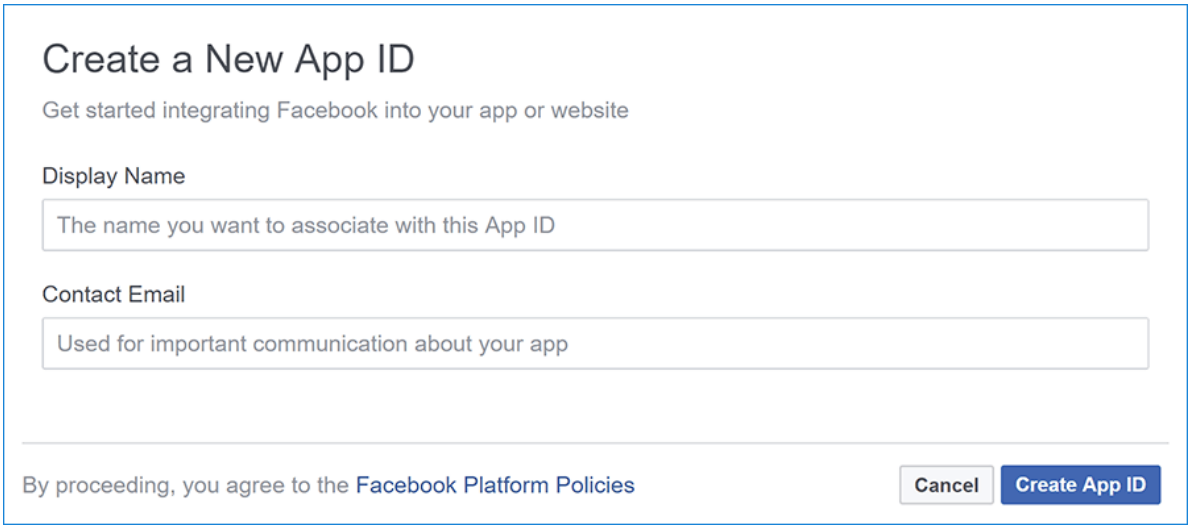
This tutorial with code examples shows how to enable your users to sign in with their Facebook account using a sample ASP.NET Core 3.0 project created on the previous page. We start by creating a Facebook App ID by following the official steps.

Create the app in Facebook

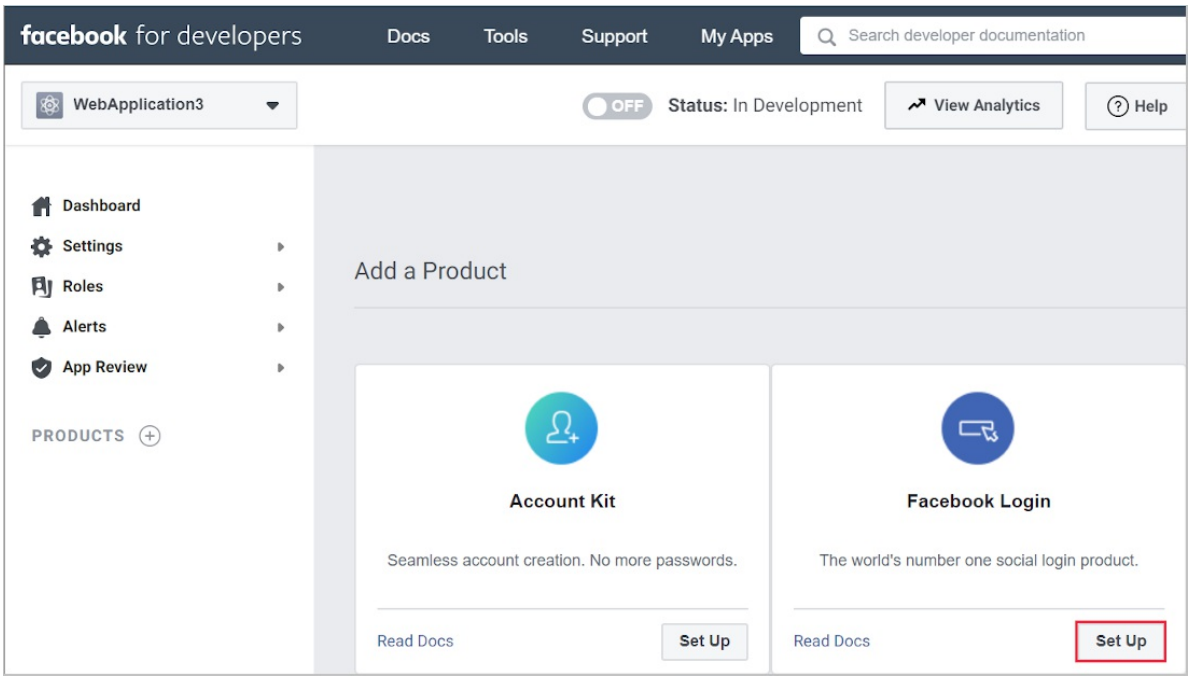
- Add the Microsoft.AspNetCore.Authentication.Facebook NuGet package to the project.
- Navigate to the Facebook Developers app page and sign in. If you don't already have a Facebook account, use the **Sign up for Facebook** link on the login page to create one. Once you have a Facebook account, follow the instructions to register as a Facebook Developer.
- From the **My Apps** menu select **Create App** to create a new App ID.



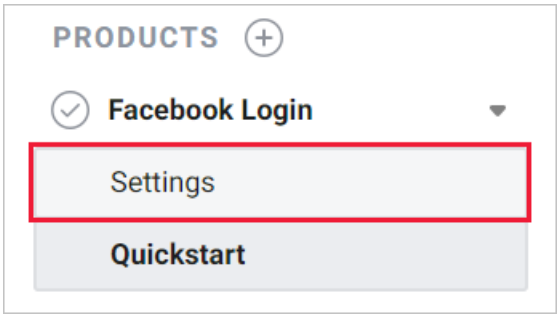
- Fill out the form and tap the **Create App ID** button.



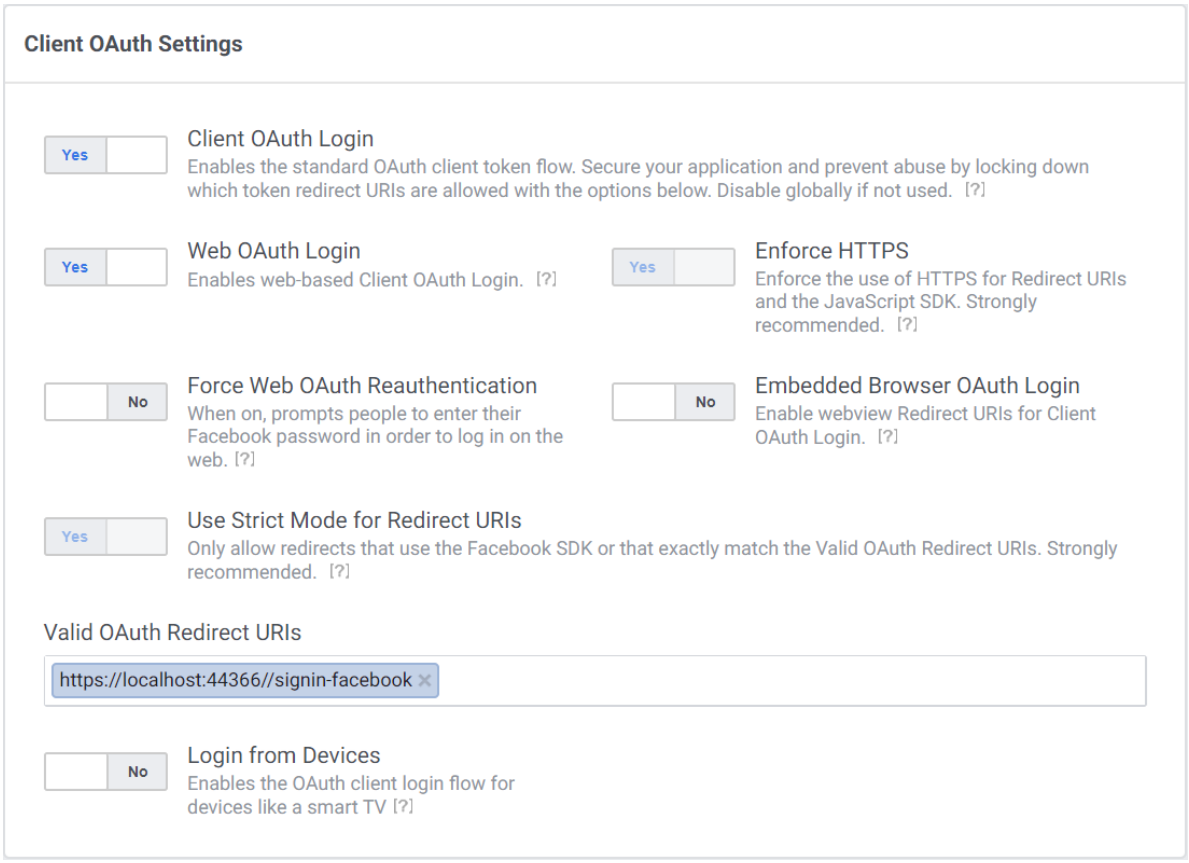
- On the new App card, select **Add a Product**. On the **Facebook Login** card, click **Set Up**



- The **Quickstart** wizard launches with **Choose a Platform** as the first page. Bypass the wizard for now by clicking the **FaceBook Login Settings** link in the menu on the lower left:



- You are presented with the **Client OAuth Settings** page:



- Enter your development URI with */signin-facebook* appended into the **Valid OAuth Redirect URIs** field (for example: `https://localhost:44320/signin-facebook`). The Facebook authentication configured later in this tutorial will automatically handle requests at */signin-facebook* route to implement the OAuth flow.

> **NOTE**
> The URI */signin-facebook* is set as the default callback of the Facebook authentication provider. You can change the default callback URI while configuring the Facebook authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the FacebookOptions class.

- Click **Save Changes**.
- Click **Settings** > **Basic** link in the left navigation.

  On this page, make a note of your `App ID` and your `App Secret`. You will add both into your ASP.NET Core application in the next section:
- When deploying the site you need to revisit the **Facebook Login** setup page and register a new public URI.

Store the Facebook app ID and secret

Store sensitive settings such as the Facebook app ID and secret values with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret`:

```
dotnet user-secrets set "Authentication:Facebook:AppId" "<app-id>"
dotnet user-secrets set "Authentication:Facebook:AppSecret" "<app-secret>"
```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

Configure Facebook Authentication

Add the Facebook service in the `ConfigureServices` method in the *Startup.cs* file:

```
services.AddAuthentication().AddFacebook(facebookOptions =>
{
    facebookOptions.AppId = Configuration["Authentication:Facebook:AppId"];
    facebookOptions.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
});
```

The AddAuthentication(String) overload sets the DefaultScheme property. The AddAuthentication(Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per

authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Sign in with Facebook

- Run the app and select **Log in**.
- Under **Use another service to log in.**, select Facebook.
- You are redirected to **Facebook** for authentication.
- Enter your Facebook credentials.
- You are redirected back to your site where you can set your email.

You are now logged in using your Facebook credentials:

React to cancel authorize external sign-in

AccessDeniedPath can provide a redirect path to the user agent when the user doesn't approve the requested authorization demand.

The following code sets the `AccessDeniedPath` to `"/AccessDeniedPathInfo"`:

```
services.AddAuthentication().AddFacebook(options =>
{
    options.AppId = Configuration["Authentication:Facebook:AppId"];
    options.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
    options.AccessDeniedPath = "/AccessDeniedPathInfo";
});
```

We recommend the `AccessDeniedPath` page contain the following information:

- Remote authentication was canceled.
- This app requires authentication.
- To try sign-in again, select the Login link.

Test AccessDeniedPath

- Navigate to facebook.com
- If you are signed in, you must sign out.
- Run the app and select Facebook sign-in.
- Select **Not now**. You are redirected to the specified `AccessDeniedPath` page.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

See the FacebookOptions API reference for more information on configuration options supported by Facebook authentication. Configuration options can be used to:

- Request different information about the user.
- Add query string arguments to customize the login experience.

Troubleshooting

- **ASP.NET Core 2.x only:** If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Facebook. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish your web site to Azure web app, you should reset the `AppSecret` in the Facebook developer portal.
- Set the `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Microsoft Account external login setup with ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By Valeriy Novytskyy and Rick Anderson

This sample shows you how to enable users to sign in with their work, school, or personal Microsoft account using the ASP.NET Core 3.0 project created on the previous page.

Create the app in Microsoft Developer Portal

- Add the Microsoft.AspNetCore.Authentication.MicrosoftAccount NuGet package to the project.
- Navigate to the Azure portal - App registrations page and create or sign into a Microsoft account:

If you don't have a Microsoft account, select **Create one**. After signing in, you are redirected to the **App registrations** page:

- Select **New registration**
- Enter a **Name**.
- Select an option for **Supported account types**.
  - The `MicrosoftAccount` package supports App Registrations created using "Accounts in any organizational directory" or "Accounts in any organizational directory and Microsoft accounts" options by default.
  - To use other options, set `AuthorizationEndpoint` and `TokenEndpoint` members of `MicrosoftAccountOptions` used to initialize the Microsoft Account authentication to the URLs displayed on **Endpoints** page of the App Registration after it is created (available by clicking Endpoints on the **Overview** page).
- Under **Redirect URI**, enter your development URL with `/signin-microsoft` appended. For example, `https://localhost:5001/signin-microsoft`. The Microsoft authentication scheme configured later in this sample will automatically handle requests at `/signin-microsoft` route to implement the OAuth flow.
- Select **Register**

Create client secret

- In the left pane, select **Certificates & secrets**.
- Under **Client secrets**, select **New client secret**
  - Add a description for the client secret.
  - Select the **Add** button.
- Under **Client secrets**, copy the value of the client secret.

The URI segment `/signin-microsoft` is set as the default callback of the Microsoft authentication provider. You can change the default callback URI while configuring the Microsoft authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the MicrosoftAccountOptions class.

Store the Microsoft client ID and secret

Store sensitive settings such as the Microsoft client ID and secret values with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Microsoft:ClientId` and `Authentication:Microsoft:ClientSecret`:

```
dotnet user-secrets set "Authentication:Microsoft:ClientId" "<client-id>"
dotnet user-secrets set "Authentication:Microsoft:ClientSecret" "<client-secret>"
```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

Configure Microsoft Account Authentication

Add the Microsoft Account service to the `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();

    services.AddAuthentication().AddMicrosoftAccount(microsoftOptions =>
    {
        microsoftOptions.ClientId = Configuration["Authentication:Microsoft:ClientId"];
        microsoftOptions.ClientSecret = Configuration["Authentication:Microsoft:ClientSecret"];
    });
}
```

The AddAuthentication(String) overload sets the DefaultScheme property. The AddAuthentication(Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

For more information about configuration options supported by Microsoft Account authentication, see the MicrosoftAccountOptions API reference. This can be used to request different information about the user.

Sign in with Microsoft Account

Run the app and click **Log in**. An option to sign in with Microsoft appears. When you click on Microsoft, you are redirected to Microsoft for authentication. After signing in with your Microsoft Account, you will be prompted to let the app access your info:

Tap **Yes** and you will be redirected back to the web site where you can set your email.

You are now logged in using your Microsoft credentials:

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Troubleshooting

- If the Microsoft Account provider redirects you to a sign in error page, note the error title and description query string parameters directly following the `#` (hashtag) in the Uri.

  Although the error message seems to indicate a problem with Microsoft authentication, the most common cause is your application Uri not matching any of the **Redirect URIs** specified for the **Web** platform.

- If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this sample ensures that this is done.

- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Microsoft. You can follow a similar approach to authenticate with other providers listed on the previous page.

- Once you publish your web site to Azure web app, create a new client secrets in the Microsoft Developer Portal.

- Set the `Authentication:Microsoft:ClientId` and `Authentication:Microsoft:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Twitter external sign-in setup with ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By Valeriy Novytskyy and Rick Anderson

This sample shows how to enable users to sign in with their Twitter account using a sample ASP.NET Core 3.0 project created on the previous page.

Create the app in Twitter

- Add the Microsoft.AspNetCore.Authentication.Twitter NuGet package to the project.
- Navigate to https://apps.twitter.com/ and sign in. If you don't already have a Twitter account, use the **Sign up now** link to create one.
- Select **Create an app**. Fill out the **App name**, **Application description** and public **Website** URI (this can be temporary until you register the domain name):
- Check the box next to **Enable Sign in with Twitter**
- Microsoft.AspNetCore.Identity requires users to have an email address by default. Go to the **Permissions** tab, click the **Edit** button and check the box next to **Request email address from users**.
- Enter your development URI with `/signin-twitter` appended into the **Callback URLs** field (for example: `https://webapp128.azurewebsites.net/signin-twitter`). The Twitter authentication scheme configured later in this sample will automatically handle requests at `/signin-twitter` route to implement the OAuth flow.

  > **NOTE**
  > The URI segment `/signin-twitter` is set as the default callback of the Twitter authentication provider. You can change the default callback URI while configuring the Twitter authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the TwitterOptions class.

- Fill out the rest of the form and select **Create**. New application details are displayed:

Store the Twitter consumer API key and secret

Store sensitive settings such as the Twitter consumer API key and secret with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secrets keys `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret`:

```
dotnet user-secrets set "Authentication:Twitter:ConsumerAPIKey" "<consumer-api-key>"
dotnet user-secrets set "Authentication:Twitter:ConsumerSecret" "<consumer-secret>"
```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.

- Automatically replaced by a `:`

These tokens can be found on the **Keys and Access Tokens** tab after creating a new Twitter application:

Configure Twitter Authentication

Add the Twitter service in the `ConfigureServices` method in *Startup.cs* file:

```
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>
        options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();

    services.AddAuthentication().AddTwitter(twitterOptions =>
    {
        twitterOptions.ConsumerKey = Configuration["Authentication:Twitter:ConsumerAPIKey"];
        twitterOptions.ConsumerSecret = Configuration["Authentication:Twitter:ConsumerSecret"];
        twitterOptions.RetrieveUserDetails = true;
    });

}
```

The AddAuthentication(String) overload sets the DefaultScheme property. The AddAuthentication(Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

See the TwitterOptions API reference for more information on configuration options supported by Twitter authentication. This can be used to request different information about the user.

Sign in with Twitter

Run the app and select **Log in**. An option to sign in with Twitter appears:

Clicking on **Twitter** redirects to Twitter for authentication:

After entering your Twitter credentials, you are redirected back to the web site where you can set your email.

You are now logged in using your Twitter credentials:

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Troubleshooting

- **ASP.NET Core 2.x only:** If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices` , attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided.* The project template used in this sample ensures that this is done.
- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Twitter. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish your web site to Azure web app, you should reset the `ConsumerSecret` in the Twitter developer portal.
- Set the `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

External OAuth authentication providers

9/22/2020 • 2 minutes to read • Edit Online

By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

The following list includes common external OAuth authentication providers that work with ASP.NET Core apps. Third-party NuGet packages, such as the ones maintained by aspnet-contrib, can be used to complement the authentication providers implemented by the ASP.NET Core team.

- LinkedIn (Instructions)
- Instagram (Instructions)
- Reddit (Instructions)
- Github (Instructions)
- Yahoo (Instructions)
- Tumblr (Instructions)
- Pinterest (Instructions)
- Pocket (Instructions)
- Flickr (Instructions)
- Dribble (Instructions)
- Vimeo (Instructions)
- SoundCloud (Instructions)
- VK (Instructions)

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request

processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Persist additional claims and tokens from external providers in ASP.NET Core

9/22/2020 • 16 minutes to read • Edit Online

An ASP.NET Core app can establish additional claims and tokens from external authentication providers, such as Facebook, Google, Microsoft, and Twitter. Each provider reveals different information about users on its platform, but the pattern for receiving and transforming user data into additional claims is the same.

View or download sample code (how to download)

Prerequisites

Decide which external authentication providers to support in the app. For each provider, register the app and obtain a client ID and client secret. For more information, see Facebook, Google, and external provider authentication in ASP.NET Core. The sample app uses the Google authentication provider.

Set the client ID and client secret

The OAuth authentication provider establishes a trust relationship with an app using a client ID and client secret. Client ID and client secret values are created for the app by the external authentication provider when the app is registered with the provider. Each external provider that the app uses must be configured independently with the provider's client ID and client secret. For more information, see the external authentication provider topics that apply to your scenario:

- Facebook authentication
- Google authentication
- Microsoft authentication
- Twitter authentication
- Other authentication providers
- OpenIdConnect

The sample app configures the Google authentication provider with a client ID and client secret provided by Google:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

Establish the authentication scope

Specify the list of permissions to retrieve from the provider by specifying the Scope. Authentication scopes for common external providers appear in the following table.

PROVIDER	SCOPE
Facebook	`https://www.facebook.com/dialog/oauth`
Google	`https://www.googleapis.com/auth/userinfo.profile`
Microsoft	`https://login.microsoftonline.com/common/oauth2/v2.0/authorize`
Twitter	`https://api.twitter.com/oauth/authenticate`

In the sample app, Google's `userinfo.profile` scope is automatically added by the framework when AddGoogle is called on the AuthenticationBuilder. If the app requires additional scopes, add them to the options. In the following example, the Google `https://www.googleapis.com/auth/user.birthday.read` scope is added in order to retrieve a user's birthday:

```
options.Scope.Add("https://www.googleapis.com/auth/user.birthday.read");
```

Map user data keys and create claims

In the provider's options, specify a MapJsonKey or MapJsonSubKey for each key/subkey in the external provider's JSON user data for the app identity to read on sign in. For more information on claim types, see ClaimTypes.

The sample app creates locale (`urn:google:locale`) and picture (`urn:google:picture`) claims from the `locale` and `picture` keys in Google user data:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

In `Microsoft.AspNetCore.Identity.UI.Pages.Account.Internal.ExternalLoginModel.OnPostConfirmationAsync`, an IdentityUser (`ApplicationUser`) is signed into the app with SignInAsync. During the sign in process, the UserManager<TUser> can store an `ApplicationUser` claims for user data available from the Principal.

In the sample app, `OnPostConfirmationAsync` (*Account/ExternalLogin.cshtml.cs*) establishes the locale (`urn:google:locale`) and picture (`urn:google:picture`) claims for the signed in `ApplicationUser`, including a claim for GivenName:

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        ErrorMessage =
            "Error loading external login information during confirmation.";

        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = new IdentityUser
        {
            UserName = Input.Email,
            Email = Input.Email
        };
```

```
        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type == ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = true;

                await _signInManager.SignInAsync(user, props);

                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);

                return LocalRedirect(returnUrl);
            }
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    LoginProvider = info.LoginProvider;
    ReturnUrl = returnUrl;
    return Page();
}
```

By default, a user's claims are stored in the authentication cookie. If the authentication cookie is too large, it can cause the app to fail because:

- The browser detects that the cookie header is too long.
- The overall size of the request is too large.

If a large amount of user data is required for processing user requests:

- Limit the number and size of user claims for request processing to only what the app requires.
- Use a custom ITicketStore for the Cookie Authentication Middleware's SessionStore to store identity across requests. Preserve large quantities of identity information on the server while only sending a small session

identifier key to the client.

Save the access token

SaveTokens defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. `SaveTokens` is set to `false` by default to reduce the size of the final authentication cookie.

The sample app sets the value of `SaveTokens` to `true` in GoogleOptions:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

When `OnPostConfirmationAsync` executes, store the access token (ExternalLoginInfo.AuthenticationTokens) from the external provider in the `ApplicationUser`'s `AuthenticationProperties`.

The sample app saves the access token in `OnPostConfirmationAsync` (new user registration) and `OnGetCallbackAsync` (previously registered user) in *Account/ExternalLogin.cshtml.cs*:

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        ErrorMessage =
            "Error loading external login information during confirmation.";

        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = new IdentityUser
        {
```

```
            UserName = Input.Email,
            Email = Input.Email
        };

        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type == ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = true;

                await _signInManager.SignInAsync(user, props);

                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);

                return LocalRedirect(returnUrl);
            }
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    LoginProvider = info.LoginProvider;
    ReturnUrl = returnUrl;
    return Page();
}
```

How to add additional custom tokens

To demonstrate how to add a custom token, which is stored as part of `SaveTokens`, the sample app adds an AuthenticationToken with the current DateTime for an AuthenticationToken.Name of `TicketCreated`:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

Creating and adding claims

The framework provides common actions and extension methods for creating and adding claims to the collection. For more information, see the ClaimActionCollectionMapExtensions and ClaimActionCollectionUniqueExtensions.

Users can define custom actions by deriving from ClaimAction and implementing the abstract Run method.

For more information, see Microsoft.AspNetCore.Authentication.OAuth.Claims.

Removal of claim actions and claims

ClaimActionCollection.Remove(String) removes all claim actions for the given ClaimType from the collection. ClaimActionCollectionMapExtensions.DeleteClaim(ClaimActionCollection, String) deletes a claim of the given ClaimType from the identity. DeleteClaim is primarily used with OpenID Connect (OIDC) to remove protocol-generated claims.

Sample app output

```
User Claims

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
    9b342344f-7aab-43c2-1ac1-ba75912ca999
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
    someone@gmail.com
AspNet.Identity.SecurityStamp
    7D4312MOWRYYBFI1KXRPHGOSTBVWSFDE
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname
    Judy
urn:google:locale
    en
urn:google:picture
    https://lh4.googleusercontent.com/-XXXXXX/XXXXXX/XXXXXX/XXXXXX/photo.jpg

Authentication Properties

.Token.access_token
    yc23.AlvoZqz56...1lxltXV7D-ZWP9
.Token.token_type
    Bearer
.Token.expires_at
    2019-04-11T22:14:51.0000000+00:00
.Token.TicketCreated
    4/11/2019 9:14:52 PM
.TokenNames
    access_token;token_type;expires_at;TicketCreated
.persistent
.issued
    Thu, 11 Apr 2019 20:51:06 GMT
.expires
    Thu, 25 Apr 2019 20:51:06 GMT
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

An ASP.NET Core app can establish additional claims and tokens from external authentication providers, such as Facebook, Google, Microsoft, and Twitter. Each provider reveals different information about users on its platform, but the pattern for receiving and transforming user data into additional claims is the same.

View or download sample code (how to download)

Prerequisites

Decide which external authentication providers to support in the app. For each provider, register the app and obtain a client ID and client secret. For more information, see Facebook, Google, and external provider authentication in ASP.NET Core. The sample app uses the Google authentication provider.

Set the client ID and client secret

The OAuth authentication provider establishes a trust relationship with an app using a client ID and client secret. Client ID and client secret values are created for the app by the external authentication provider when the app is registered with the provider. Each external provider that the app uses must be configured independently with the provider's client ID and client secret. For more information, see the external authentication provider topics that apply to your scenario:

- Facebook authentication
- Google authentication
- Microsoft authentication
- Twitter authentication
- Other authentication providers
- OpenIdConnect

The sample app configures the Google authentication provider with a client ID and client secret provided by Google:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

Establish the authentication scope

Specify the list of permissions to retrieve from the provider by specifying the Scope. Authentication scopes for common external providers appear in the following table.

PROVIDER	SCOPE
Facebook	`https://www.facebook.com/dialog/oauth`
Google	`https://www.googleapis.com/auth/userinfo.profile`

PROVIDER	SCOPE
Microsoft	`https://login.microsoftonline.com/common/oauth2/v2.0/authorize`
Twitter	`https://api.twitter.com/oauth/authenticate`

In the sample app, Google's `userinfo.profile` scope is automatically added by the framework when AddGoogle is called on the AuthenticationBuilder. If the app requires additional scopes, add them to the options. In the following example, the Google `https://www.googleapis.com/auth/user.birthday.read` scope is added in order to retrieve a user's birthday:

```
options.Scope.Add("https://www.googleapis.com/auth/user.birthday.read");
```

Map user data keys and create claims

In the provider's options, specify a MapJsonKey or MapJsonSubKey for each key/subkey in the external provider's JSON user data for the app identity to read on sign in. For more information on claim types, see ClaimTypes.

The sample app creates locale (`urn:google:locale`) and picture (`urn:google:picture`) claims from the `locale` and `picture` keys in Google user data:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

In `Microsoft.AspNetCore.Identity.UI.Pages.Account.Internal.ExternalLoginModel.OnPostConfirmationAsync` , an IdentityUser (`ApplicationUser`) is signed into the app with SignInAsync. During the sign in process, the UserManager<TUser> can store an `ApplicationUser` claims for user data available from the Principal.

In the sample app, `OnPostConfirmationAsync` (*Account/ExternalLogin.cshtml.cs*) establishes the locale (`urn:google:locale`) and picture (`urn:google:picture`) claims for the signed in `ApplicationUser` , including a claim for

GivenName:

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        ErrorMessage =
            "Error loading external login information during confirmation.";

        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = new IdentityUser
        {
            UserName = Input.Email,
            Email = Input.Email
        };

        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type == ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = true;

                await _signInManager.SignInAsync(user, props);

                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);

                return LocalRedirect(returnUrl);
            }
        }
```

```
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    LoginProvider = info.LoginProvider;
    ReturnUrl = returnUrl;
    return Page();
}
```

By default, a user's claims are stored in the authentication cookie. If the authentication cookie is too large, it can cause the app to fail because:

- The browser detects that the cookie header is too long.
- The overall size of the request is too large.

If a large amount of user data is required for processing user requests:

- Limit the number and size of user claims for request processing to only what the app requires.
- Use a custom ITicketStore for the Cookie Authentication Middleware's SessionStore to store identity across requests. Preserve large quantities of identity information on the server while only sending a small session identifier key to the client.

Save the access token

SaveTokens defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. `SaveTokens` is set to `false` by default to reduce the size of the final authentication cookie.

The sample app sets the value of `SaveTokens` to `true` in GoogleOptions:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

When `OnPostConfirmationAsync` executes, store the access token (ExternalLoginInfo.AuthenticationTokens) from the external provider in the `ApplicationUser`'s `AuthenticationProperties`.

The sample app saves the access token in `OnPostConfirmationAsync` (new user registration) and `OnGetCallbackAsync` (previously registered user) in *Account/ExternalLogin.cshtml.cs*:

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        ErrorMessage =
            "Error loading external login information during confirmation.";

        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = new IdentityUser
        {
            UserName = Input.Email,
            Email = Input.Email
        };

        var result = await _userManager.CreateAsync(user);

        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);

            if (result.Succeeded)
            {
                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type == ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type == "urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = true;

                await _signInManager.SignInAsync(user, props);

                _logger.LogInformation(
                    "User created an account using {Name} provider.",
                    info.LoginProvider);
```

```
                return LocalRedirect(returnUrl);
            }
        }

        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    LoginProvider = info.LoginProvider;
    ReturnUrl = returnUrl;
    return Page();
}
```

How to add additional custom tokens

To demonstrate how to add a custom token, which is stored as part of `SaveTokens`, the sample app adds an AuthenticationToken with the current DateTime for an AuthenticationToken.Name of `TicketCreated`:

```
services.AddAuthentication().AddGoogle(options =>
{
    // Provide the Google Client ID
    options.ClientId = "XXXXXXXXXXXXXXX.apps.googleusercontent.com";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientId" "{Client ID}"

    // Provide the Google Client Secret
    options.ClientSecret = "{Client Secret}";
    // Register with User Secrets using:
    // dotnet user-secrets set "Authentication:Google:ClientSecret" "{Client Secret}"

    options.ClaimActions.MapJsonKey("urn:google:picture", "picture", "url");
    options.ClaimActions.MapJsonKey("urn:google:locale", "locale", "string");
    options.SaveTokens = true;

    options.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens = ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

Creating and adding claims

The framework provides common actions and extension methods for creating and adding claims to the collection. For more information, see the ClaimActionCollectionMapExtensions and ClaimActionCollectionUniqueExtensions.

Users can define custom actions by deriving from ClaimAction and implementing the abstract Run method.

For more information, see Microsoft.AspNetCore.Authentication.OAuth.Claims.

Removal of claim actions and claims

ClaimActionCollection.Remove(String) removes all claim actions for the given ClaimType from the collection. ClaimActionCollectionMapExtensions.DeleteClaim(ClaimActionCollection, String) deletes a claim of the given ClaimType from the identity. DeleteClaim is primarily used with OpenID Connect (OIDC) to remove protocol-generated claims.

Sample app output

```
User Claims

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
    9b342344f-7aab-43c2-1ac1-ba75912ca999
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
    someone@gmail.com
AspNet.Identity.SecurityStamp
    7D4312MOWRYYBFI1KXRPHGOSTBVWSFDE
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname
    Judy
urn:google:locale
    en
urn:google:picture
    https://lh4.googleusercontent.com/-XXXXXX/XXXXXX/XXXXXX/XXXXXX/photo.jpg

Authentication Properties

.Token.access_token
    yc23.AlvoZqz56...1lxltXV7D-ZWP9
.Token.token_type
    Bearer
.Token.expires_at
    2019-04-11T22:14:51.0000000+00:00
.Token.TicketCreated
    4/11/2019 9:14:52 PM
.TokenNames
    access_token;token_type;expires_at;TicketCreated
.persistent
.issued
    Thu, 11 Apr 2019 20:51:06 GMT
.expires
    Thu, 25 Apr 2019 20:51:06 GMT
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Additional resources

- dotnet/AspNetCore engineering SocialSample app: The linked sample app is on the dotnet/AspNetCore GitHub repo's `master` engineering branch. The `master` branch contains code under active development for the next release of ASP.NET Core. To see a version of the sample app for a released version of ASP.NET Core, use the

Branch drop down list to select a release branch (for example `release/{X.Y}`).

Policy schemes in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Authentication policy schemes make it easier to have a single logical authentication scheme potentially use multiple approaches. For example, a policy scheme might use Google authentication for challenges, and cookie authentication for everything else. Authentication policy schemes make it:

- Easy to forward any authentication action to another scheme.
- Forward dynamically based on the request.

All authentication schemes that use derived AuthenticationSchemeOptions and the associated AuthenticationHandler<TOptions>:

- Are automatically policy schemes in ASP.NET Core 2.1 and later.
- Can be enabled via configuring the scheme's options.

```
public class AuthenticationSchemeOptions
{
    /// <summary>
    /// If set, this specifies a default scheme that authentication handlers should
    /// forward all authentication operations to, by default. The default forwarding
    /// logic checks in this order:
    /// 1. The most specific ForwardAuthenticate/Challenge/Forbid/SignIn/SignOut
    /// 2. The ForwardDefaultSelector
    /// 3. ForwardDefault
    /// The first non null result is used as the target scheme to forward to.
    /// </summary>
    public string ForwardDefault { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should forward
    /// AuthenticateAsync calls to. For example:
    /// Context.AuthenticateAsync("ThisScheme") =>
    ///               Context.AuthenticateAsync("ForwardAuthenticateValue");
    /// Set the target to the current scheme to disable forwarding and allow
    /// normal processing.
    /// </summary>
    public string ForwardAuthenticate { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should forward
    /// ChallengeAsync calls to. For example:
    /// Context.ChallengeAsync("ThisScheme") =>
    ///                         Context.ChallengeAsync("ForwardChallengeValue");
    /// Set the target to the current scheme to disable forwarding and allow normal
    /// processing.
    /// </summary>
    public string ForwardChallenge { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should forward
    /// ForbidAsync calls to.For example:
    /// Context.ForbidAsync("ThisScheme")
    ///                               => Context.ForbidAsync("ForwardForbidValue");
    /// Set the target to the current scheme to disable forwarding and allow normal
    /// processing.
    /// </summary>
    public string ForwardForbid { get; set; }

    /// <summary>
```

```
    /// If set, this specifies the target scheme that this scheme should forward
    /// SignInAsync calls to. For example:
    /// Context.SignInAsync("ThisScheme") =>
    ///                               Context.SignInAsync("ForwardSignInValue");
    /// Set the target to the current scheme to disable forwarding and allow normal
    /// processing.
    /// </summary>
    public string ForwardSignIn { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should forward
    /// SignOutAsync calls to. For example:
    /// Context.SignOutAsync("ThisScheme") =>
    ///                             Context.SignOutAsync("ForwardSignOutValue");
    /// Set the target to the current scheme to disable forwarding and allow normal
    /// processing.
    /// </summary>
    public string ForwardSignOut { get; set; }

    /// <summary>
    /// Used to select a default scheme for the current request that authentication
    /// handlers should forward all authentication operations to by default. The
    /// default forwarding checks in this order:
    /// 1. The most specific ForwardAuthenticate/Challenge/Forbid/SignIn/SignOut
    /// 2. The ForwardDefaultSelector
    /// 3. ForwardDefault.
    /// The first non null result will be used as the target scheme to forward to.
    /// </summary>
    public Func<HttpContext, string> ForwardDefaultSelector { get; set; }
}
```

Examples

The following example shows a higher level scheme that combines lower level schemes. Google authentication is used for challenges, and cookie authentication is used for everything else:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
       .AddCookie(options => options.ForwardChallenge = "Google")
       .AddGoogle(options => { });
}
```

The following example enables dynamic selection of schemes on a per request basis. That is, how to mix cookies and API authentication:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
        .AddCookie(options =>
        {
            // For example, can foward any requests that start with /api
            // to the api scheme.
            options.ForwardDefaultSelector = ctx =>
               ctx.Request.Path.StartsWithSegments("/api") ? "Api" : null;
        })
        .AddYourApiAuth("Api");
}
```

Authenticate users with WS-Federation in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

This tutorial demonstrates how to enable users to sign in with a WS-Federation authentication provider like Active Directory Federation Services (ADFS) or Azure Active Directory (AAD). It uses the ASP.NET Core sample app described in Facebook, Google, and external provider authentication.

For ASP.NET Core apps, WS-Federation support is provided by Microsoft.AspNetCore.Authentication.WsFederation. This component is ported from Microsoft.Owin.Security.WsFederation and shares many of that component's mechanics. However, the components differ in a couple of important ways.

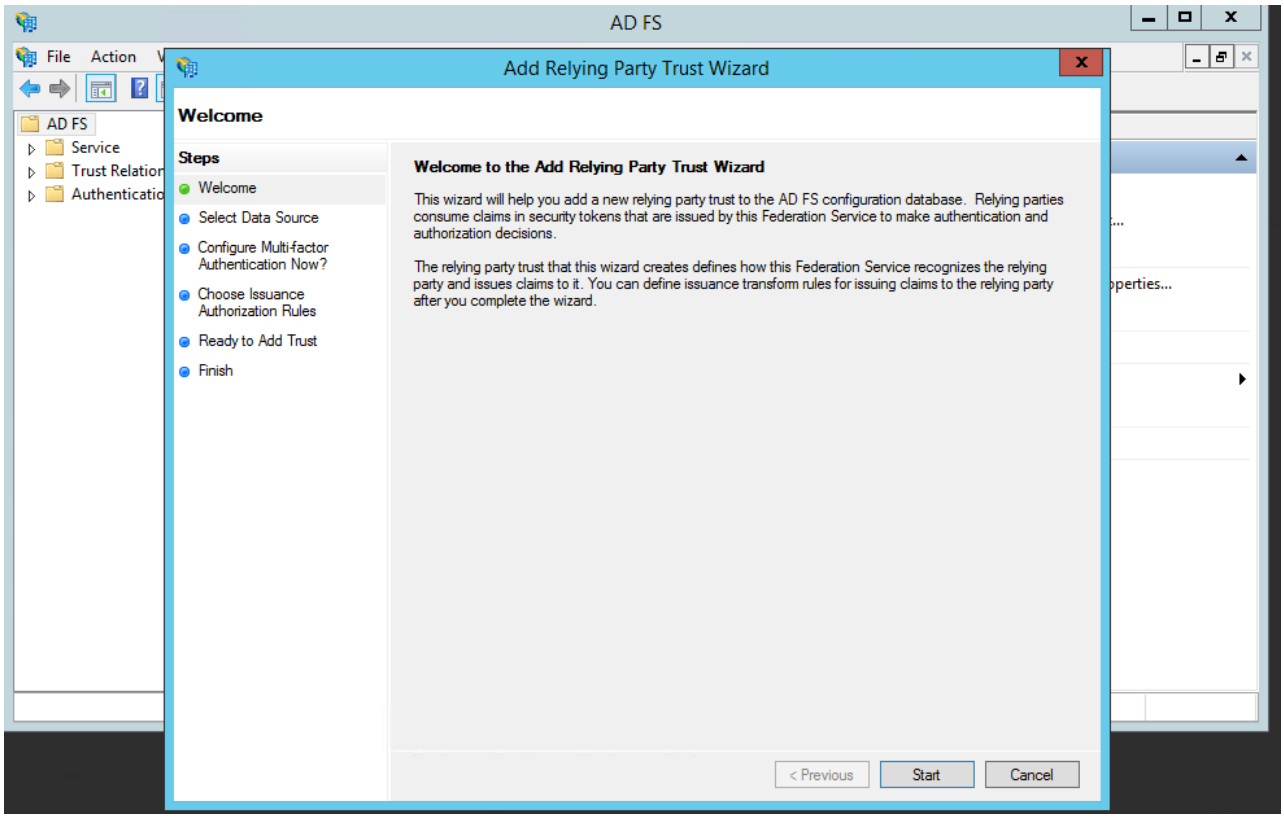
By default, the new middleware:

- Doesn't allow unsolicited logins. This feature of the WS-Federation protocol is vulnerable to XSRF attacks. However, it can be enabled with the `AllowUnsolicitedLogins` option.
- Doesn't check every form post for sign-in messages. Only requests to the `CallbackPath` are checked for sign-ins. `CallbackPath` defaults to `/signin-wsfed` but can be changed via the inherited RemoteAuthenticationOptions.CallbackPath property of the WsFederationOptions class. This path can be shared with other authentication providers by enabling the SkipUnrecognizedRequests option.

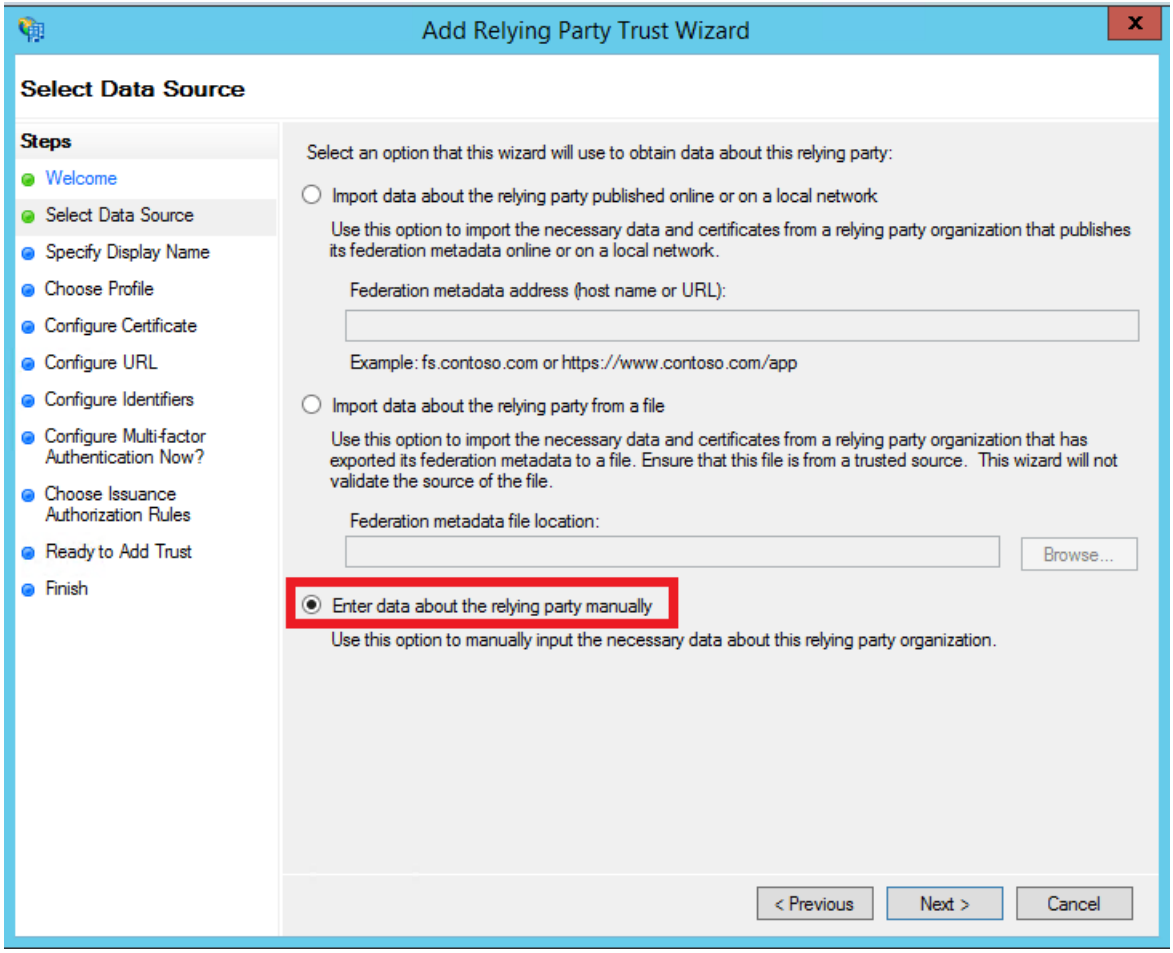
Register the app with Active Directory

Active Directory Federation Services

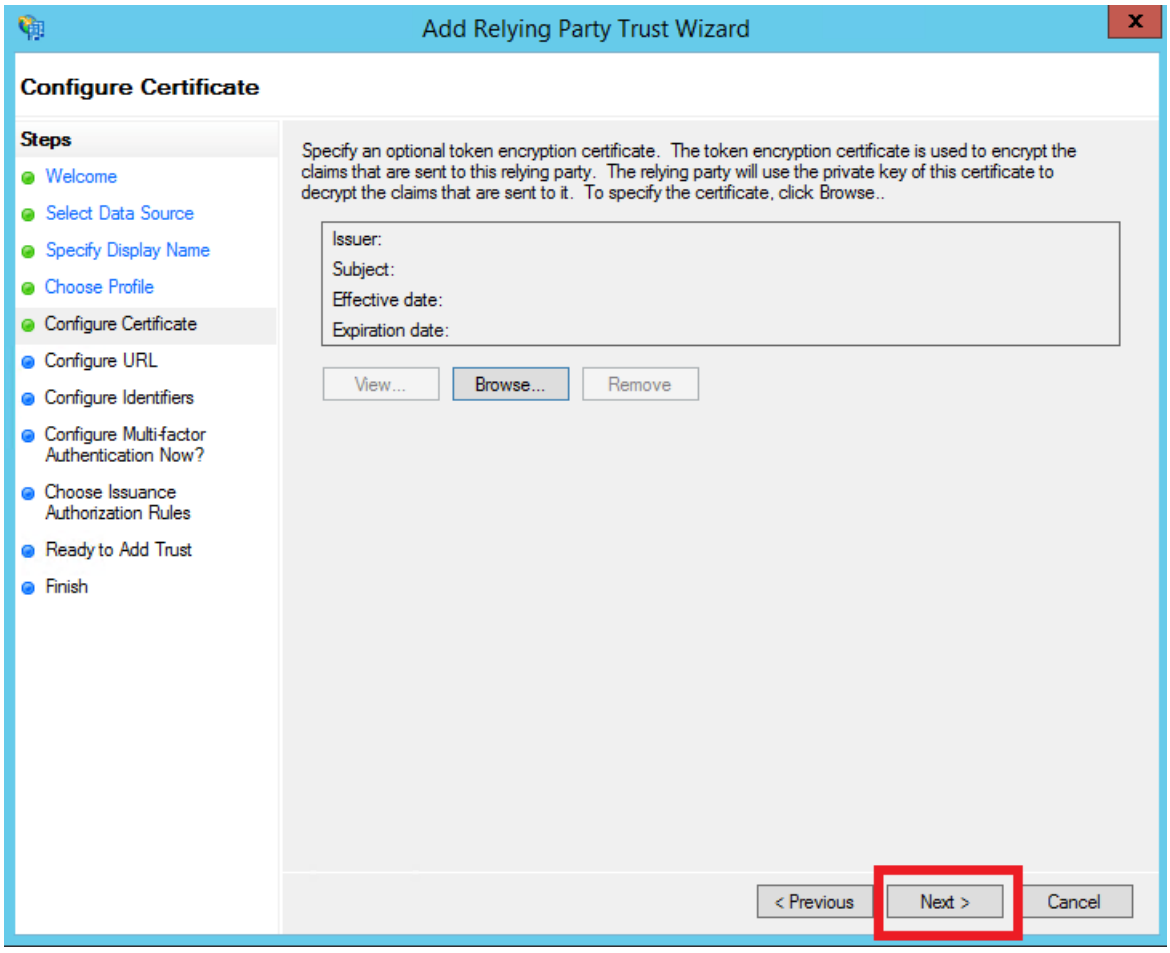
- Open the server's **Add Relying Party Trust Wizard** from the ADFS Management console:



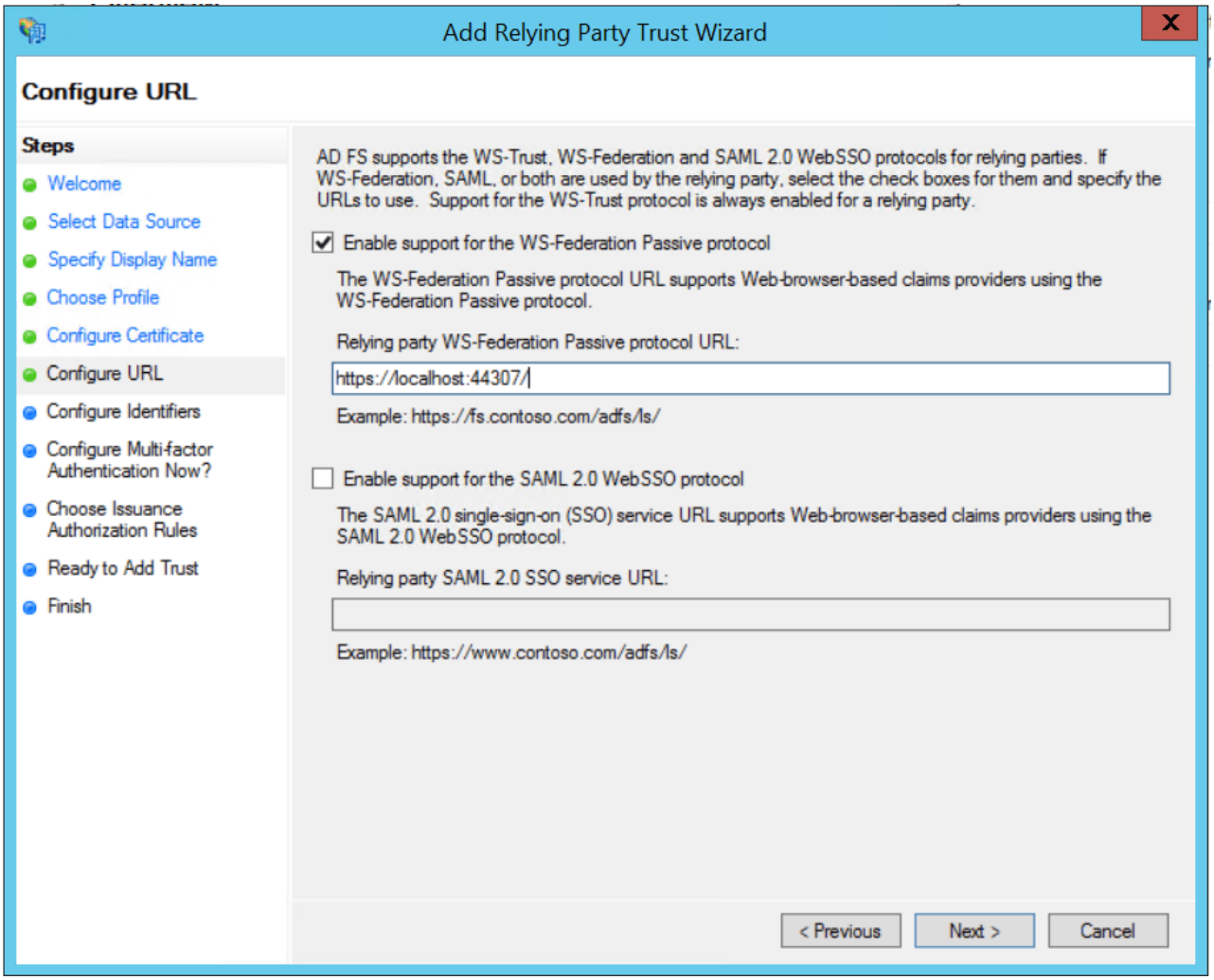
- Choose to enter data manually:



- Enter a display name for the relying party. The name isn't important to the ASP.NET Core app.
- Microsoft.AspNetCore.Authentication.WsFederation lacks support for token encryption, so don't configure a token encryption certificate:

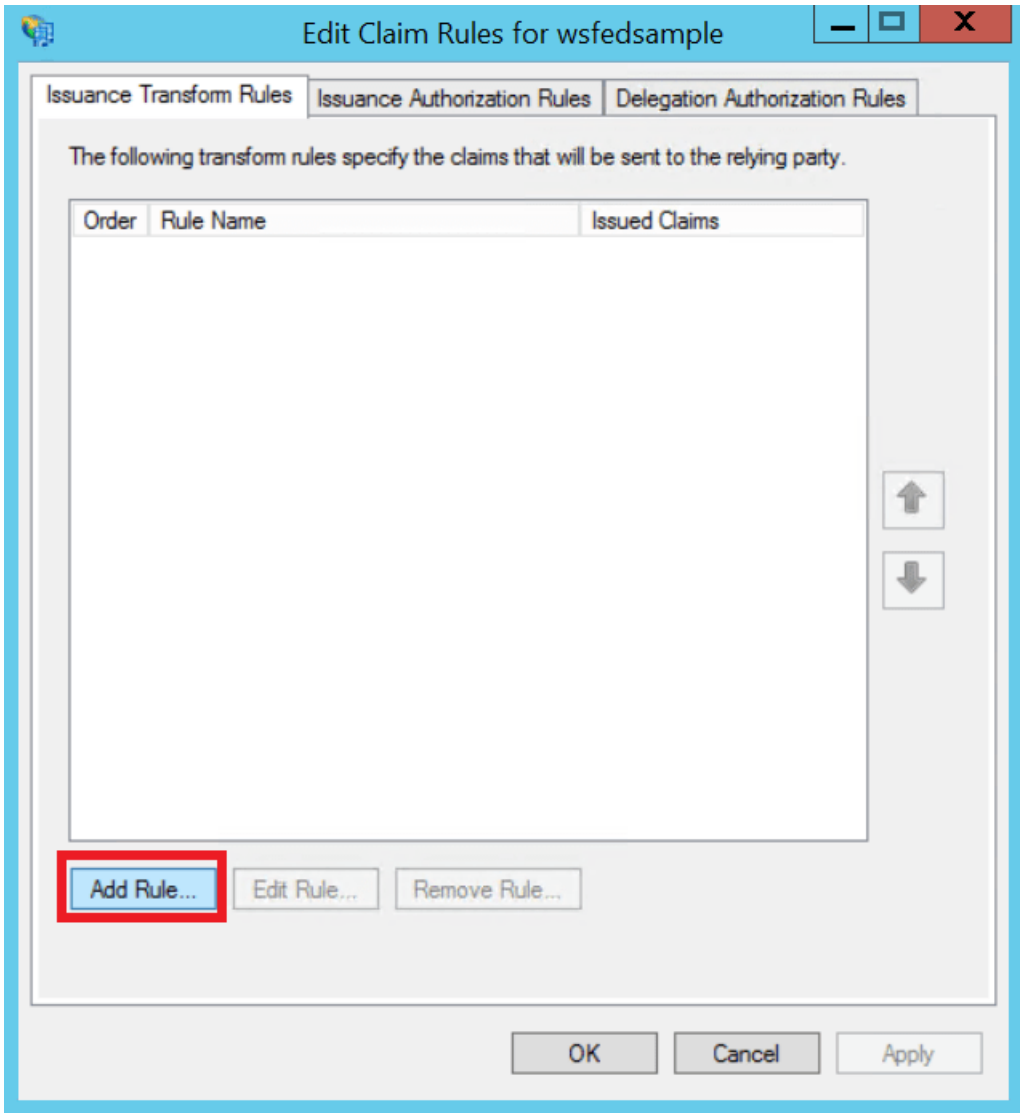


- Enable support for WS-Federation Passive protocol, using the app's URL. Verify the port is correct for the app:

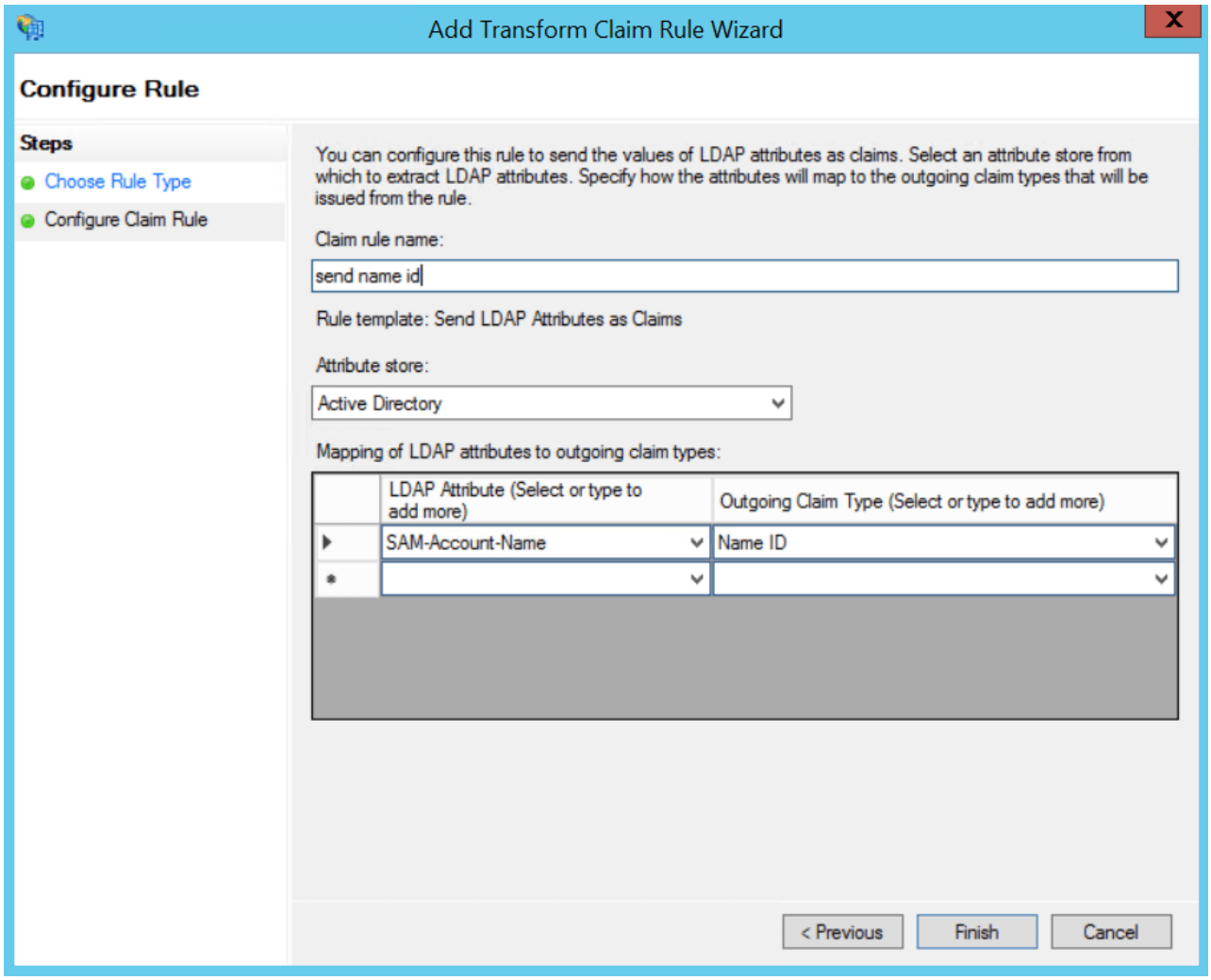


> **NOTE**
>
> This must be an HTTPS URL. IIS Express can provide a self-signed certificate when hosting the app during development. Kestrel requires manual certificate configuration. See the Kestrel documentation for more details.

- Click **Next** through the rest of the wizard and **Close** at the end.
- ASP.NET Core Identity requires a **Name ID** claim. Add one from the **Edit Claim Rules** dialog:



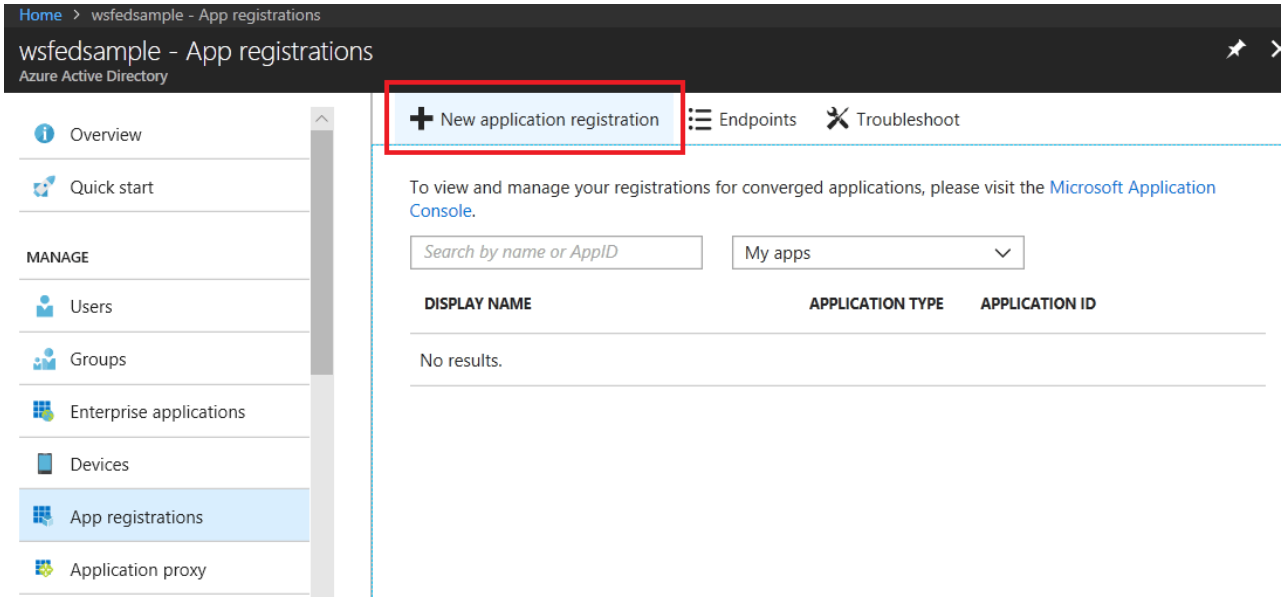
- In the **Add Transform Claim Rule Wizard**, leave the default **Send LDAP Attributes as Claims** template selected, and click **Next**. Add a rule mapping the **SAM-Account-Name** LDAP attribute to the **Name ID** outgoing claim:



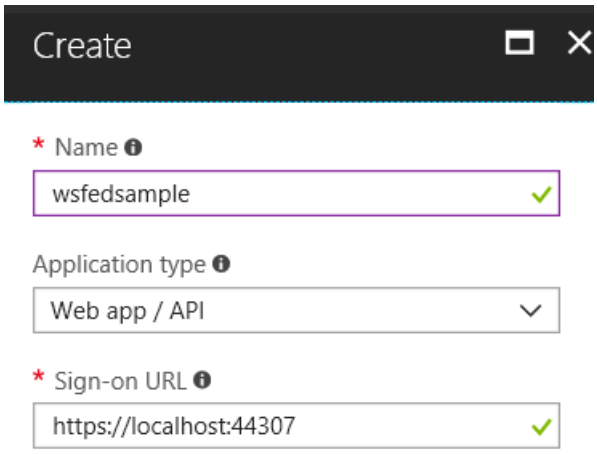
- Click **Finish** > **OK** in the **Edit Claim Rules** window.

Azure Active Directory

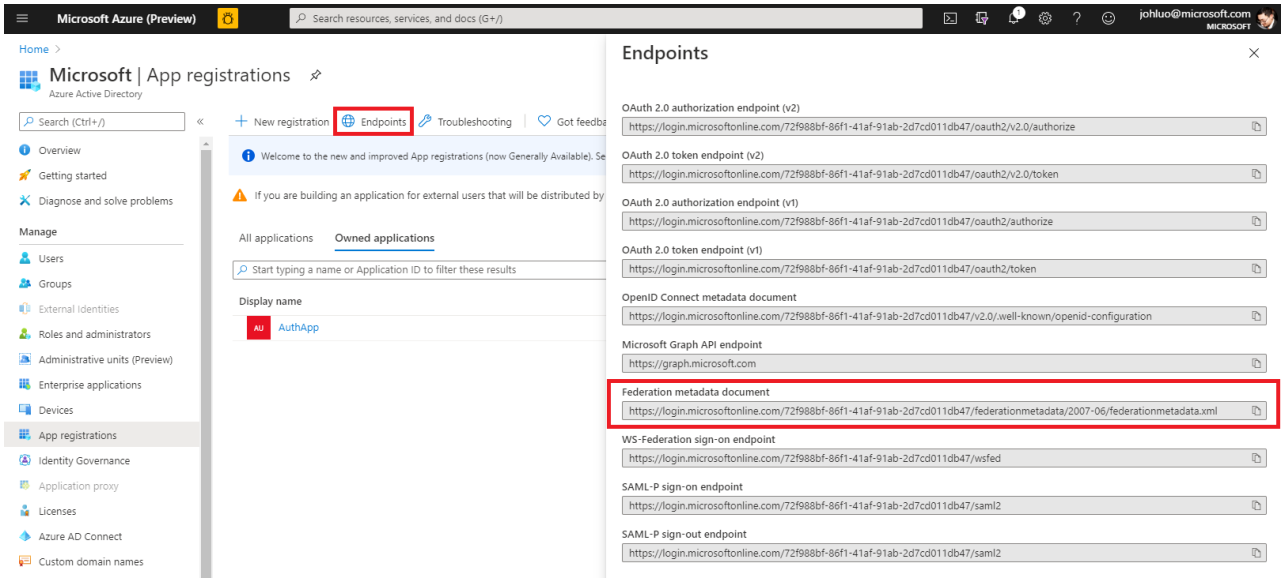
- Navigate to the AAD tenant's app registrations blade. Click **New application registration**:



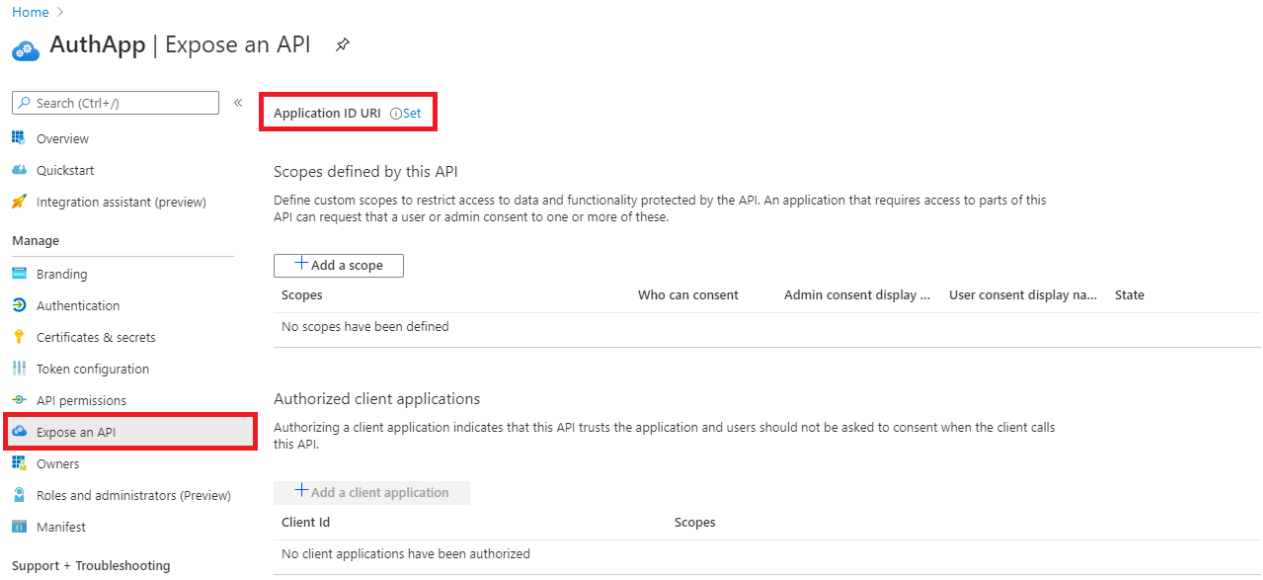
- Enter a name for the app registration. This isn't important to the ASP.NET Core app.
- Enter the URL the app listens on as the **Sign-on URL**:



- Click **Endpoints** and note the **Federation Metadata Document** URL. This is the WS-Federation middleware's `MetadataAddress`:



- Navigate to the new app registration. Click **Expose an API**. Click Application ID URI **Set** > **Save**. Make note of the **Application ID URI**. This is the WS-Federation middleware's `Wtrealm`:



Use WS-Federation without ASP.NET Core Identity

The WS-Federation middleware can be used without Identity. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(sharedOptions =>
    {
        sharedOptions.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultChallengeScheme = WsFederationDefaults.AuthenticationScheme;
    })
     .AddWsFederation(options =>
     {
         options.Wtrealm = Configuration["wsfed:realm"];
         options.MetadataAddress = Configuration["wsfed:metadata"];
     })
     .AddCookie();

    services.AddControllersWithViews();
    services.AddRazorPages();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(sharedOptions =>
    {
        sharedOptions.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultSignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultChallengeScheme = WsFederationDefaults.AuthenticationScheme;
    })
    .AddWsFederation(options =>
    {
        options.Wtrealm = Configuration["wsfed:realm"];
        options.MetadataAddress = Configuration["wsfed:metadata"];
    })
    .AddCookie();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseAuthentication();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Add WS-Federation as an external login provider for ASP.NET Core Identity

- Add a dependency on Microsoft.AspNetCore.Authentication.WsFederation to the project.
- Add WS-Federation to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();


    services.AddAuthentication()
        .AddWsFederation(options =>
        {
            // MetadataAddress represents the Active Directory instance used to authenticate users.
            options.MetadataAddress = "https://<ADFS FQDN or AAD tenant>/FederationMetadata/2007-
06/FederationMetadata.xml";

            // Wtrealm is the app's identifier in the Active Directory instance.
            // For ADFS, use the relying party's identifier, its WS-Federation Passive protocol URL:
            options.Wtrealm = "https://localhost:44307/";

            // For AAD, use the Application ID URI from the app registration's Overview blade:
            options.Wtrealm = "api://bbd35166-7c13-49f3-8041-9551f2847b69";
        });

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddAuthentication()
        .AddWsFederation(options =>
        {
            // MetadataAddress represents the Active Directory instance used to authenticate users.
            options.MetadataAddress = "https://<ADFS FQDN or AAD tenant>/FederationMetadata/2007-
06/FederationMetadata.xml";

            // Wtrealm is the app's identifier in the Active Directory instance.
            // For ADFS, use the relying party's identifier, its WS-Federation Passive protocol URL:
            options.Wtrealm = "https://localhost:44307/";

            // For AAD, use the Application ID URI from the app registration's Overview blade:
            options.Wtrealm = "api://bbd35166-7c13-49f3-8041-9551f2847b69";
        });

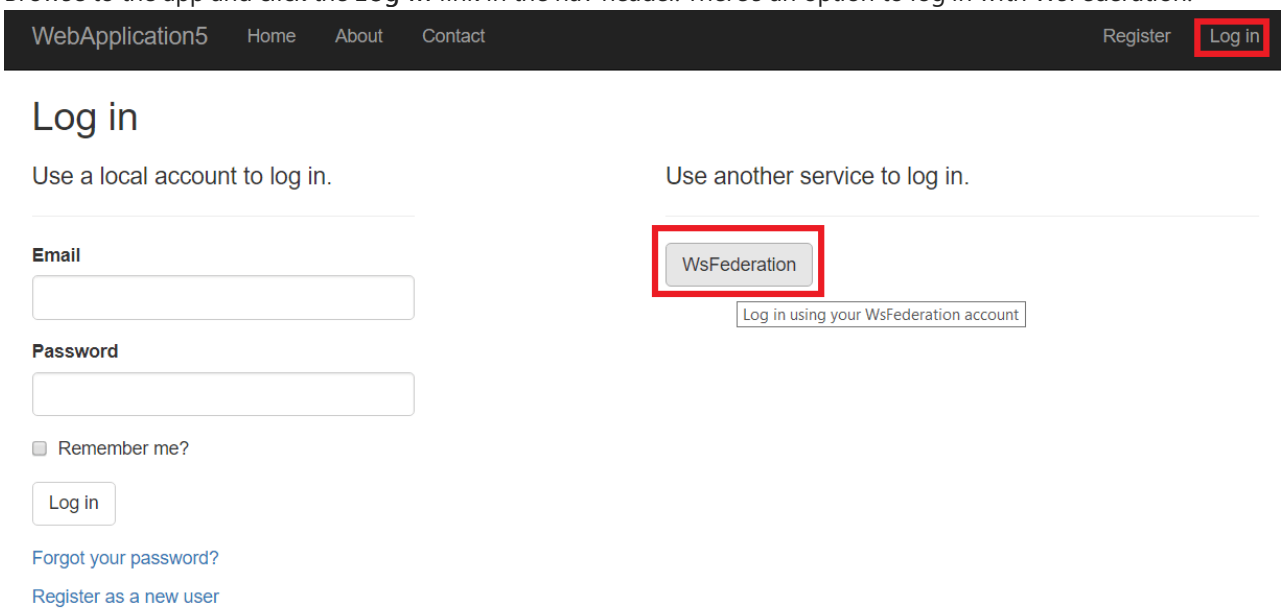
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

The AddAuthentication(String) overload sets the DefaultScheme property. The AddAuthentication(Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

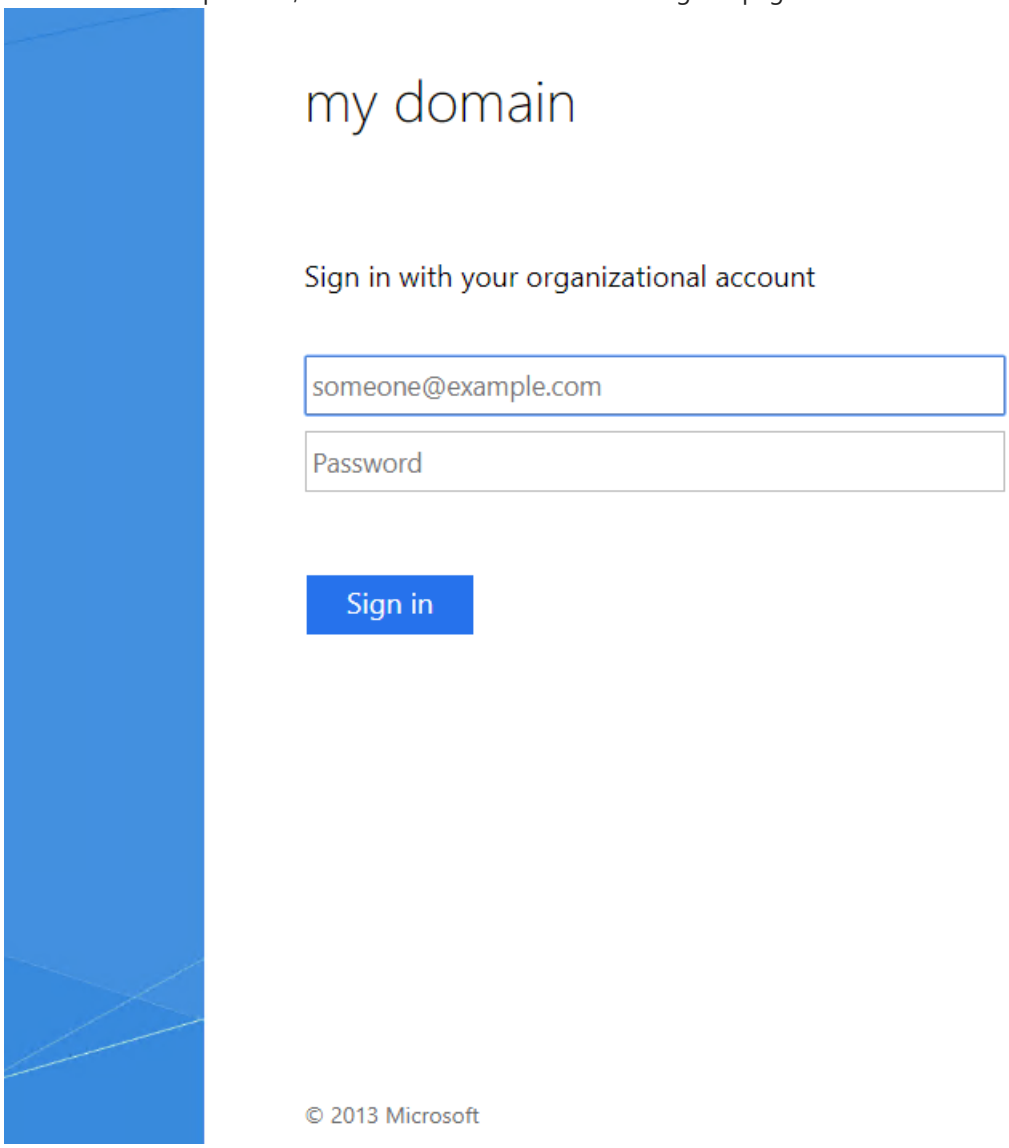
AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Log in with WS-Federation

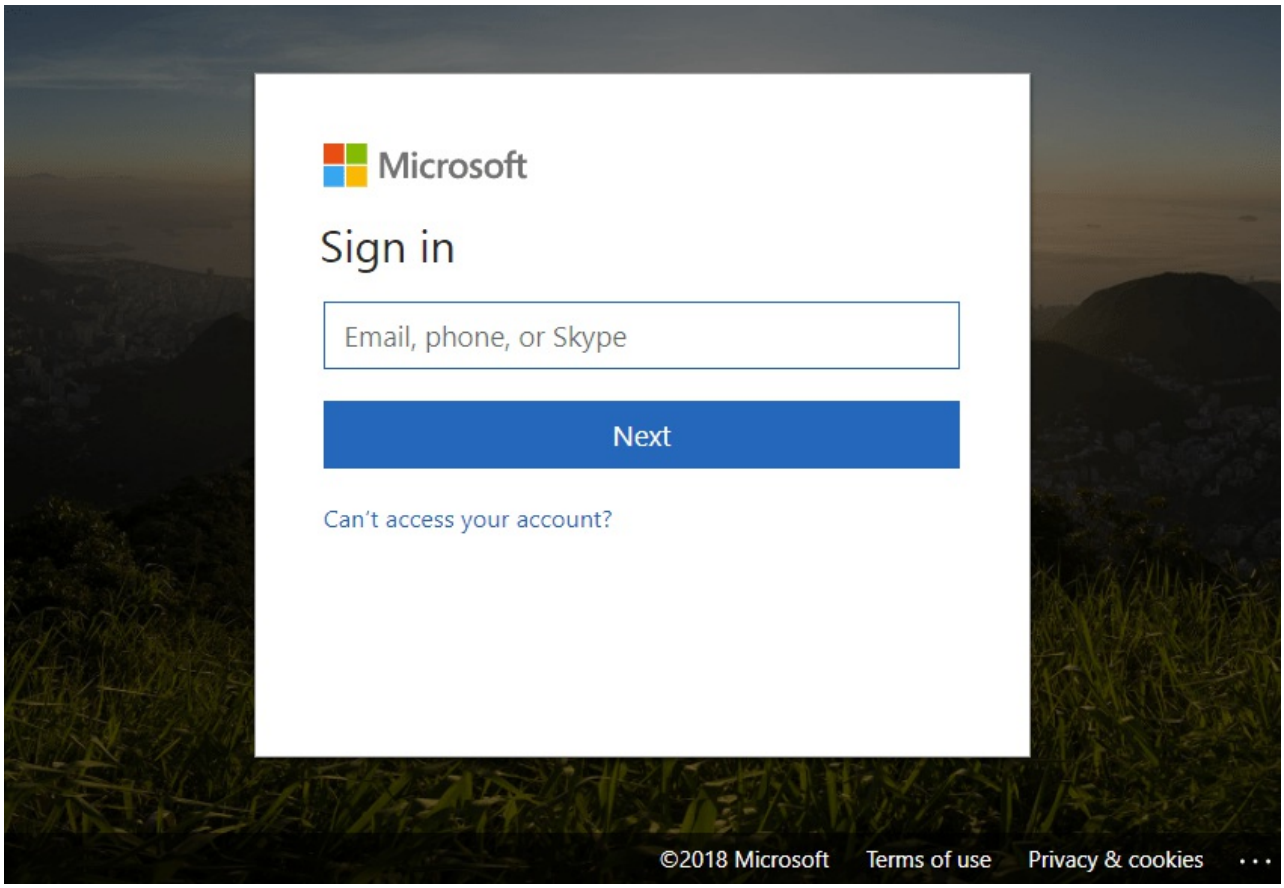
Browse to the app and click the **Log in** link in the nav header. There's an option to log in with WsFederation:



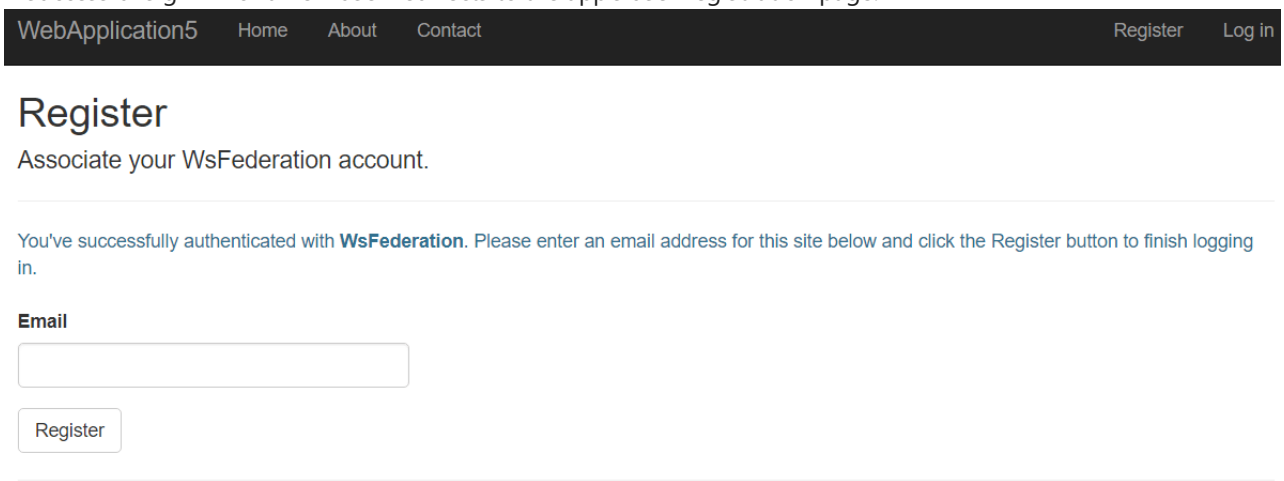
With ADFS as the provider, the button redirects to an ADFS sign-in page:



With Azure Active Directory as the provider, the button redirects to an AAD sign-in page:



A successful sign-in for a new user redirects to the app's user registration page:



Account confirmation and password recovery in ASP.NET Core

9/22/2020 • 15 minutes to read • Edit Online

By Rick Anderson, Ponant, and Joe Audette

This tutorial shows how to build an ASP.NET Core app with email confirmation and password reset. This tutorial is **not** a beginning topic. You should be familiar with:

- ASP.NET Core
- Authentication
- Entity Framework Core

Prerequisites

.NET Core 3.0 SDK or later

Create and test a web app with authentication

Run the following commands to create a web app with authentication.

```
dotnet new webapp -au Individual -uld -o WebPWrecover
cd WebPWrecover
dotnet run
```

Run the app, select the **Register** link, and register a user. Once registered, you are redirected to the to `/Identity/Account/RegisterConfirmation` page which contains a link to simulate email confirmation:

- Select the `Click here to confirm your account` link.
- Select the **Login** link and sign-in with the same credentials.
- Select the `Hello YourEmail@provider.com!` link, which redirects you to the `/Identity/Account/Manage/PersonalData` page.
- Select the **Personal data** tab on the left, and then select **Delete**.

Configure an email provider

In this tutorial, SendGrid is used to send email. You need a SendGrid account and key to send email. You can use other email providers. We recommend you use SendGrid or another email service to send email. SMTP is difficult to secure and set up correctly.

The SendGrid account may require adding a Sender.

Create a class to fetch the secure email key. For this sample, create *Services/AuthMessageSenderOptions.cs*:

```
public class AuthMessageSenderOptions
{
    public string SendGridUser { get; set; }
    public string SendGridKey { get; set; }
}
```

Configure SendGrid user secrets

Set the `SendGridUser` and `SendGridKey` with the secret-manager tool. For example:

```
dotnet user-secrets set SendGridUser RickAndMSFT
dotnet user-secrets set SendGridKey <key>

Successfully saved SendGridUser = RickAndMSFT to the secret store.
```

On Windows, Secret Manager stores keys/value pairs in a *secrets.json* file in the `%APPDATA%/Microsoft/UserSecrets/<WebAppName-userSecretsId>` directory.

The contents of the *secrets.json* file aren't encrypted. The following markup shows the *secrets.json* file. The `SendGridKey` value has been removed.

```
{
  "SendGridUser": "RickAndMSFT",
  "SendGridKey": "<key removed>"
}
```

For more information, see the Options pattern and configuration.

Install SendGrid

This tutorial shows how to add email notifications through SendGrid, but you can send email using SMTP and other mechanisms.

Install the `SendGrid` NuGet package:

- Visual Studio
- .NET Core CLI

From the Package Manager Console, enter the following command:

```
Install-Package SendGrid
```

See Get Started with SendGrid for Free to register for a free SendGrid account.

Implement IEmailSender

To Implement `IEmailSender`, create *Services/EmailSender.cs* with code similar to the following:

```
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;
using System.Threading.Tasks;

namespace WebPWrecover.Services
{
    public class EmailSender : IEmailSender
    {
        public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public AuthMessageSenderOptions Options { get; } //set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            return Execute(Options.SendGridKey, subject, message, email);
        }

        public Task Execute(string apiKey, string subject, string message, string email)
        {
            var client = new SendGridClient(apiKey);
            var msg = new SendGridMessage()
            {
                From = new EmailAddress("Joe@contoso.com", Options.SendGridUser),
                Subject = subject,
                PlainTextContent = message,
                HtmlContent = message
            };
            msg.AddTo(new EmailAddress(email));

            // Disable click tracking.
            // See https://sendgrid.com/docs/User_Guide/Settings/tracking.html
            msg.SetClickTracking(false, false);

            return client.SendEmailAsync(msg);
        }
    }
}
```

Configure startup to support email

Add the following code to the `ConfigureServices` method in the *Startup.cs* file:

- Add `EmailSender` as a transient service.
- Register the `AuthMessageSenderOptions` configuration instance.

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
                options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    // requires
    // using Microsoft.AspNetCore.Identity.UI.Services;
    // using WebPWrecover.Services;
    services.AddTransient<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration);

    services.AddRazorPages();
}
```

Scaffold RegisterConfirmation

Follow the instructions for Scaffold Identity and scaffold `RegisterConfirmation`.

Disable default account verification

With the default templates, the user is redirected to the `Account.RegisterConfirmation` where they can select a link to have the account confirmed. The default `Account.RegisterConfirmation` is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set `DisplayConfirmAccountLink = false` in */Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs*:

```
[AllowAnonymous]
public class RegisterConfirmationModel : PageModel
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IEmailSender _sender;

    public RegisterConfirmationModel(UserManager<IdentityUser> userManager, IEmailSender sender)
    {
        _userManager = userManager;
        _sender = sender;
    }

    public string Email { get; set; }

    public bool DisplayConfirmAccountLink { get; set; }

    public string EmailConfirmationUrl { get; set; }

    public async Task<IActionResult> OnGetAsync(string email, string returnUrl = null)
    {
        if (email == null)
        {
            return RedirectToPage("/Index");
        }

        var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
        {
            return NotFound($"Unable to load user with email '{email}'.");
        }

        Email = email;
        // Once you add a real email sender, you should remove this code that lets you confirm the account
        DisplayConfirmAccountLink = false;
        if (DisplayConfirmAccountLink)
        {
            var userId = await _userManager.GetUserIdAsync(user);
            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            EmailConfirmationUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = userId, code = code, returnUrl = returnUrl },
                protocol: Request.Scheme);
        }

        return Page();
    }
}
```

Register, confirm email, and reset password

Run the web app, and test the account confirmation and password recovery flow.

- Run the app and register a new user
- Check your email for the account confirmation link. See Debug email if you don't get the email.
- Click the link to confirm your email.
- Sign in with your email and password.
- Sign out.

Test password reset

- If you're signed in, select **Logout**.

- Select the **Log in** link and select the **Forgot your password?** link.
- Enter the email you used to register the account.
- An email with a link to reset your password is sent. Check your email and click the link to reset your password. After your password has been successfully reset, you can sign in with your email and new password.

Resend email confirmation

In ASP.NET Core 5.0 and later, select the **Resend email confirmation** link on the **Login** page.

Change email and activity timeout

The default inactivity timeout is 14 days. The following code sets the inactivity timeout to 5 days:

```
services.ConfigureApplicationCookie(o => {
    o.ExpireTimeSpan = TimeSpan.FromDays(5);
    o.SlidingExpiration = true;
});
```

Change all data protection token lifespans

The following code changes all data protection tokens timeout period to 3 hours:

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
                  options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.Configure<DataProtectionTokenProviderOptions>(o =>
       o.TokenLifespan = TimeSpan.FromHours(3));

    services.AddTransient<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration);

    services.AddRazorPages();
}
```

The built in Identity user tokens (see AspNetCore/src/Identity/Extensions.Core/src/TokenOptions.cs)have a one day timeout.

Change the email token lifespan

The default token lifespan of the Identity user tokens is one day. This section shows how to change the email token lifespan.

Add a custom DataProtectorTokenProvider<TUser> and DataProtectionTokenProviderOptions:

```
public class CustomEmailConfirmationTokenProvider<TUser>
                                     : DataProtectorTokenProvider<TUser> where TUser : class
{
    public CustomEmailConfirmationTokenProvider(IDataProtectionProvider dataProtectionProvider,
        IOptions<EmailConfirmationTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
                                       : base(dataProtectionProvider, options, logger)
    {

    }
}
public class EmailConfirmationTokenProviderOptions : DataProtectionTokenProviderOptions
{
    public EmailConfirmationTokenProviderOptions()
    {
        Name = "EmailDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(4);
    }
}
```

Add the custom provider to the service container:

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(config =>
    {
        config.SignIn.RequireConfirmedEmail = true;
        config.Tokens.ProviderMap.Add("CustomEmailConfirmation",
            new TokenProviderDescriptor(
                typeof(CustomEmailConfirmationTokenProvider<IdentityUser>)));
        config.Tokens.EmailConfirmationTokenProvider = "CustomEmailConfirmation";
      }).AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddTransient<CustomEmailConfirmationTokenProvider<IdentityUser>>();

    services.AddTransient<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration);

    services.AddRazorPages();
}
```

Debug email

If you can't get email working:

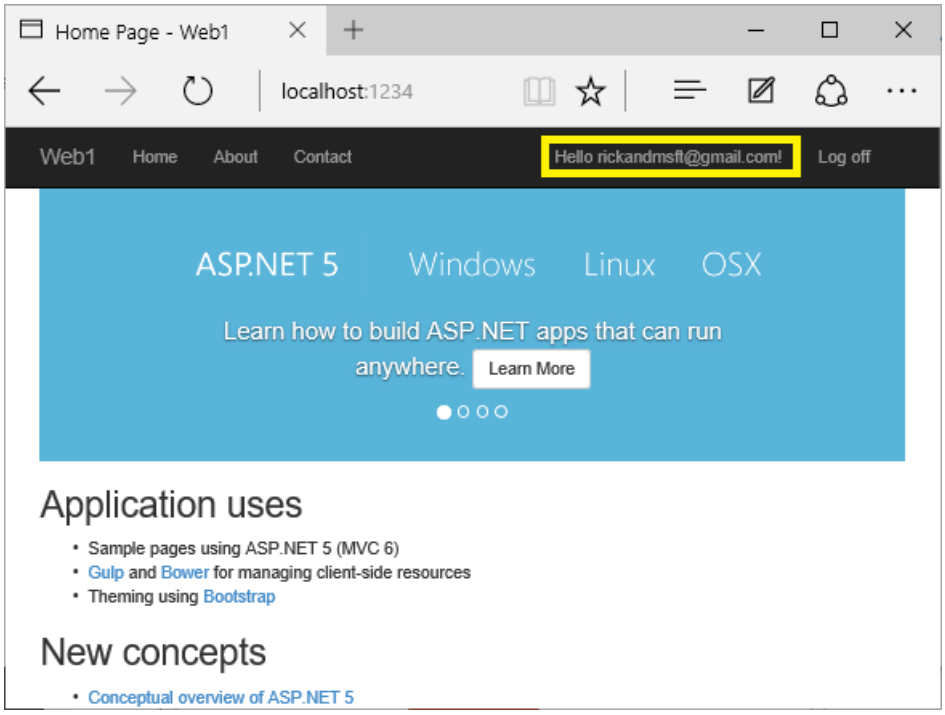
- Set a breakpoint in `EmailSender.Execute` to verify `SendGridClient.SendEmailAsync` is called.
- Create a console app to send email using similar code to `EmailSender.Execute`.
- Review the Email Activity page.
- Check your spam folder.
- Try another email alias on a different email provider (Microsoft, Yahoo, Gmail, etc.)
- Try sending to different email accounts.

A security best practice is to **not** use production secrets in test and development. If you publish the app to Azure, set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is set up to read keys from environment variables.

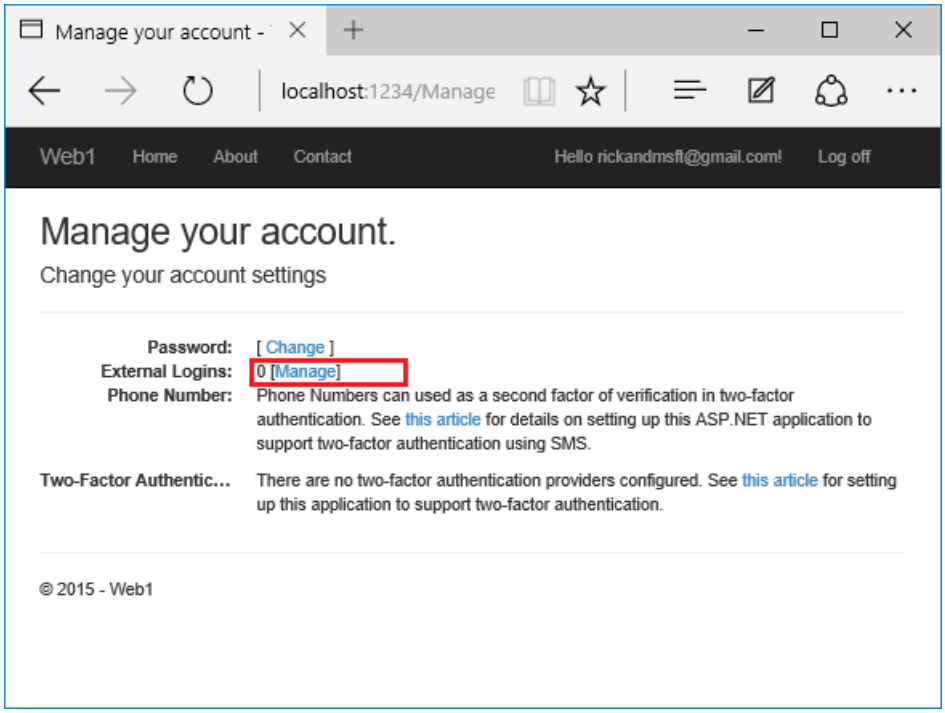
Combine social and local login accounts

To complete this section, you must first enable an external authentication provider. See Facebook, Google, and external provider authentication.

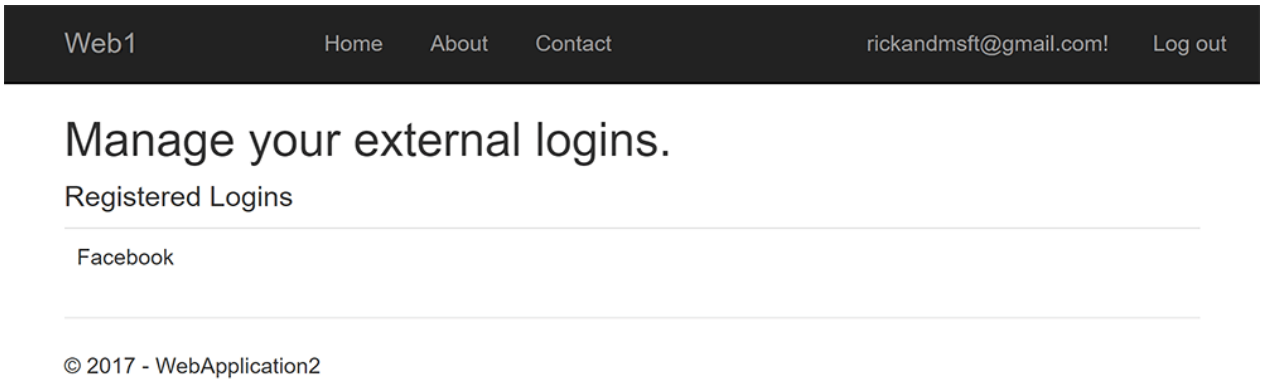
You can combine local and social accounts by clicking on your email link. In the following sequence, "RickAndMSFT@gmail.com" is first created as a local login; however, you can create the account as a social login first, then add a local login.



Click on the **Manage** link. Note the 0 external (social logins) associated with this account.



Click the link to another login service and accept the app requests. In the following image, Facebook is the external authentication provider:



The two accounts have been combined. You are able to sign in with either account. You might want your users to add local accounts in case their social login authentication service is down, or more likely they've lost access to their social account.

Enable account confirmation after a site has users

Enabling account confirmation on a site with users locks out all the existing users. Existing users are locked out because their accounts aren't confirmed. To work around existing user lockout, use one of the following approaches:

- Update the database to mark all existing users as being confirmed.
- Confirm existing users. For example, batch-send emails with confirmation links.

Prerequisites

.NET Core 2.2 SDK or later

Create a web app and scaffold Identity

Run the following commands to create a web app with authentication.

```
dotnet new webapp -au Individual -uld -o WebPWrecover
cd WebPWrecover
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet aspnet-codegenerator identity -dc WebPWrecover.Data.ApplicationDbContext --files
"Account.Register;Account.Login;Account.Logout;Account.ConfirmEmail"
dotnet ef database drop -f
dotnet ef database update
dotnet run
```

NOTE

If PasswordOptions are configured in `Startup.ConfigureServices`, `[StringLength]` attribute configuration might be required for the `Password` property in scaffolded Identity pages. An `InputModel` `Password` property is found in the `Areas/Identity/Pages/Account/Register.cshtml.cs` file after scaffolding Identity.

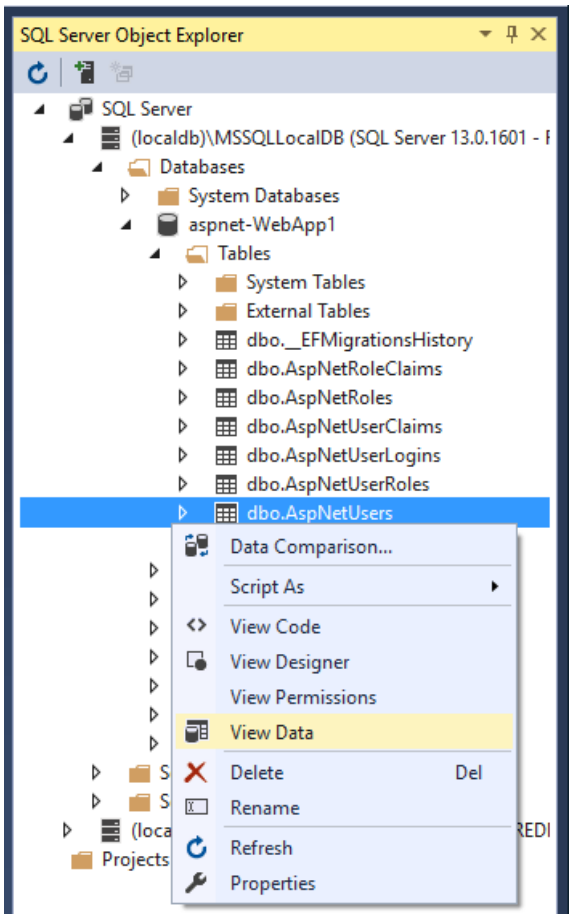
Test new user registration

Run the app, select the **Register** link, and register a user. At this point, the only validation on the email is with the `[EmailAddress]` attribute. After submitting the registration, you are logged into the app. Later in the tutorial, the code is updated so new users can't sign in until their email is validated.

View the Identity database

- Visual Studio
- .NET Core CLI

- From the **View** menu, select **SQL Server Object Explorer** (SSOX).
- Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



Note the table's `EmailConfirmed` field is `False`.

You might want to use this email again in the next step when the app sends a confirmation email. Right-click on the row and select **Delete**. Deleting the email alias makes it easier in the following steps.

Require email confirmation

It's a best practice to confirm the email of a new user registration. Email confirmation helps to verify they're not impersonating someone else (that is, they haven't registered with someone else's email). Suppose you had a discussion forum, and you wanted to prevent "yli@example.com" from registering as "nolivetto@contoso.com". Without email confirmation, "nolivetto@contoso.com" could receive unwanted email from your app. Suppose the user accidentally registered as "ylo@example.com" and hadn't noticed the misspelling of "yli". They wouldn't be able to use password recovery because the app doesn't have their correct email. Email confirmation provides limited protection from bots. Email confirmation doesn't provide protection from malicious users with many email accounts.

You generally want to prevent new users from posting any data to your web site before they have a confirmed email.

Update `Startup.ConfigureServices` to require a confirmed email:

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<IdentityUser>(config =>
    {
        config.SignIn.RequireConfirmedEmail = true;
    })
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    // requires
    // using Microsoft.AspNetCore.Identity.UI.Services;
    // using WebPWrecover.Services;
    services.AddTransient<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration);

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

`config.SignIn.RequireConfirmedEmail = true;` prevents registered users from logging in until their email is confirmed.

Configure email provider

In this tutorial, SendGrid is used to send email. You need a SendGrid account and key to send email. You can use other email providers. ASP.NET Core 2.x includes `System.Net.Mail`, which allows you to send email from your app. We recommend you use SendGrid or another email service to send email. SMTP is difficult to secure and set up correctly.

Create a class to fetch the secure email key. For this sample, create *Services/AuthMessageSenderOptions.cs*:

```
public class AuthMessageSenderOptions
{
    public string SendGridUser { get; set; }
    public string SendGridKey { get; set; }
}
```

Configure SendGrid user secrets

Set the `SendGridUser` and `SendGridKey` with the secret-manager tool. For example:

```
C:/WebAppl>dotnet user-secrets set SendGridUser RickAndMSFT
info: Successfully saved SendGridUser = RickAndMSFT to the secret store.
```

On Windows, Secret Manager stores keys/value pairs in a *secrets.json* file in the `%APPDATA%/Microsoft/UserSecrets/<WebAppName-userSecretsId>` directory.

The contents of the *secrets.json* file aren't encrypted. The following markup shows the *secrets.json* file. The `SendGridKey` value has been removed.

```
{
  "SendGridUser": "RickAndMSFT",
  "SendGridKey": "<key removed>"
}
```

For more information, see the Options pattern and configuration.

Install SendGrid

This tutorial shows how to add email notifications through SendGrid, but you can send email using SMTP and other mechanisms.

Install the `SendGrid` NuGet package:

- Visual Studio
- .NET Core CLI

From the Package Manager Console, enter the following command:

```
Install-Package SendGrid
```

See Get Started with SendGrid for Free to register for a free SendGrid account.

Implement IEmailSender

To Implement `IEmailSender`, create *Services/EmailSender.cs* with code similar to the following:

```
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;
using System.Threading.Tasks;

namespace WebPWrecover.Services
{
    public class EmailSender : IEmailSender
    {
        public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public AuthMessageSenderOptions Options { get; } //set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            return Execute(Options.SendGridKey, subject, message, email);
        }

        public Task Execute(string apiKey, string subject, string message, string email)
        {
            var client = new SendGridClient(apiKey);
            var msg = new SendGridMessage()
            {
                From = new EmailAddress("Joe@contoso.com", "Joe Smith"),
                Subject = subject,
                PlainTextContent = message,
                HtmlContent = message
            };
            msg.AddTo(new EmailAddress(email));

            // Disable click tracking.
            // See https://sendgrid.com/docs/User_Guide/Settings/tracking.html
            msg.SetClickTracking(false, false);

            return client.SendEmailAsync(msg);
        }
    }
}
```

Configure startup to support email

Add the following code to the `ConfigureServices` method in the *Startup.cs* file:

- Add `EmailSender` as a transient service.
- Register the `AuthMessageSenderOptions` configuration instance.

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<IdentityUser>(config =>
    {
        config.SignIn.RequireConfirmedEmail = true;
    })
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    // requires
    // using Microsoft.AspNetCore.Identity.UI.Services;
    // using WebPWrecover.Services;
    services.AddTransient<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration);

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Enable account confirmation and password recovery

The template has the code for account confirmation and password recovery. Find the `OnPostAsync` method in *Areas/Identity/Pages/Account/Register.cshtml.cs*.

Prevent newly registered users from being automatically signed in by commenting out the following line:

```
await _signInManager.SignInAsync(user, isPersistent: false);
```

The complete method is shown with the changed line highlighted:

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with password.");

            var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { userId = user.Id, code = code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your email",
                $"Please confirm your account by <a href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking
here</a>.");

            //await _signInManager.SignInAsync(user, isPersistent: false);
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

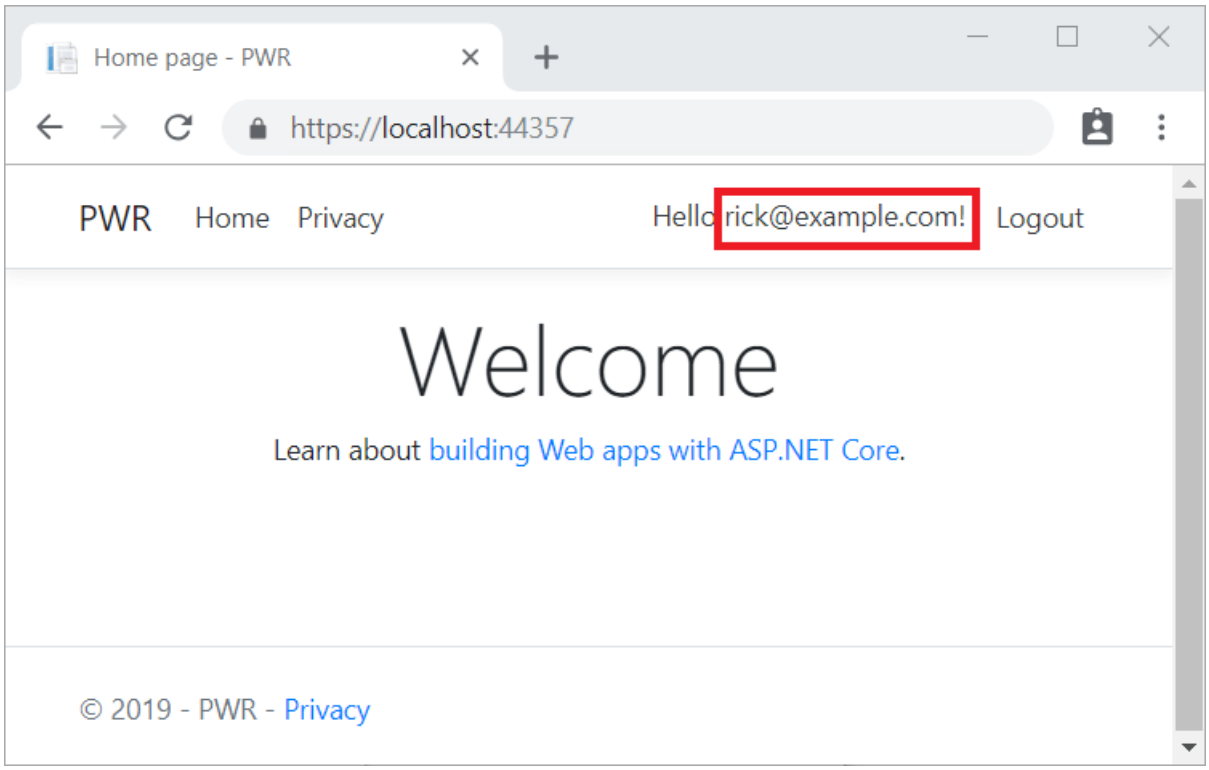
Register, confirm email, and reset password

Run the web app, and test the account confirmation and password recovery flow.

- Run the app and register a new user
- Check your email for the account confirmation link. See Debug email if you don't get the email.
- Click the link to confirm your email.
- Sign in with your email and password.
- Sign out.

View the manage page

Select your user name in the browser:



The manage page is displayed with the **Profile** tab selected. The **Email** shows a check box indicating the email has been confirmed.

Test password reset

- If you're signed in, select **Logout**.
- Select the **Log in** link and select the **Forgot your password?** link.
- Enter the email you used to register the account.
- An email with a link to reset your password is sent. Check your email and click the link to reset your password. After your password has been successfully reset, you can sign in with your email and new password.

Change email and activity timeout

The default inactivity timeout is 14 days. The following code sets the inactivity timeout to 5 days:

```
services.ConfigureApplicationCookie(o => {
    o.ExpireTimeSpan = TimeSpan.FromDays(5);
    o.SlidingExpiration = true;
});
```

Change all data protection token lifespans

The following code changes all data protection tokens timeout period to 3 hours:

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<IdentityUser>(config =>
    {
        config.SignIn.RequireConfirmedEmail = true;
    })
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.Configure<DataProtectionTokenProviderOptions>(o =>
                o.TokenLifespan = TimeSpan.FromHours(3));

    services.AddTransient<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration);

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

The built in Identity user tokens (see AspNetCore/src/Identity/Extensions.Core/src/TokenOptions.cs)have a one day timeout.

Change the email token lifespan

The default token lifespan of the Identity user tokens is one day. This section shows how to change the email token lifespan.

Add a custom DataProtectorTokenProvider<TUser> and DataProtectionTokenProviderOptions:

```
public class CustomEmailConfirmationTokenProvider<TUser>
                                      : DataProtectorTokenProvider<TUser> where TUser : class
{
    public CustomEmailConfirmationTokenProvider(IDataProtectionProvider dataProtectionProvider,
        IOptions<EmailConfirmationTokenProviderOptions> options)
                                                          : base(dataProtectionProvider, options)
    {

    }
}
public class EmailConfirmationTokenProviderOptions : DataProtectionTokenProviderOptions
{
    public EmailConfirmationTokenProviderOptions()
    {
        Name = "EmailDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(4);
    }
}
```

Add the custom provider to the service container:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<IdentityUser>(config =>
    {
        config.SignIn.RequireConfirmedEmail = true;
        config.Tokens.ProviderMap.Add("CustomEmailConfirmation",
            new TokenProviderDescriptor(
                typeof(CustomEmailConfirmationTokenProvider<IdentityUser>)));
        config.Tokens.EmailConfirmationTokenProvider = "CustomEmailConfirmation";
    })
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddTransient<CustomEmailConfirmationTokenProvider<IdentityUser>>();
    services.AddTransient<IEmailSender, EmailSender>();
    services.Configure<AuthMessageSenderOptions>(Configuration); // For SendGrid key.

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Resend email confirmation

See this GitHub issue.

Debug email

If you can't get email working:

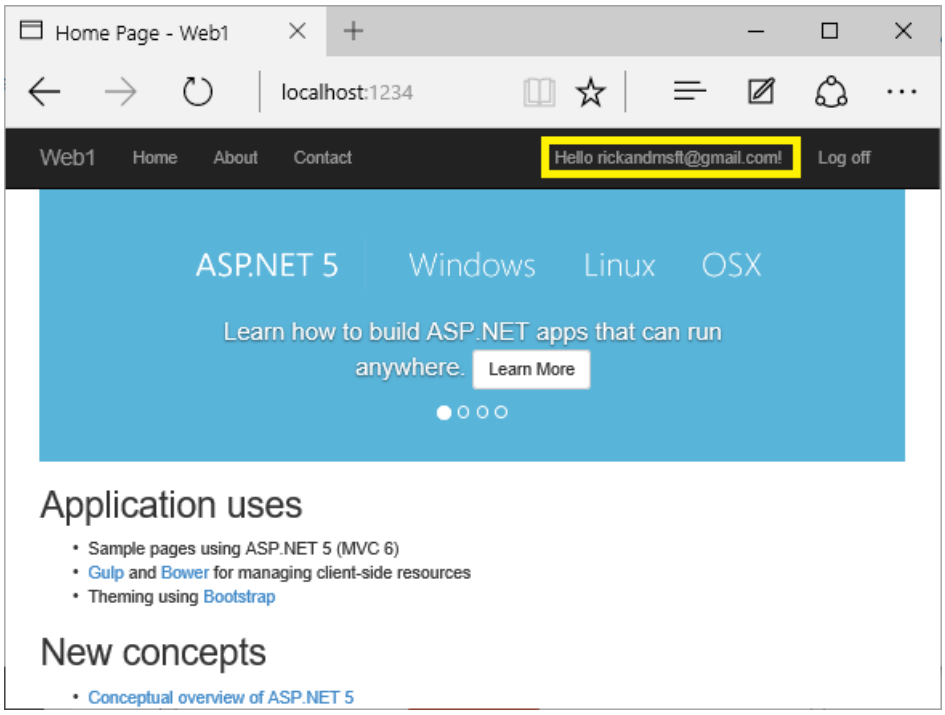
- Set a breakpoint in `EmailSender.Execute` to verify `SendGridClient.SendEmailAsync` is called.
- Create a console app to send email using similar code to `EmailSender.Execute`.
- Review the Email Activity page.
- Check your spam folder.
- Try another email alias on a different email provider (Microsoft, Yahoo, Gmail, etc.)
- Try sending to different email accounts.

A security best practice is to **not** use production secrets in test and development. If you publish the app to Azure, you can set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is set up to read keys from environment variables.

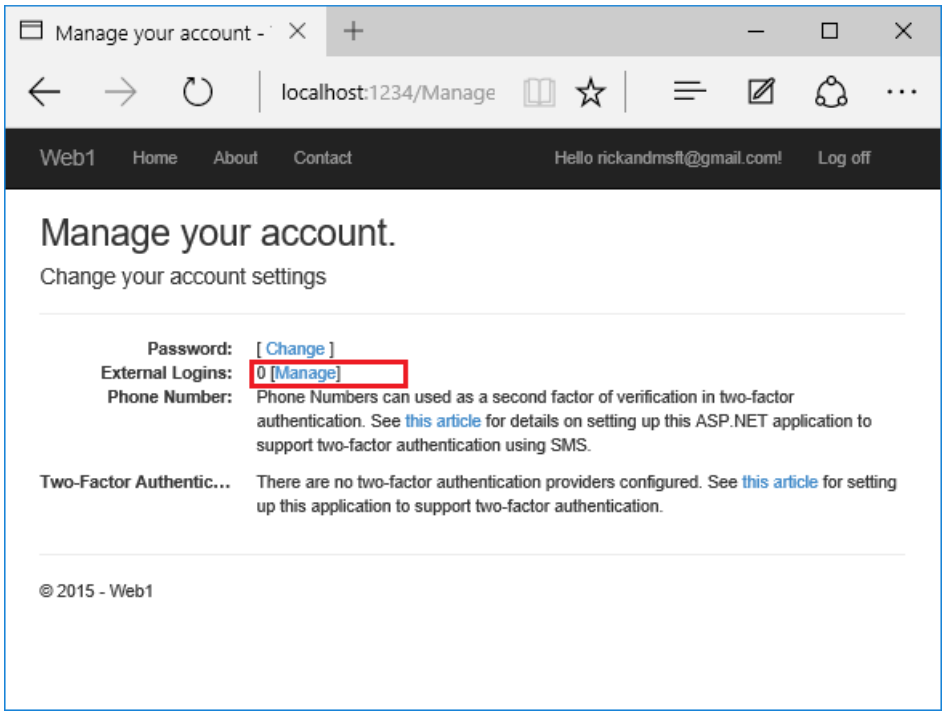
Combine social and local login accounts

To complete this section, you must first enable an external authentication provider. See Facebook, Google, and external provider authentication.

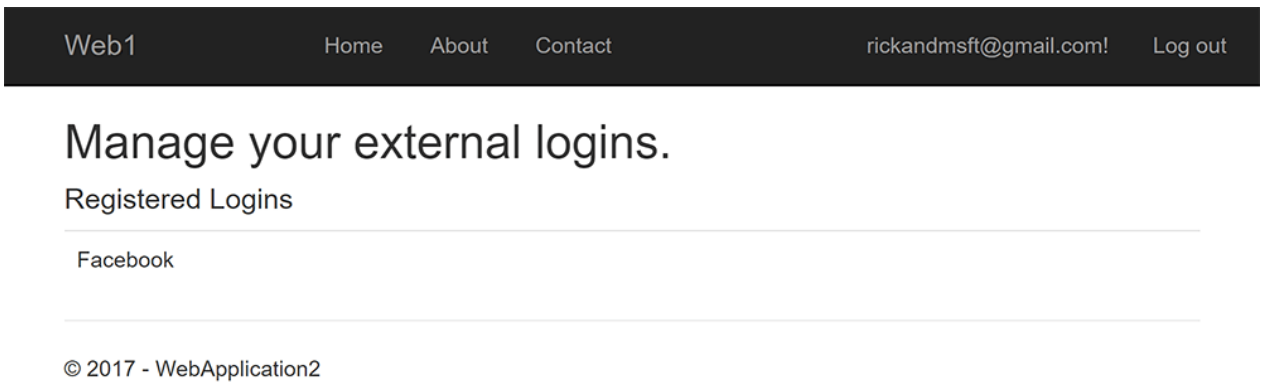
You can combine local and social accounts by clicking on your email link. In the following sequence, "RickAndMSFT@gmail.com" is first created as a local login; however, you can create the account as a social login first, then add a local login.



Click on the **Manage** link. Note the 0 external (social logins) associated with this account.



Click the link to another login service and accept the app requests. In the following image, Facebook is the external authentication provider:



The two accounts have been combined. You are able to sign in with either account. You might want your users to add local accounts in case their social login authentication service is down, or more likely they've lost access to their social account.

Enable account confirmation after a site has users

Enabling account confirmation on a site with users locks out all the existing users. Existing users are locked out because their accounts aren't confirmed. To work around existing user lockout, use one of the following approaches:

- Update the database to mark all existing users as being confirmed.
- Confirm existing users. For example, batch-send emails with confirmation links.

Enable QR Code generation for TOTP authenticator apps in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

QR Codes requires ASP.NET Core 2.0 or later.

ASP.NET Core ships with support for authenticator applications for individual authentication. Two factor authentication (2FA) authenticator apps, using a Time-based One-time Password Algorithm (TOTP), are the industry recommended approach for 2FA. 2FA using TOTP is preferred to SMS 2FA. An authenticator app provides a 6 to 8 digit code which users must enter after confirming their username and password. Typically an authenticator app is installed on a smart phone.

The ASP.NET Core web app templates support authenticators, but don't provide support for QRCode generation. QRCode generators ease the setup of 2FA. This document will guide you through adding QR Code generation to the 2FA configuration page.

Two factor authentication does not happen using an external authentication provider, such as Google or Facebook. External logins are protected by whatever mechanism the external login provider provides. Consider, for example, the Microsoft authentication provider requires a hardware key or another 2FA approach. If the default templates enforced "local" 2FA then users would be required to satisfy two 2FA approaches, which is not a commonly used scenario.

Adding QR Codes to the 2FA configuration page

These instructions use *qrcode.js* from the https://davidshimjs.github.io/qrcodejs/ repo.

- Download the qrcode.js javascript library to the `wwwroot\lib` folder in your project.
- Follow the instructions in Scaffold Identity to generate */Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml*.
- In */Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml*, locate the `Scripts` section at the end of the file:
- In *Pages/Account/Manage/EnableAuthenticator.cshtml* (Razor Pages) or *Views/Manage/EnableAuthenticator.cshtml* (MVC), locate the `Scripts` section at the end of the file:

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

- Update the `Scripts` section to add a reference to the `qrcodejs` library you added and a call to generate the QR Code. It should look as follows:

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")

    <script type="text/javascript" src="~/lib/qrcode.js"></script>
    <script type="text/javascript">
        new QRCode(document.getElementById("qrCode"),
            {
                text: "@Html.Raw(Model.AuthenticatorUri)",
                width: 150,
                height: 150
            });
    </script>
}
```

- Delete the paragraph which links you to these instructions.

Run your app and ensure that you can scan the QR code and validate the code the authenticator proves.

Change the site name in the QR Code

The site name in the QR Code is taken from the project name you choose when initially creating your project. You can change it by looking for the `GenerateQrCodeUri(string email, string unformattedKey)` method in the */Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml.cs*.

The site name in the QR Code is taken from the project name you choose when initially creating your project. You can change it by looking for the `GenerateQrCodeUri(string email, string unformattedKey)` method in the *Pages/Account/Manage/EnableAuthenticator.cshtml.cs* (Razor Pages) file or the *Controllers/ManageController.cs* (MVC) file.

The default code from the template looks as follows:

```
private string GenerateQrCodeUri(string email, string unformattedKey)
{
    return string.Format(
        AuthenticatorUriFormat,
        _urlEncoder.Encode("Razor Pages"),
        _urlEncoder.Encode(email),
        unformattedKey);
}
```

The second parameter in the call to `string.Format` is your site name, taken from your solution name. It can be changed to any value, but it must always be URL encoded.

Using a different QR Code library

You can replace the QR Code library with your preferred library. The HTML contains a `qrCode` element into which you can place a QR Code by whatever mechanism your library provides.

The correctly formatted URL for the QR Code is available in the:

- `AuthenticatorUri` property of the model.
- `data-url` property in the `qrCodeData` element.

TOTP client and server time skew

TOTP (Time-based One-Time Password) authentication depends on both the server and authenticator device having an accurate time. Tokens only last for 30 seconds. If TOTP 2FA logins are failing, check that the server time is accurate, and preferably synchronized to an accurate NTP service.

Two-factor authentication with SMS in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

By Rick Anderson and Swiss-Devs

WARNING

Two factor authentication (2FA) authenticator apps, using a Time-based One-time Password Algorithm (TOTP), are the industry recommended approach for 2FA. 2FA using TOTP is preferred to SMS 2FA. For more information, see Enable QR Code generation for TOTP authenticator apps in ASP.NET Core for ASP.NET Core 2.0 and later.

This tutorial shows how to set up two-factor authentication (2FA) using SMS. Instructions are given for twilio and ASPSMS, but you can use any other SMS provider. We recommend you complete Account Confirmation and Password Recovery before starting this tutorial.

View or download sample code. How to download.

Create a new ASP.NET Core project

Create a new ASP.NET Core web app named `Web2FA` with individual user accounts. Follow the instructions in Enforce HTTPS in ASP.NET Core to set up and require HTTPS.

Create an SMS account

Create an SMS account, for example, from twilio or ASPSMS. Record the authentication credentials (for twilio: accountSid and authToken, for ASPSMS: Userkey and Password).

Figuring out SMS Provider credentials

Twilio:

From the Dashboard tab of your Twilio account, copy the **Account SID** and **Auth token**.

ASPSMS:

From your account settings, navigate to **Userkey** and copy it together with your **Password**.

We will later store these values in with the secret-manager tool within the keys `SMSAccountIdentification` and `SMSAccountPassword`.

Specifying SenderID / Originator

Twilio: From the Numbers tab, copy your Twilio **phone number**.

ASPSMS: Within the Unlock Originators Menu, unlock one or more Originators or choose an alphanumeric Originator (Not supported by all networks).

We will later store this value with the secret-manager tool within the key `SMSAccountFrom`.

Provide credentials for the SMS service

We'll use the Options pattern to access the user account and key settings.

- Create a class to fetch the secure SMS key. For this sample, the `SMSoptions` class is created in the *Services/SMSoptions.cs* file.

```
namespace Web2FA.Services
{
    public class SMSoptions
    {
        public string SMSAccountIdentification { get; set; }
        public string SMSAccountPassword { get; set; }
        public string SMSAccountFrom { get; set; }
    }
}
```

Set the `SMSAccountIdentification`, `SMSAccountPassword` and `SMSAccountFrom` with the secret-manager tool. For example:

```
C:/Web2FA/src/WebApp1>dotnet user-secrets set SMSAccountIdentification 12345
info: Successfully saved SMSAccountIdentification = 12345 to the secret store.
```

- Add the NuGet package for the SMS provider. From the Package Manager Console (PMC) run:

Twilio:

```
Install-Package Twilio
```

ASPSMS:

```
Install-Package ASPSMS
```

- Add code in the *Services/MessageServices.cs* file to enable SMS. Use either the Twilio or the ASPSMS section:

Twilio:

```
using Microsoft.Extensions.Options;
using System.Threading.Tasks;
using Twilio;
using Twilio.Rest.Api.V2010.Account;
using Twilio.Types;

namespace Web2FA.Services
{
    // This class is used by the application to send Email and SMS
    // when you turn on two-factor authentication in ASP.NET Identity.
    // For more details see this link https://go.microsoft.com/fwlink/?LinkID=532713
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public AuthMessageSender(IOptions<SMSoptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public SMSoptions Options { get; }  // set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            // Plug in your email service here to send an email.
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            // Plug in your SMS service here to send a text message.
            // Your Account SID from twilio.com/console
            var accountSid = Options.SMSAccountIdentification;
            // Your Auth Token from twilio.com/console
            var authToken = Options.SMSAccountPassword;

            TwilioClient.Init(accountSid, authToken);

            return MessageResource.CreateAsync(
              to: new PhoneNumber(number),
              from: new PhoneNumber(Options.SMSAccountFrom),
              body: message);
        }
    }
}
```

ASPSMS:

```
using Microsoft.Extensions.Options;
using System.Threading.Tasks;

namespace Web2FA.Services
{
    // This class is used by the application to send Email and SMS
    // when you turn on two-factor authentication in ASP.NET Identity.
    // For more details see this link https://go.microsoft.com/fwlink/?LinkID=532713
    public class AuthMessageSender : IEmailSender, ISmsSender
    {
        public AuthMessageSender(IOptions<SMSoptions> optionsAccessor)
        {
            Options = optionsAccessor.Value;
        }

        public SMSoptions Options { get; }  // set only via Secret Manager

        public Task SendEmailAsync(string email, string subject, string message)
        {
            // Plug in your email service here to send an email.
            return Task.FromResult(0);
        }

        public Task SendSmsAsync(string number, string message)
        {
            ASPSMS.SMS SMSSender = new ASPSMS.SMS();

            SMSSender.Userkey = Options.SMSAccountIdentification;
            SMSSender.Password = Options.SMSAccountPassword;
            SMSSender.Originator = Options.SMSAccountFrom;

            SMSSender.AddRecipient(number);
            SMSSender.MessageData = message;

            SMSSender.SendTextSMS();

            return Task.FromResult(0);
        }
    }
}
```

Configure startup to use `SMSoptions`

Add `SMSoptions` to the service container in the `ConfigureServices` method in the *Startup.cs*:

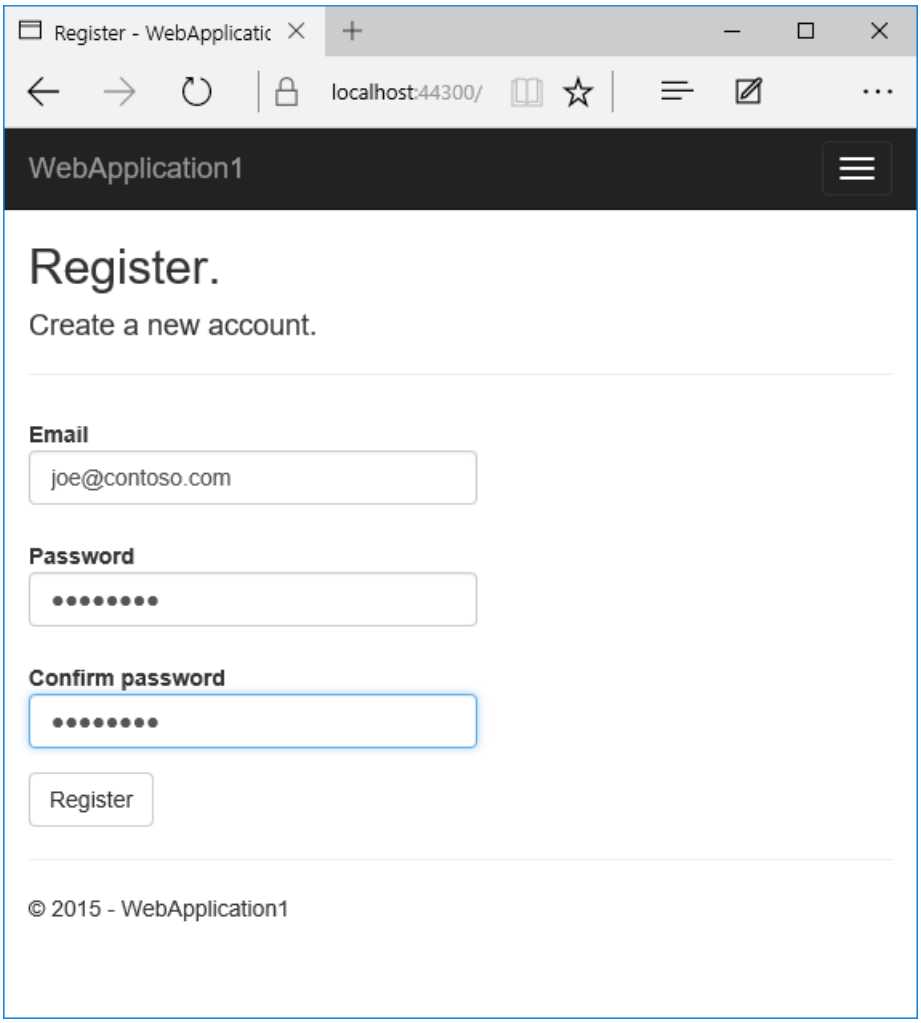
```
    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSoptions>(Configuration);
}
```

Enable two-factor authentication

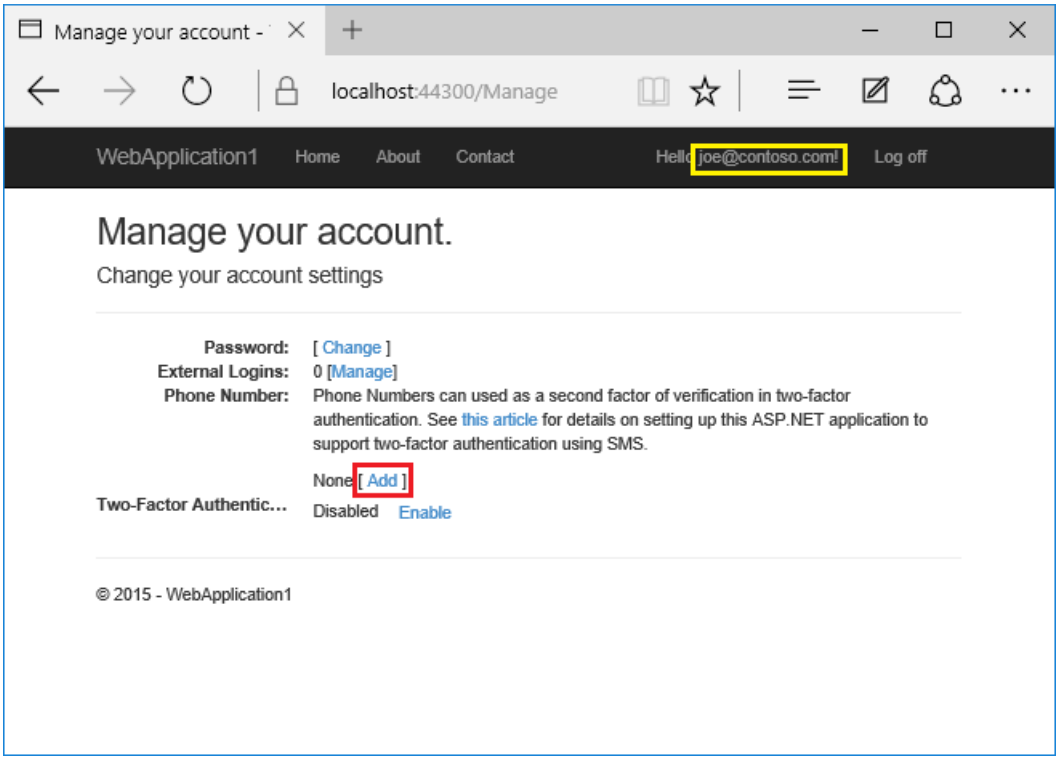
Open the *Views/Manage/Index.cshtml* Razor view file and remove the comment characters (so no markup is commented out).

Log in with two-factor authentication

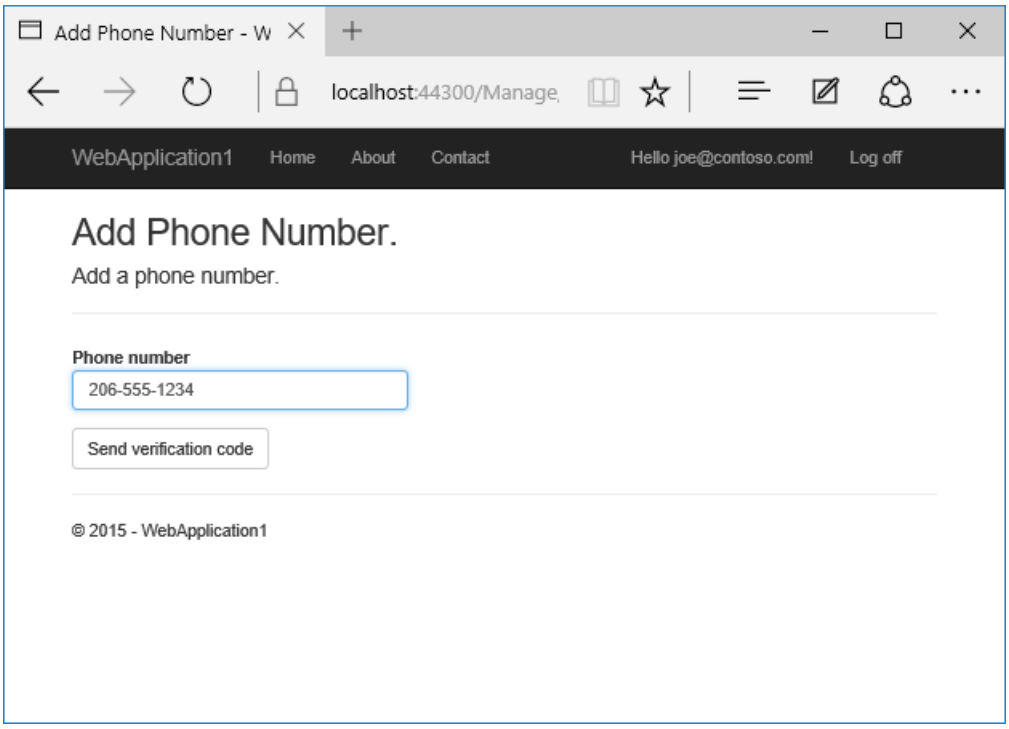
- Run the app and register a new user



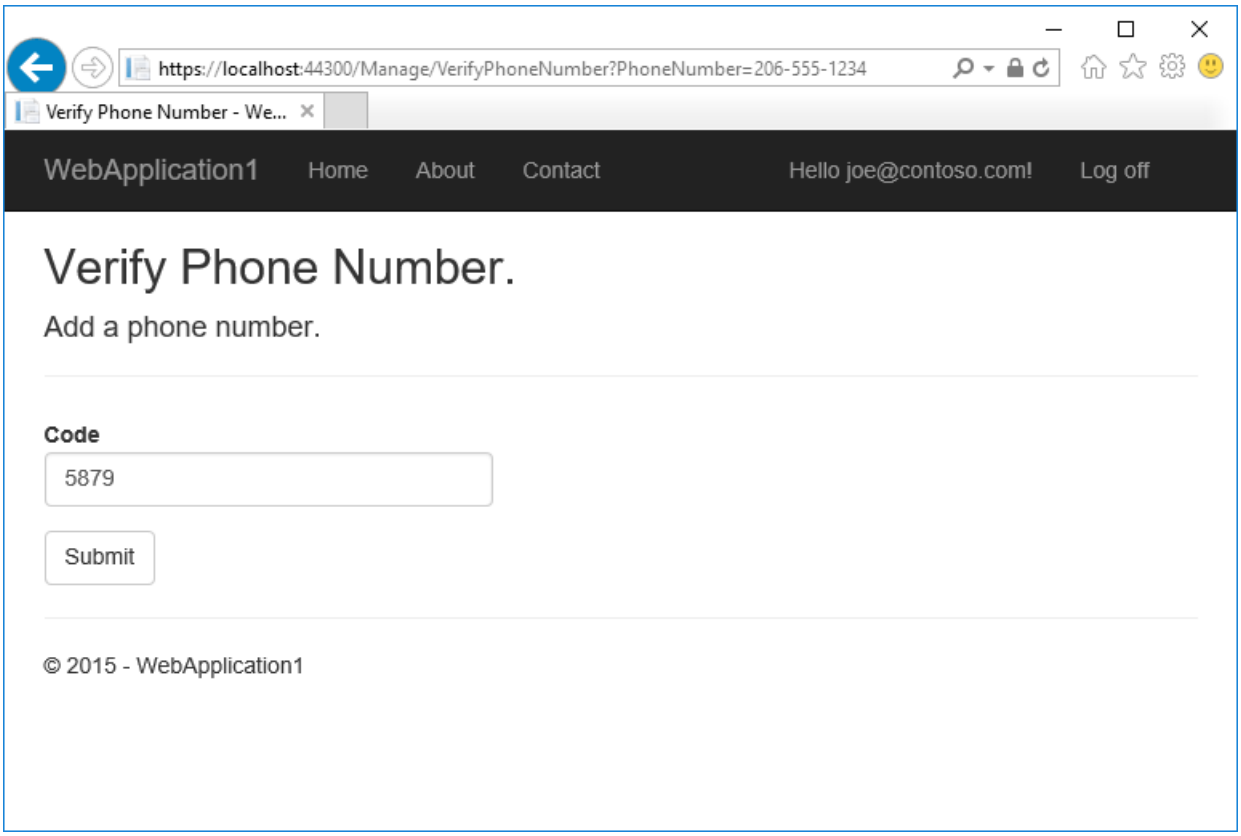
- Tap on your user name, which activates the `Index` action method in Manage controller. Then tap the phone number **Add** link.



- Add a phone number that will receive the verification code, and tap **Send verification code**.

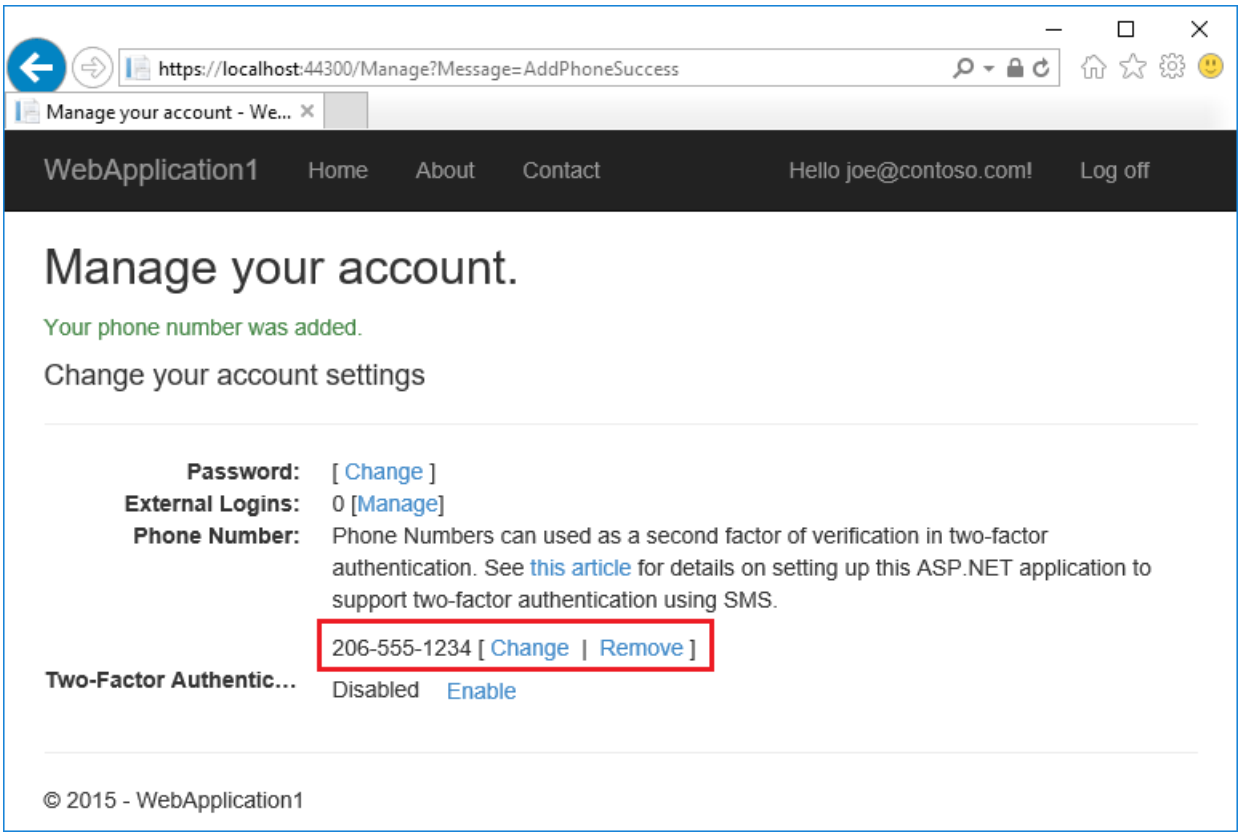


- You will get a text message with the verification code. Enter it and tap **Submit**

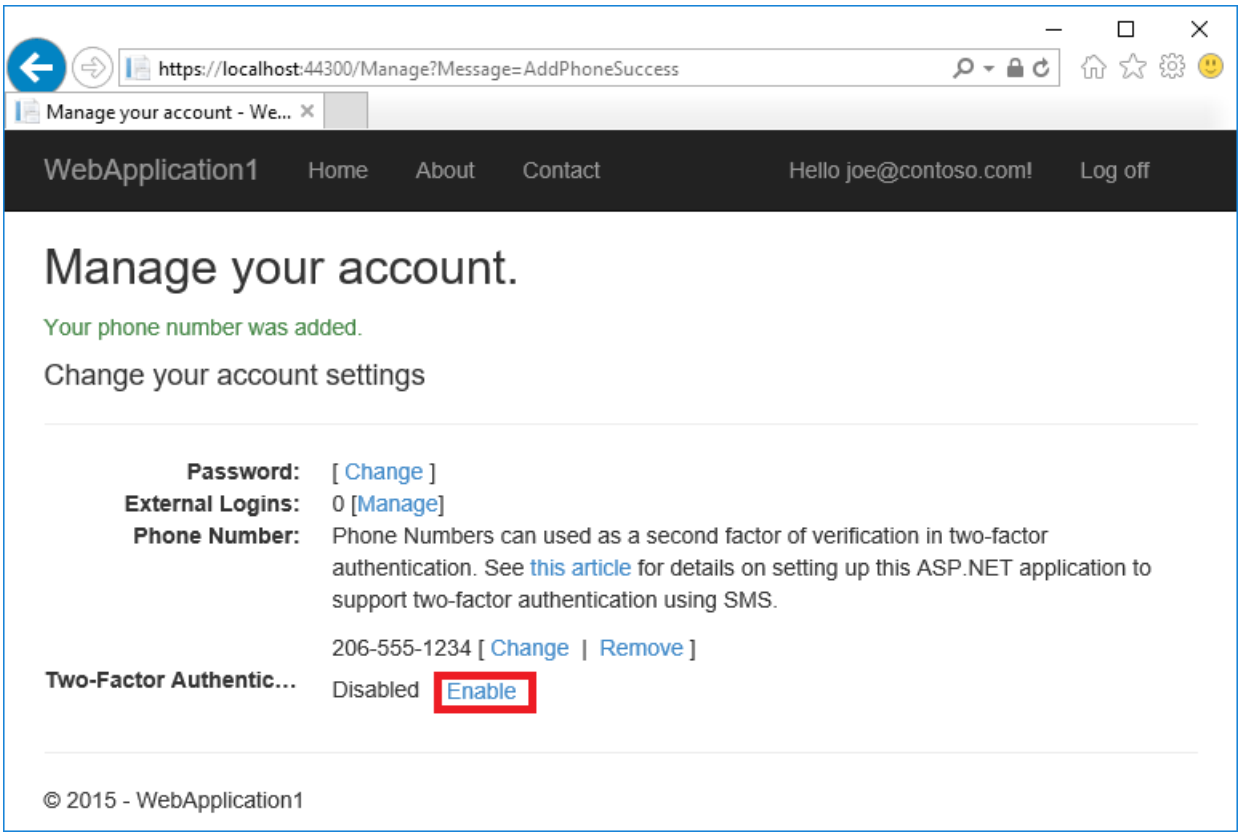


If you don't get a text message, see twilio log page.

- The Manage view shows your phone number was added successfully.



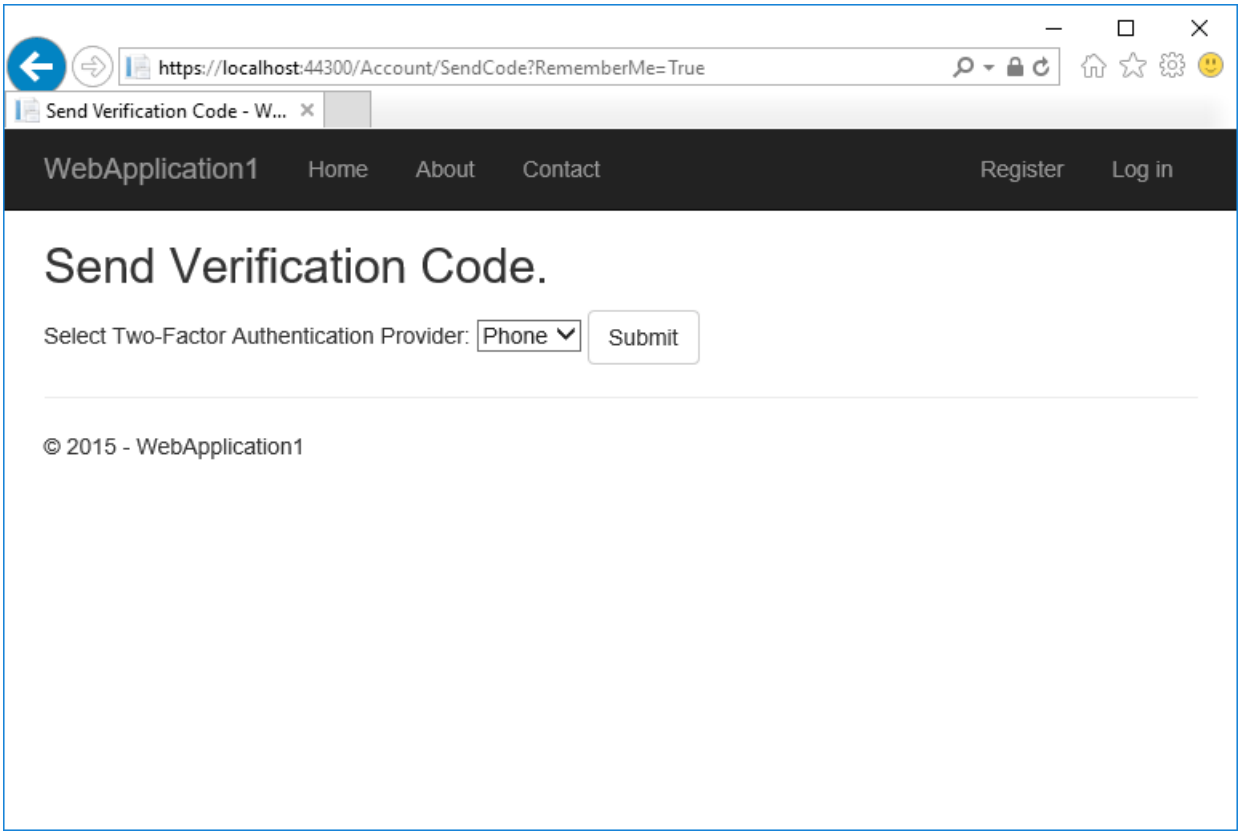
- Tap **Enable** to enable two-factor authentication.



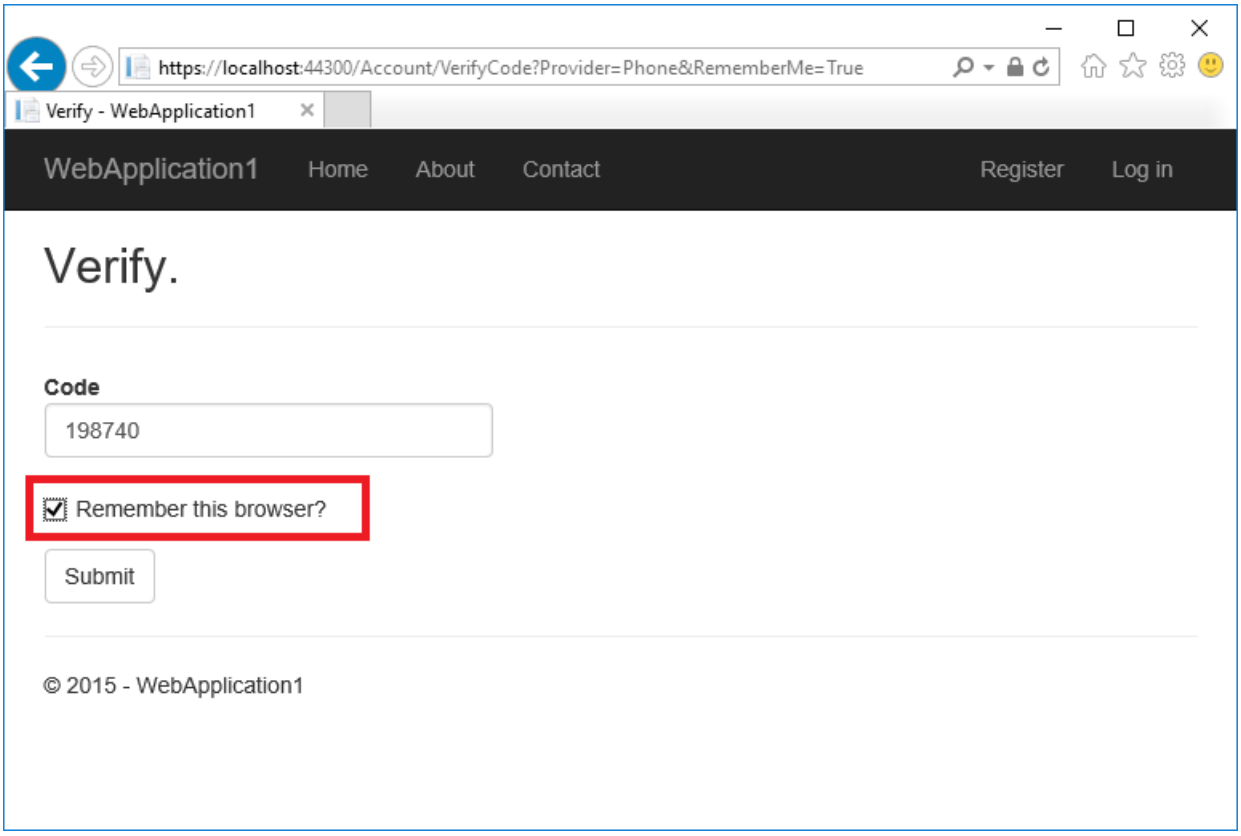
Test two-factor authentication

- Log off.
- Log in.
- The user account has enabled two-factor authentication, so you have to provide the second factor of authentication . In this tutorial you have enabled phone verification. The built in templates also allow you to set up email as the second factor. You can set up additional second factors for authentication such as QR

codes. Tap **Submit**.



- Enter the code you get in the SMS message.
- Clicking on the **Remember this browser** check box will exempt you from needing to use 2FA to log on when using the same device and browser. Enabling 2FA and clicking on **Remember this browser** will provide you with strong 2FA protection from malicious users trying to access your account, as long as they don't have access to your device. You can do this on any private device you regularly use. By setting **Remember this browser**, you get the added security of 2FA from devices you don't regularly use, and you get the convenience on not having to go through 2FA on your own devices.



Account lockout for protecting against brute force attacks

Account lockout is recommended with 2FA. Once a user signs in through a local account or social account, each failed attempt at 2FA is stored. If the maximum failed access attempts is reached, the user is locked out (default: 5 minute lockout after 5 failed access attempts). A successful authentication resets the failed access attempts count and resets the clock. The maximum failed access attempts and lockout time can be set with MaxFailedAccessAttempts and DefaultLockoutTimeSpan. The following configures account lockout for 10 minutes after 10 failed access attempts:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    services.Configure<IdentityOptions>(options =>
    {
        options.Lockout.MaxFailedAccessAttempts = 10;
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(10);
    });

    // Add application services.
    services.AddTransient<IEmailSender, AuthMessageSender>();
    services.AddTransient<ISmsSender, AuthMessageSender>();
    services.Configure<SMSoptions>(Configuration);
}
```

Confirm that PasswordSignInAsync sets `lockoutOnFailure` to `true`:

```
var result = await _signInManager.PasswordSignInAsync(
                Input.Email, Input.Password, Input.RememberMe, lockoutOnFailure: true);
```

Use cookie authentication without ASP.NET Core Identity

9/22/2020 • 14 minutes to read • Edit Online

By Rick Anderson

ASP.NET Core Identity is a complete, full-featured authentication provider for creating and maintaining logins. However, a cookie-based authentication provider without ASP.NET Core Identity can be used. For more information, see Introduction to Identity on ASP.NET Core.

View or download sample code (how to download)

For demonstration purposes in the sample app, the user account for the hypothetical user, Maria Rodriguez, is hardcoded into the app. Use the **Email** address `maria.rodriguez@contoso.com` and any password to sign in the user. The user is authenticated in the `AuthenticateUser` method in the *Pages/Account/Login.cshtml.cs* file. In a real-world example, the user would be authenticated against a database.

Configuration

If the app doesn't use the Microsoft.AspNetCore.App metapackage, create a package reference in the project file for the Microsoft.AspNetCore.Authentication.Cookies package.

In the `Startup.ConfigureServices` method, create the Authentication Middleware services with the AddAuthentication and AddCookie methods:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

AuthenticationScheme passed to `AddAuthentication` sets the default authentication scheme for the app. `AuthenticationScheme` is useful when there are multiple instances of cookie authentication and you want to authorize with a specific scheme. Setting the `AuthenticationScheme` to CookieAuthenticationDefaults.AuthenticationScheme provides a value of "Cookies" for the scheme. You can supply any string value that distinguishes the scheme.

The app's authentication scheme is different from the app's cookie authentication scheme. When a cookie authentication scheme isn't provided to AddCookie, it uses `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookies").

The authentication cookie's IsEssential property is set to `true` by default. Authentication cookies are allowed when a site visitor hasn't consented to data collection. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

In `Startup.Configure`, call `UseAuthentication` and `UseAuthorization` to set the `HttpContext.User` property and run Authorization Middleware for requests. Call the `UseAuthentication` and `UseAuthorization` methods before calling `UseEndpoints`:

```
app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapRazorPages();
});
```

The CookieAuthenticationOptions class is used to configure the authentication provider options.

Set `CookieAuthenticationOptions` in the service configuration for authentication in the `Startup.ConfigureServices` method:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        ...
    });
```

Cookie Policy Middleware

Cookie Policy Middleware enables cookie policy capabilities. Adding the middleware to the app processing pipeline is order sensitive—it only affects downstream components registered in the pipeline.

```
app.UseCookiePolicy(cookiePolicyOptions);
```

Use CookiePolicyOptions provided to the Cookie Policy Middleware to control global characteristics of cookie processing and hook into cookie processing handlers when cookies are appended or deleted.

The default MinimumSameSitePolicy value is `SameSiteMode.Lax` to permit OAuth2 authentication. To strictly enforce a same-site policy of `SameSiteMode.Strict`, set the `MinimumSameSitePolicy`. Although this setting breaks OAuth2 and other cross-origin authentication schemes, it elevates the level of cookie security for other types of apps that don't rely on cross-origin request processing.

```
var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

The Cookie Policy Middleware setting for `MinimumSameSitePolicy` can affect the setting of `Cookie.SameSite` in `CookieAuthenticationOptions` settings according to the matrix below.

MINIMUMSAMESITEPOLICY	COOKIE.SAMESITE	RESULTANT COOKIE.SAMESITE SETTING
SameSiteMode.None	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Lax SameSiteMode.Lax SameSiteMode.Strict

MINIMUMSAMESITEPOLICY	COOKIE.SAMESITE	RESULTANT COOKIE.SAMESITE SETTING
SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Strict SameSiteMode.Strict SameSiteMode.Strict

Create an authentication cookie

To create a cookie holding user information, construct a ClaimsPrincipal. The user information is serialized and stored in the cookie.

Create a ClaimsIdentity with any required Claims and call SignInAsync to sign in the user:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("FullName", user.FullName),
    new Claim(ClaimTypes.Role, "Administrator"),
};

var claimsIdentity = new ClaimsIdentity(
    claims, CookieAuthenticationDefaults.AuthenticationScheme);

var authProperties = new AuthenticationProperties
{
    //AllowRefresh = <bool>,
    // Refreshing the authentication session should be allowed.

    //ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
    // The time at which the authentication ticket expires. A
    // value set here overrides the ExpireTimeSpan option of
    // CookieAuthenticationOptions set with AddCookie.

    //IsPersistent = true,
    // Whether the authentication session is persisted across
    // multiple requests. When used with cookies, controls
    // whether the cookie's lifetime is absolute (matching the
    // lifetime of the authentication ticket) or session-based.

    //IssuedUtc = <DateTimeOffset>,
    // The time at which the authentication ticket was issued.

    //RedirectUri = <string>
    // The full path or absolute URI to be used as an http
    // redirect response value.
};

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    authProperties);
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

`SignInAsync` creates an encrypted cookie and adds it to the current response. If `AuthenticationScheme` isn't specified, the default scheme is used.

RedirectUri is only used on a few specific paths by default, for example, the login path and logout paths. For more information see the CookieAuthenticationHandler source.

ASP.NET Core's Data Protection system is used for encryption. For an app hosted on multiple machines, load balancing across apps, or using a web farm, configure data protection to use the same key ring and app identifier.

Sign out

To sign out the current user and delete their cookie, call SignOutAsync:

```
await HttpContext.SignOutAsync(
    CookieAuthenticationDefaults.AuthenticationScheme);
```

If `CookieAuthenticationDefaults.AuthenticationScheme` (or "Cookies") isn't used as the scheme (for example, "ContosoCookie"), supply the scheme used when configuring the authentication provider. Otherwise, the default scheme is used.

When the browser closes it automatically deletes session based cookies (non-persistent cookies), but no cookies are cleared when an individual tab is closed. The server is not notified of tab or browser close events.

React to back-end changes

Once a cookie is created, the cookie is the single source of identity. If a user account is disabled in back-end systems:

- The app's cookie authentication system continues to process requests based on the authentication cookie.
- The user remains signed into the app as long as the authentication cookie is valid.

The ValidatePrincipal event can be used to intercept and override validation of the cookie identity. Validating the cookie on every request mitigates the risk of revoked users accessing the app.

One approach to cookie validation is based on keeping track of when the user database changes. If the database hasn't been changed since the user's cookie was issued, there's no need to re-authenticate the user if their cookie is still valid. In the sample app, the database is implemented in `IUserRepository` and stores a `LastChanged` value. When a user is updated in the database, the `LastChanged` value is set to the current time.

In order to invalidate a cookie when the database changes based on the `LastChanged` value, create the cookie with a `LastChanged` claim containing the current `LastChanged` value from the database:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};

var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

To implement an override for the `ValidatePrincipal` event, write a method with the following signature in a class that derives from CookieAuthenticationEvents:

```
ValidatePrincipal(CookieValidatePrincipalContext)
```

The following is an example implementation of `CookieAuthenticationEvents`:

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents
{
    private readonly IUserRepository _userRepository;

    public CustomCookieAuthenticationEvents(IUserRepository userRepository)
    {
        // Get the database from registered DI services.
        _userRepository = userRepository;
    }

    public override async Task ValidatePrincipal(CookieValidatePrincipalContext context)
    {
        var userPrincipal = context.Principal;

        // Look for the LastChanged claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !_userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}
```

Register the events instance during cookie service registration in the `Startup.ConfigureServices` method. Provide a scoped service registration for your `CustomCookieAuthenticationEvents` class:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.EventsType = typeof(CustomCookieAuthenticationEvents);
    });

services.AddScoped<CustomCookieAuthenticationEvents>();
```

Consider a situation in which the user's name is updated—a decision that doesn't affect security in any way. If you want to non-destructively update the user principal, call `context.ReplacePrincipal` and set the `context.ShouldRenew` property to `true`.

> **WARNING**
>
> The approach described here is triggered on every request. Validating authentication cookies for all users on every request can result in a large performance penalty for the app.

Persistent cookies

You may want the cookie to persist across browser sessions. This persistence should only be enabled with explicit user consent with a "Remember Me" check box on sign in or a similar mechanism.

The following code snippet creates an identity and corresponding cookie that survives through browser closures. Any sliding expiration settings previously configured are honored. If the cookie expires while the browser is closed, the browser clears the cookie once it's restarted.

Set IsPersistent to `true` in AuthenticationProperties:

```
// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

Absolute cookie expiration

An absolute expiration time can be set with ExpiresUtc. To create a persistent cookie, `IsPersistent` must also be set. Otherwise, the cookie is created with a session-based lifetime and could expire either before or after the authentication ticket that it holds. When `ExpiresUtc` is set, it overrides the value of the ExpireTimeSpan option of CookieAuthenticationOptions, if set.

The following code snippet creates an identity and corresponding cookie that lasts for 20 minutes. This ignores any sliding expiration settings previously configured.

```
// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

ASP.NET Core Identity is a complete, full-featured authentication provider for creating and maintaining logins. However, a cookie-based authentication provider without ASP.NET Core Identity can be used. For more information, see Introduction to Identity on ASP.NET Core.

View or download sample code (how to download)

For demonstration purposes in the sample app, the user account for the hypothetical user, Maria Rodriguez, is hardcoded into the app. Use the **Email** address `maria.rodriguez@contoso.com` and any password to sign in the user. The user is authenticated in the `AuthenticateUser` method in the *Pages/Account/Login.cshtml.cs* file. In a real-world example, the user would be authenticated against a database.

Configuration

If the app doesn't use the Microsoft.AspNetCore.App metapackage, create a package reference in the project file for the Microsoft.AspNetCore.Authentication.Cookies package.

In the `Startup.ConfigureServices` method, create the Authentication Middleware service with the AddAuthentication and AddCookie methods:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

AuthenticationScheme passed to `AddAuthentication` sets the default authentication scheme for the app. `AuthenticationScheme` is useful when there are multiple instances of cookie authentication and you want to authorize with a specific scheme. Setting the `AuthenticationScheme` to CookieAuthenticationDefaults.AuthenticationScheme provides a value of "Cookies" for the scheme. You can supply any string value that distinguishes the scheme.

The app's authentication scheme is different from the app's cookie authentication scheme. When a cookie authentication scheme isn't provided to AddCookie, it uses `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookies").

The authentication cookie's IsEssential property is set to `true` by default. Authentication cookies are allowed when a site visitor hasn't consented to data collection. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

In the `Startup.Configure` method, call the `UseAuthentication` method to invoke the Authentication Middleware that sets the `HttpContext.User` property. Call the `UseAuthentication` method before calling `UseMvcWithDefaultRoute` or `UseMvc`:

```
app.UseAuthentication();
```

The CookieAuthenticationOptions class is used to configure the authentication provider options.

Set `CookieAuthenticationOptions` in the service configuration for authentication in the `Startup.ConfigureServices` method:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        ...
    });
```

Cookie Policy Middleware

Cookie Policy Middleware enables cookie policy capabilities. Adding the middleware to the app processing pipeline is order sensitive—it only affects downstream components registered in the pipeline.

```
app.UseCookiePolicy(cookiePolicyOptions);
```

Use CookiePolicyOptions provided to the Cookie Policy Middleware to control global characteristics of cookie processing and hook into cookie processing handlers when cookies are appended or deleted.

The default MinimumSameSitePolicy value is `SameSiteMode.Lax` to permit OAuth2 authentication. To strictly enforce a same-site policy of `SameSiteMode.Strict`, set the `MinimumSameSitePolicy`. Although this setting breaks OAuth2 and other cross-origin authentication schemes, it elevates the level of cookie security for other types of apps that don't rely on cross-origin request processing.

```
var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

The Cookie Policy Middleware setting for `MinimumSameSitePolicy` can affect the setting of `Cookie.SameSite` in `CookieAuthenticationOptions` settings according to the matrix below.

MINIMUMSAMESITEPOLICY	COOKIE.SAMESITE	RESULTANT COOKIE.SAMESITE SETTING
SameSiteMode.None	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Lax SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Strict SameSiteMode.Strict SameSiteMode.Strict

Create an authentication cookie

To create a cookie holding user information, construct a ClaimsPrincipal. The user information is serialized and stored in the cookie.

Create a ClaimsIdentity with any required Claims and call SignInAsync to sign in the user:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("FullName", user.FullName),
    new Claim(ClaimTypes.Role, "Administrator"),
};

var claimsIdentity = new ClaimsIdentity(
    claims, CookieAuthenticationDefaults.AuthenticationScheme);

var authProperties = new AuthenticationProperties
{
    //AllowRefresh = <bool>,
    // Refreshing the authentication session should be allowed.

    //ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
    // The time at which the authentication ticket expires. A
    // value set here overrides the ExpireTimeSpan option of
    // CookieAuthenticationOptions set with AddCookie.

    //IsPersistent = true,
    // Whether the authentication session is persisted across
    // multiple requests. When used with cookies, controls
    // whether the cookie's lifetime is absolute (matching the
    // lifetime of the authentication ticket) or session-based.

    //IssuedUtc = <DateTimeOffset>,
    // The time at which the authentication ticket was issued.

    //RedirectUri = <string>
    // The full path or absolute URI to be used as an http
    // redirect response value.
};

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    authProperties);
```

`SignInAsync` creates an encrypted cookie and adds it to the current response. If `AuthenticationScheme` isn't specified, the default scheme is used.

ASP.NET Core's Data Protection system is used for encryption. For an app hosted on multiple machines, load balancing across apps, or using a web farm, configure data protection to use the same key ring and app identifier.

Sign out

To sign out the current user and delete their cookie, call SignOutAsync:

```
await HttpContext.SignOutAsync(
    CookieAuthenticationDefaults.AuthenticationScheme);
```

If `CookieAuthenticationDefaults.AuthenticationScheme` (or "Cookies") isn't used as the scheme (for example, "ContosoCookie"), supply the scheme used when configuring the authentication provider. Otherwise, the default scheme is used.

React to back-end changes

Once a cookie is created, the cookie is the single source of identity. If a user account is disabled in back-end systems:

- The app's cookie authentication system continues to process requests based on the authentication cookie.
- The user remains signed into the app as long as the authentication cookie is valid.

The ValidatePrincipal event can be used to intercept and override validation of the cookie identity. Validating the cookie on every request mitigates the risk of revoked users accessing the app.

One approach to cookie validation is based on keeping track of when the user database changes. If the database hasn't been changed since the user's cookie was issued, there's no need to re-authenticate the user if their cookie is still valid. In the sample app, the database is implemented in `IUserRepository` and stores a `LastChanged` value. When a user is updated in the database, the `LastChanged` value is set to the current time.

In order to invalidate a cookie when the database changes based on the `LastChanged` value, create the cookie with a `LastChanged` claim containing the current `LastChanged` value from the database:

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};

var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

To implement an override for the `ValidatePrincipal` event, write a method with the following signature in a class that derives from CookieAuthenticationEvents:

```
ValidatePrincipal(CookieValidatePrincipalContext)
```

The following is an example implementation of `CookieAuthenticationEvents`:

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents
{
    private readonly IUserRepository _userRepository;

    public CustomCookieAuthenticationEvents(IUserRepository userRepository)
    {
        // Get the database from registered DI services.
        _userRepository = userRepository;
    }

    public override async Task ValidatePrincipal(CookieValidatePrincipalContext context)
    {
        var userPrincipal = context.Principal;

        // Look for the LastChanged claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !_userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}
```

Register the events instance during cookie service registration in the `Startup.ConfigureServices` method. Provide a scoped service registration for your `CustomCookieAuthenticationEvents` class:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.EventsType = typeof(CustomCookieAuthenticationEvents);
    });

services.AddScoped<CustomCookieAuthenticationEvents>();
```

Consider a situation in which the user's name is updated—a decision that doesn't affect security in any way. If you want to non-destructively update the user principal, call `context.ReplacePrincipal` and set the `context.ShouldRenew` property to `true`.

> **WARNING**
>
> The approach described here is triggered on every request. Validating authentication cookies for all users on every request can result in a large performance penalty for the app.

Persistent cookies

You may want the cookie to persist across browser sessions. This persistence should only be enabled with explicit user consent with a "Remember Me" check box on sign in or a similar mechanism.

The following code snippet creates an identity and corresponding cookie that survives through browser closures. Any sliding expiration settings previously configured are honored. If the cookie expires while the browser is closed, the browser clears the cookie once it's restarted.

Set IsPersistent to `true` in AuthenticationProperties:

```
// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

Absolute cookie expiration

An absolute expiration time can be set with ExpiresUtc. To create a persistent cookie, `IsPersistent` must also be set. Otherwise, the cookie is created with a session-based lifetime and could expire either before or after the authentication ticket that it holds. When `ExpiresUtc` is set, it overrides the value of the ExpireTimeSpan option of CookieAuthenticationOptions, if set.

The following code snippet creates an identity and corresponding cookie that lasts for 20 minutes. This ignores any sliding expiration settings previously configured.

```
// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

Additional resources

- Authorize with a specific scheme in ASP.NET Core
- Claims-based authorization in ASP.NET Core
- Policy-based role checks
- Host ASP.NET Core in a web farm

Use social sign-in provider authentication without ASP.NET Core Identity

9/22/2020 • 4 minutes to read • Edit Online

By Kirk Larkin and Rick Anderson

Facebook, Google, and external provider authentication in ASP.NET Core describes how to enable users to sign in using OAuth 2.0 with credentials from external authentication providers. The approach described in that topic includes ASP.NET Core Identity as an authentication provider.

This sample demonstrates how to use an external authentication provider **without** ASP.NET Core Identity. This is useful for apps that don't require all of the features of ASP.NET Core Identity, but still require integration with a trusted external authentication provider.

This sample uses Google authentication for authenticating users. Using Google authentication shifts many of the complexities of managing the sign-in process to Google. To integrate with a different external authentication provider, see the following topics:

- Facebook authentication
- Microsoft authentication
- Twitter authentication
- Other providers

Configuration

In the `ConfigureServices` method, configure the app's authentication schemes with the AddAuthentication, AddCookie, and AddGoogle methods:

```
public void ConfigureServices(IServiceCollection services)
{
    // requires
    // using Microsoft.AspNetCore.Authentication.Cookies;
    // using Microsoft.AspNetCore.Authentication.Google;
    // NuGet package Microsoft.AspNetCore.Authentication.Google
    services
        .AddAuthentication(options =>
        {
            options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme = GoogleDefaults.AuthenticationScheme;
        })
        .AddCookie()
        .AddGoogle(options =>
        {
            options.ClientId = Configuration["Authentication:Google:ClientId"];
            options.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
        });

    services.AddRazorPages();
}
```

The call to AddAuthentication sets the app's DefaultScheme. The `DefaultScheme` is the default scheme used by the following `HttpContext` authentication extension methods:

- AuthenticateAsync

- ChallengeAsync
- ForbidAsync
- SignInAsync
- SignOutAsync

Setting the app's `DefaultScheme` to CookieAuthenticationDefaults.AuthenticationScheme ("Cookies") configures the app to use Cookies as the default scheme for these extension methods. Setting the app's DefaultChallengeScheme to GoogleDefaults.AuthenticationScheme ("Google") configures the app to use Google as the default scheme for calls to `ChallengeAsync`. `DefaultChallengeScheme` overrides `DefaultScheme`. See AuthenticationOptions for additional properties that override `DefaultScheme` when set.

In `Startup.Configure`, call `UseAuthentication` and `UseAuthorization` between calling `UseRouting` and `UseEndpoints`. This sets the `HttpContext.User` property and runs the Authorization Middleware for requests:

```
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
});
```

To learn more about authentication schemes, see Authentication Concepts. To learn more about cookie authentication, see Use cookie authentication without ASP.NET Core Identity.

Apply authorization

Test the app's authentication configuration by applying the `AuthorizeAttribute` attribute to a controller, action, or page. The following code limits access to the *Privacy* page to users that have been authenticated:

```
[Authorize]
public class PrivacyModel : PageModel
{

}
```

Sign out

To sign out the current user and delete their cookie, call SignOutAsync. The following code adds a `Logout` page handler to the *Index* page:

```
public class IndexModel : PageModel
{
    public async Task<IActionResult> OnPostLogoutAsync()
    {
        await HttpContext.SignOutAsync();
        return RedirectToPage();
    }
}
```

Notice that the call to `SignOutAsync` does not specify an authentication scheme. The app's `DefaultScheme` of `CookieAuthenticationDefaults.AuthenticationScheme` is used as a fall back.

Additional resources

- Simple authorization in ASP.NET Core
- Persist additional claims and tokens from external providers in ASP.NET Core

Facebook, Google, and external provider authentication in ASP.NET Core describes how to enable users to sign in using OAuth 2.0 with credentials from external authentication providers. The approach described in that topic includes ASP.NET Core Identity as an authentication provider.

This sample demonstrates how to use an external authentication provider **without** ASP.NET Core Identity. This is useful for apps that don't require all of the features of ASP.NET Core Identity, but still require integration with a trusted external authentication provider.

This sample uses Google authentication for authenticating users. Using Google authentication shifts many of the complexities of managing the sign-in process to Google. To integrate with a different external authentication provider, see the following topics:

- Facebook authentication
- Microsoft authentication
- Twitter authentication
- Other providers

Configuration

In the `ConfigureServices` method, configure the app's authentication schemes with the `AddAuthentication`, `AddCookie`, and `AddGoogle` methods:

```
services
    .AddAuthentication(options =>
    {
        options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme = GoogleDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddGoogle(options =>
    {
        options.ClientId = Configuration["Authentication:Google:ClientId"];
        options.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
    });
```

The call to AddAuthentication sets the app's DefaultScheme. The `DefaultScheme` is the default scheme used by the following `HttpContext` authentication extension methods:

- AuthenticateAsync
- ChallengeAsync
- ForbidAsync
- SignInAsync
- SignOutAsync

Setting the app's `DefaultScheme` to CookieAuthenticationDefaults.AuthenticationScheme ("Cookies") configures the app to use Cookies as the default scheme for these extension methods. Setting the app's DefaultChallengeScheme to GoogleDefaults.AuthenticationScheme ("Google") configures the app to use Google as the default scheme for calls to `ChallengeAsync`. `DefaultChallengeScheme` overrides `DefaultScheme`. See AuthenticationOptions for additional properties that override `DefaultScheme` when set.

In the `Configure` method, call the `UseAuthentication` method to invoke the Authentication Middleware that sets

the `HttpContext.User` property. Call the `UseAuthentication` method before calling `UseMvcWithDefaultRoute` or `UseMvc`:

```
app.UseAuthentication();
```

To learn more about authentication schemes, see Authentication Concepts. To learn more about cookie authentication, see Use cookie authentication without ASP.NET Core Identity.

Apply authorization

Test the app's authentication configuration by applying the `AuthorizeAttribute` attribute to a controller, action, or page. The following code limits access to the *Privacy* page to users that have been authenticated:

```
[Authorize]
public class PrivacyModel : PageModel
{

}
```

Sign out

To sign out the current user and delete their cookie, call SignOutAsync. The following code adds a `Logout` page handler to the *Index* page:

```
public class IndexModel : PageModel
{
    public async Task<IActionResult> OnPostLogoutAsync()
    {
        await HttpContext.SignOutAsync();
        return RedirectToPage();
    }
}
```

Notice that the call to `SignOutAsync` does not specify an authentication scheme. The app's `DefaultScheme` of `CookieAuthenticationDefaults.AuthenticationScheme` is used as a fall back.

Additional resources

- Simple authorization in ASP.NET Core
- Persist additional claims and tokens from external providers in ASP.NET Core

Azure Active Directory with ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

These tutorials and samples demonstrate authentication in ASP.NET Core using Microsoft identity platform and Azure Active Directory. For additional tutorials and samples using ASP.NET Core with Azure AD, see Microsoft identity platform.

Application Scenarios

- Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app
- Web app that signs in users
- Web app that calls web APIs
- Protected web API
- Web API that calls other web APIs
- Web app that signs in users with Azure AD B2C

Samples

- Enable your ASP.NET Core app to sign-in users and call web APIs using Azure AD V2:
  - See this associated video
- Calling an ASP.NET Core 2.0 Web API from a WPF application using Azure AD V2:
  - See this associated video
- An ASP.NET Core web API with Azure AD B2C

Cloud authentication with Azure Active Directory B2C in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

By Cam Soper

Azure Active Directory B2C (Azure AD B2C) is a cloud identity management solution for web and mobile apps. The service provides authentication for apps hosted in the cloud and on-premises. Authentication types include individual accounts, social network accounts, and federated enterprise accounts. Additionally, Azure AD B2C can provide multi-factor authentication with minimal configuration.

> **TIP**
> Azure Active Directory (Azure AD) and Azure AD B2C are separate product offerings. An Azure AD tenant represents an organization, while an Azure AD B2C tenant represents a collection of identities to be used with relying party applications. To learn more, see Azure AD B2C: Frequently asked questions (FAQ).

In this tutorial, learn how to:

- Create an Azure Active Directory B2C tenant
- Register an app in Azure AD B2C
- Use Visual Studio to create an ASP.NET Core web app configured to use the Azure AD B2C tenant for authentication
- Configure policies controlling the behavior of the Azure AD B2C tenant

Prerequisites

The following are required for this walkthrough:

- Microsoft Azure subscription
- Visual Studio 2019

Create the Azure Active Directory B2C tenant

Create an Azure Active Directory B2C tenant as described in the documentation. When prompted, associating the tenant with an Azure subscription is optional for this tutorial.

Register the app in Azure AD B2C

In the newly created Azure AD B2C tenant, register your app using the steps in the documentation under the **Register a web app** section. Stop at the **Create a web app client secret** section. A client secret isn't required for this tutorial.

Use the following values:

SETTING	VALUE	NOTES
Name	*<app name>*	Enter a **Name** for the app that describes your app to consumers.

SETTING	VALUE	NOTES
Include web app / web API	Yes	
Allow implicit flow	Yes	
Reply URL	`https://localhost:44300/signin-oidc`	Reply URLs are endpoints where Azure AD B2C returns any tokens that your app requests. Visual Studio provides the Reply URL to use. For now, enter `https://localhost:44300/signin-oidc` to complete the form.
App ID URI	Leave blank	Not required for this tutorial.
Include native client	No	

WARNING

If setting up a non-localhost Reply URL, be aware of the constraints on what is allowed in the Reply URL list.

After the app is registered, the list of apps in the tenant is displayed. Select the app that was just registered. Select the **Copy** icon to the right of the **Application ID** field to copy it to the clipboard.

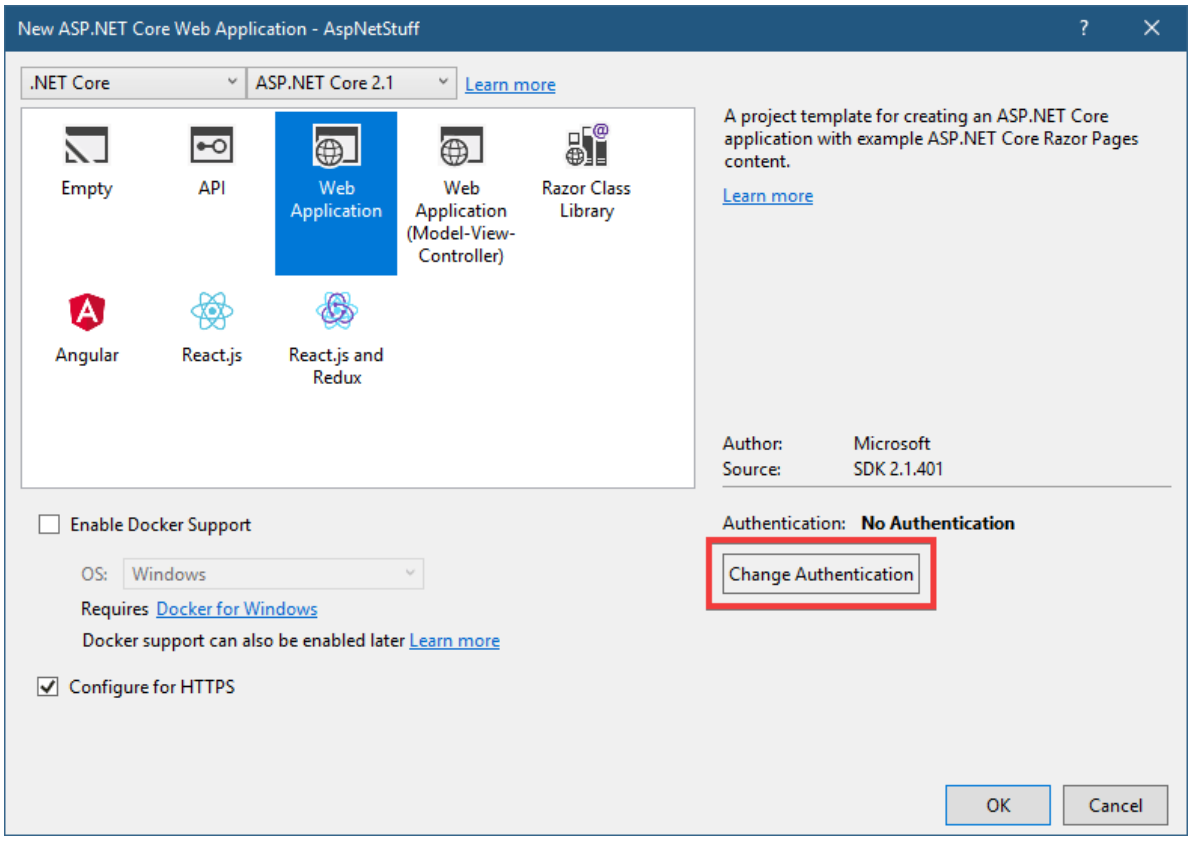
Nothing more can be configured in the Azure AD B2C tenant at this time, but leave the browser window open. There is more configuration after the ASP.NET Core app is created.

Create an ASP.NET Core app in Visual Studio

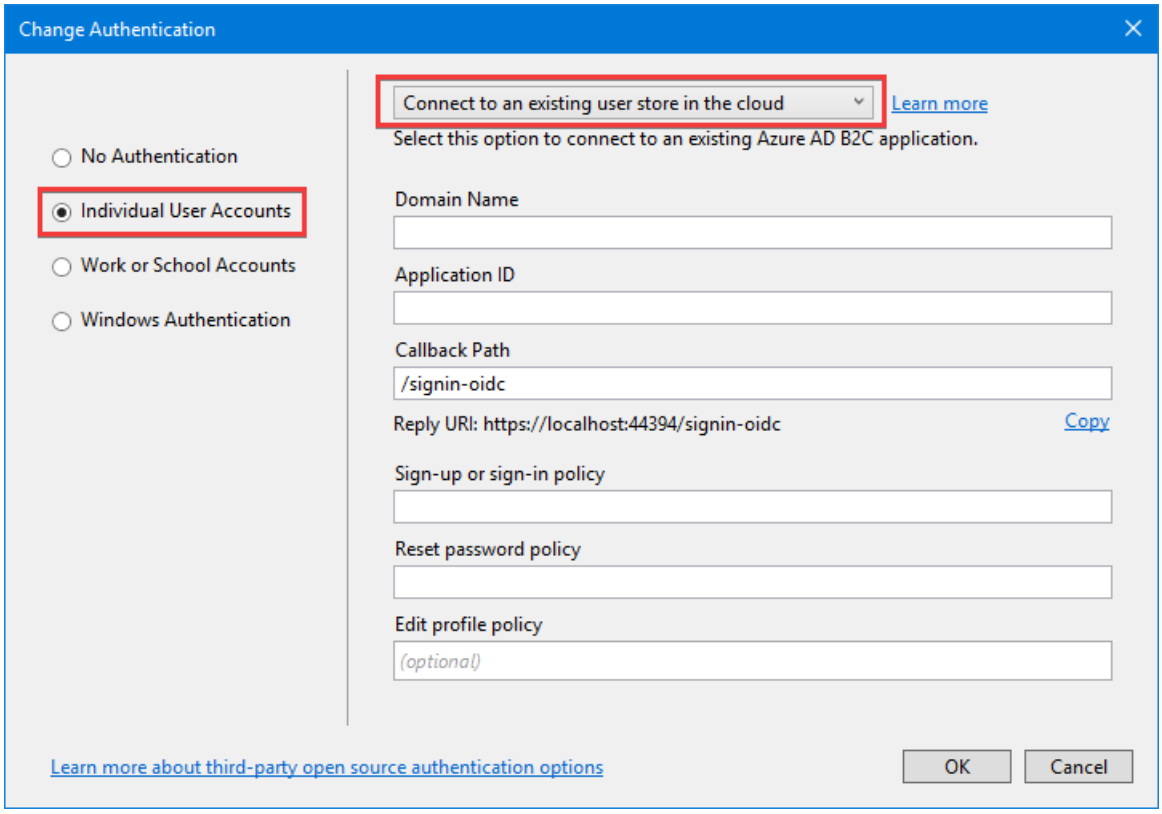
The Visual Studio Web Application template can be configured to use the Azure AD B2C tenant for authentication.

In Visual Studio:

1. Create a new ASP.NET Core Web Application.
2. Select **Web Application** from the list of templates.
3. Select the **Change Authentication** button.



4. In the **Change Authentication** dialog, select **Individual User Accounts**, and then select **Connect to an existing user store in the cloud** in the dropdown.



5. Complete the form with the following values:

SETTING	VALUE
Domain Name	*<the domain name of your B2C tenant>*
Application ID	*<paste the Application ID from the clipboard>*

SETTING	VALUE
Callback Path	*<use the default value>*
Sign-up or sign-in policy	`B2C_1_SiUpIn`
Reset password policy	`B2C_1_SSPR`
Edit profile policy	*<leave blank>*

Select the **Copy** link next to **Reply URI** to copy the Reply URI to the clipboard. Select **OK** to close the **Change Authentication** dialog. Select **OK** to create the web app.

Finish the B2C app registration

Return to the browser window with the B2C app properties still open. Change the temporary **Reply URL** specified earlier to the value copied from Visual Studio. Select **Save** at the top of the window.

> **TIP**
> If you didn't copy the Reply URL, use the HTTPS address from the Debug tab in the web project properties, and append the **CallbackPath** value from *appsettings.json*.

Configure policies

Use the steps in the Azure AD B2C documentation to create a sign-up or sign-in policy, and then create a password reset policy. Use the example values provided in the documentation for **Identity providers**, **Sign-up attributes**, and **Application claims**. Using the **Run now** button to test the policies as described in the documentation is optional.

> **WARNING**
> Ensure the policy names are exactly as described in the documentation, as those policies were used in the **Change Authentication** dialog in Visual Studio. The policy names can be verified in *appsettings.json*.

Configure the underlying OpenIdConnectOptions/JwtBearer/Cookie options

To configure the underlying options directly, use the appropriate scheme constant in `Startup.ConfigureServices`:

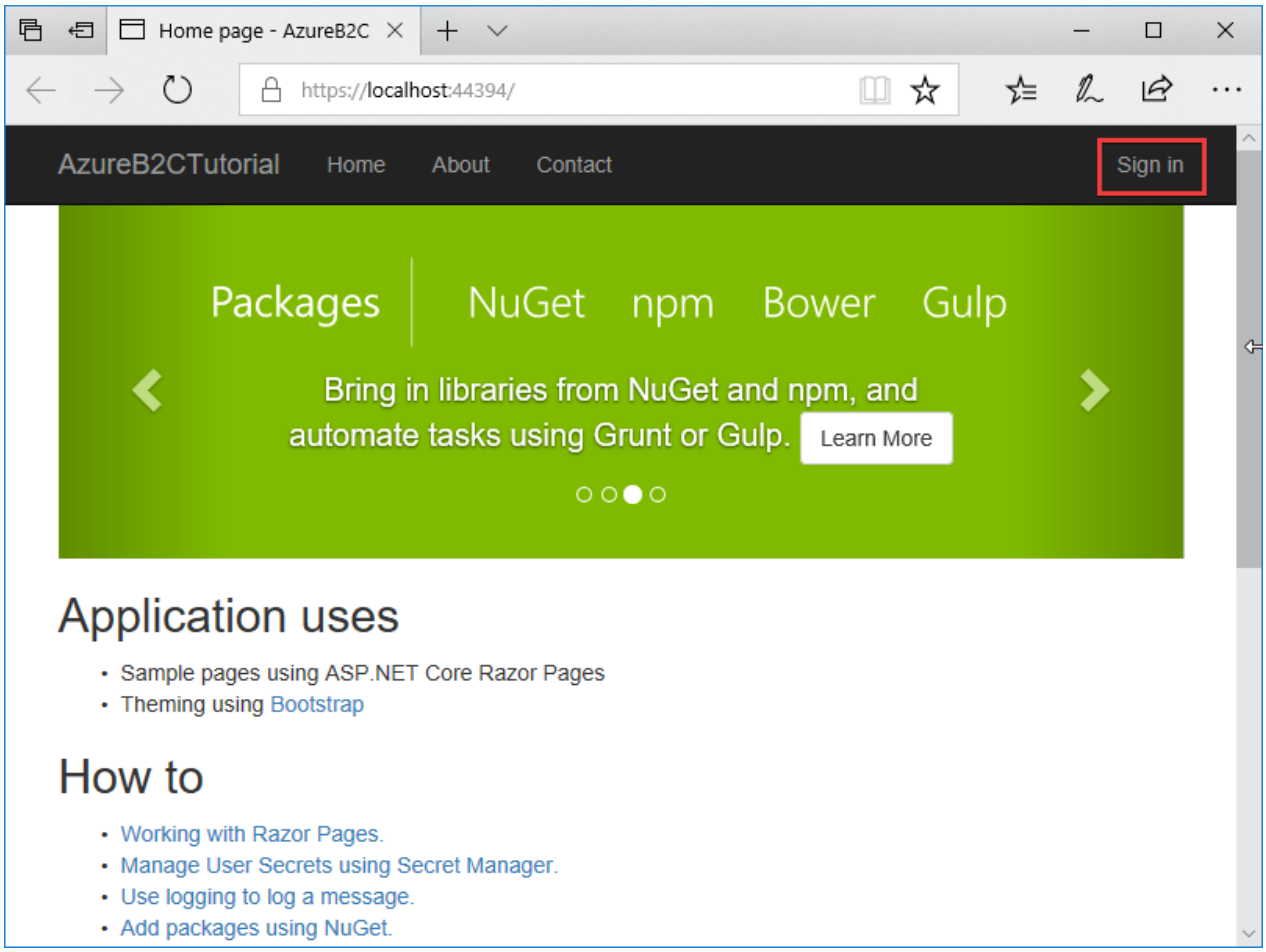
```
services.Configure<OpenIdConnectOptions>(
    AzureAD[B2C]Defaults.OpenIdScheme, options =>
    {
        // Omitted for brevity
    });

services.Configure<CookieAuthenticationOptions>(
    AzureAD[B2C]Defaults.CookieScheme, options =>
    {
        // Omitted for brevity
    });

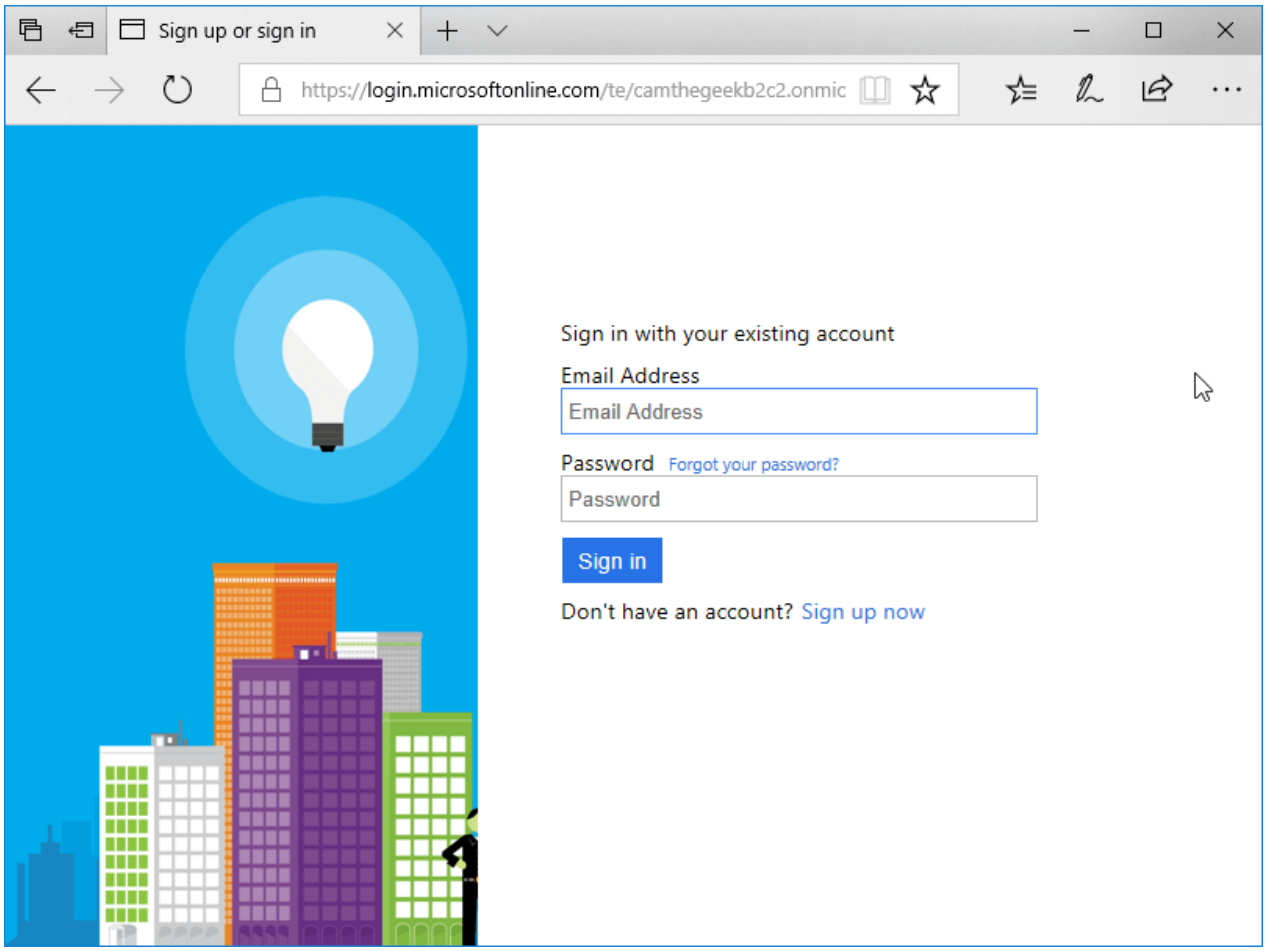
services.Configure<JwtBearerOptions>(
    AzureAD[B2C]Defaults.JwtBearerAuthenticationScheme, options =>
    {
        // Omitted for brevity
    });
```

Run the app

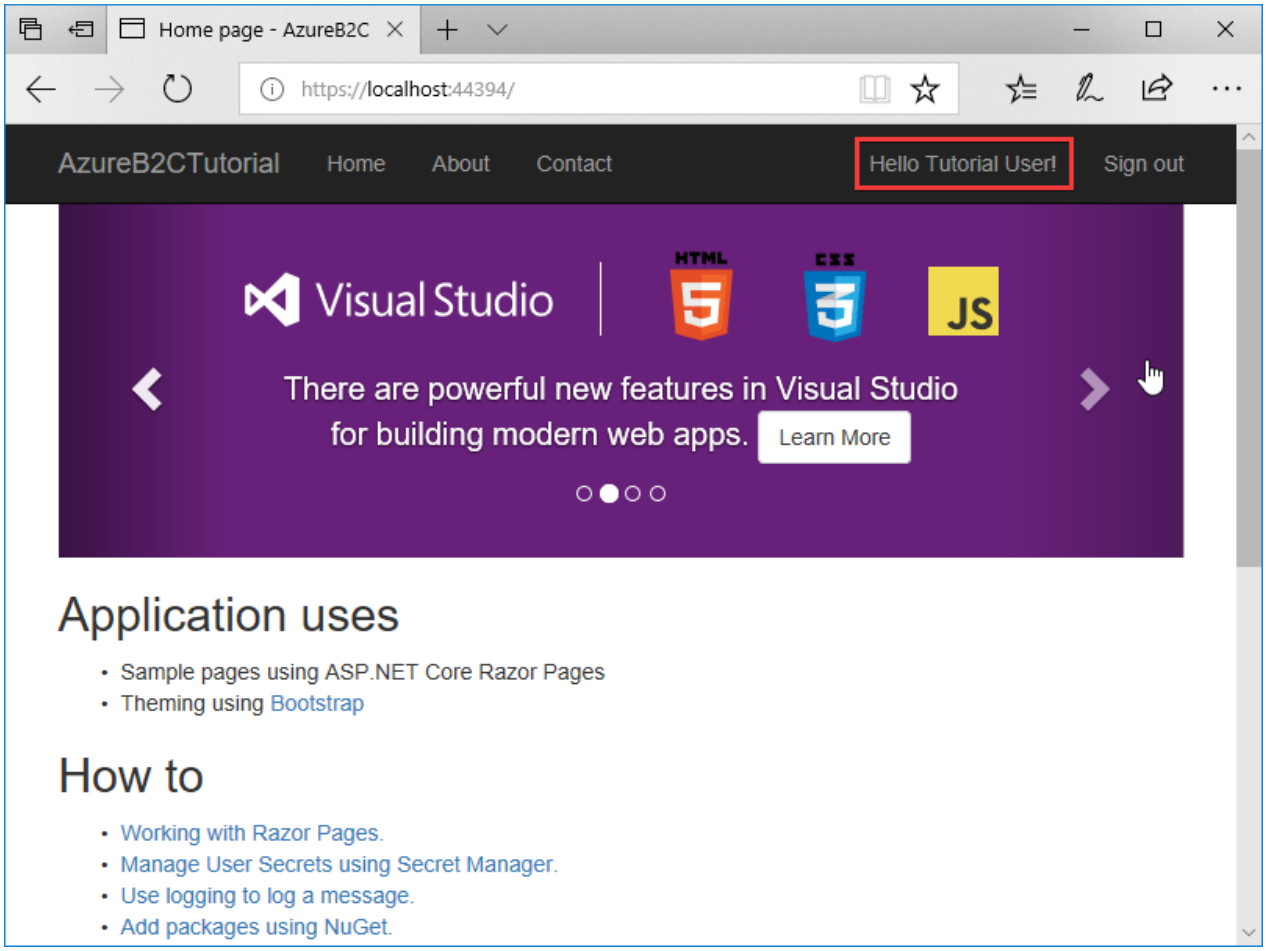
In Visual Studio, press **F5** to build and run the app. After the web app launches, select **Accept** to accept the use of cookies (if prompted), and then select **Sign in**.



The browser redirects to the Azure AD B2C tenant. Sign in with an existing account (if one was created testing the policies) or select **Sign up now** to create a new account. The **Forgot your password?** link is used to reset a forgotten password.



After successfully signing in, the browser redirects to the web app.



Next steps

In this tutorial, you learned how to:

- Create an Azure Active Directory B2C tenant
- Register an app in Azure AD B2C
- Use Visual Studio to create an ASP.NET Core Web Application configured to use the Azure AD B2C tenant for authentication
- Configure policies controlling the behavior of the Azure AD B2C tenant

Now that the ASP.NET Core app is configured to use Azure AD B2C for authentication, the Authorize attribute can be used to secure your app. Continue developing your app by learning to:

- Customize the Azure AD B2C user interface.
- Configure password complexity requirements.
- Enable multi-factor authentication.
- Configure additional identity providers, such as Microsoft, Facebook, Google, Amazon, Twitter, and others.
- Use the Azure AD Graph API to retrieve additional user information, such as group membership, from the Azure AD B2C tenant.
- How to secure a Web API built with ASP.NET Core using the Azure AD B2C.
- Tutorial: Grant access to an ASP.NET web API using Azure Active Directory B2C.

Articles based on ASP.NET Core projects created with individual user accounts

9/22/2020 • 2 minutes to read • Edit Online

ASP.NET Core Identity is included in project templates in Visual Studio with the "Individual User Accounts" option.

The authentication templates are available in .NET Core CLI with `-au Individual`:

```
dotnet new mvc -au Individual
dotnet new webapp -au Individual
```

```
dotnet new mvc -au Individual
dotnet new razor -au Individual
```

See this GitHub issue for web API authentication.

No Authentication

Authentication is specified in the .NET Core CLI with the `-au` option. In Visual Studio, the **Change Authentication** dialog is available for new web applications. The default for new web apps in Visual Studio is **No Authentication**.

Projects created with no authentication:

- Don't contain web pages and UI to sign in and sign out.
- Don't contain authentication code.

Windows Authentication

Windows Authentication is specified for new web apps in the .NET Core CLI with the `-au Windows` option. In Visual Studio, the **Change Authentication** dialog provides the **Windows Authentication** options.

If Windows Authentication is selected, the app is configured to use the Windows Authentication IIS module. Windows Authentication is intended for Intranet web sites.

dotnet new webapp authentication options

The following table shows the authentication options available for new web apps:

OPTION	TYPE OF AUTHENTICATION	LINK FOR MORE INFORMATION
None	No authentication	
Individual	Individual authentication	Introduction to Identity on ASP.NET Core
IndividualB2C	Cloud-hosted individual authentication with Azure AD B2C	Azure AD B2C

OPTION	TYPE OF AUTHENTICATION	LINK FOR MORE INFORMATION
SingleOrg	Organizational authentication for a single tenant	Azure AD
MultiOrg	Organizational authentication for multiple tenants	Azure AD
Windows	Windows authentication	Windows Authentication

Visual Studio new webapp authentication options

The following table shows the authentication options available when creating a new web app with Visual Studio:

OPTION	TYPE OF AUTHENTICATION	LINK FOR MORE INFORMATION
None	No authentication	
Individual User Accounts / Store user accounts in-app	Individual authentication	Introduction to Identity on ASP.NET Core
Individual User Accounts / Connect to an existing user store in the cloud	Cloud-hosted individual authentication with Azure AD B2C	Azure AD B2C
Work or School Cloud / Single Org	Organizational authentication for a single tenant	Azure AD
Work or School Cloud / Multiple Org	Organizational authentication for multiple tenants	Azure AD
Windows	Windows authentication	Windows Authentication

Additional resources

The following articles show how to use the code generated in ASP.NET Core templates that use individual user accounts:

- Account confirmation and password recovery in ASP.NET Core
- Create an ASP.NET Core app with user data protected by authorization

Configure certificate authentication in ASP.NET Core

9/22/2020 • 13 minutes to read • Edit Online

`Microsoft.AspNetCore.Authentication.Certificate` contains an implementation similar to Certificate Authentication for ASP.NET Core. Certificate authentication happens at the TLS level, long before it ever gets to ASP.NET Core. More accurately, this is an authentication handler that validates the certificate and then gives you an event where you can resolve that certificate to a `ClaimsPrincipal`.

Configure your server for certificate authentication, be it IIS, Kestrel, Azure Web Apps, or whatever else you're using.

Proxy and load balancer scenarios

Certificate authentication is a stateful scenario primarily used where a proxy or load balancer doesn't handle traffic between clients and servers. If a proxy or load balancer is used, certificate authentication only works if the proxy or load balancer:

- Handles the authentication.
- Passes the user authentication information to the app (for example, in a request header), which acts on the authentication information.

An alternative to certificate authentication in environments where proxies and load balancers are used is Active Directory Federated Services (ADFS) with OpenID Connect (OIDC).

Get started

Acquire an HTTPS certificate, apply it, and configure your server to require certificates.

In your web app, add a reference to the Microsoft.AspNetCore.Authentication.Certificate package. Then in the `Startup.ConfigureServices` method, call `services.AddAuthentication(CertificateAuthenticationDefaults.AuthenticationScheme).AddCertificate(...);` with your options, providing a delegate for `OnCertificateValidated` to do any supplementary validation on the client certificate sent with requests. Turn that information into a `ClaimsPrincipal` and set it on the `context.Principal` property.

If authentication fails, this handler returns a `403 (Forbidden)` response rather a `401 (Unauthorized)`, as you might expect. The reasoning is that the authentication should happen during the initial TLS connection. By the time it reaches the handler, it's too late. There's no way to upgrade the connection from an anonymous connection to one with a certificate.

Also add `app.UseAuthentication();` in the `Startup.Configure` method. Otherwise, the `HttpContext.User` will not be set to `ClaimsPrincipal` created from the certificate. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
        .AddCertificate()
        // Adding an ICertificateValidationCache results in certificate auth caching the results.
        // The default implementation uses a memory cache.
        .AddCertificateCache();

    // All other service configuration
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();

    // All other app configuration
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
        .AddCertificate();

    // All other service configuration
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();

    // All other app configuration
}
```

The preceding example demonstrates the default way to add certificate authentication. The handler constructs a user principal using the common certificate properties.

Configure certificate validation

The `CertificateAuthenticationOptions` handler has some built-in validations that are the minimum validations you should perform on a certificate. Each of these settings is enabled by default.

AllowedCertificateTypes = Chained, SelfSigned, or All (Chained | SelfSigned)

Default value: `CertificateTypes.Chained`

This check validates that only the appropriate certificate type is allowed. If the app is using self-signed certificates, this option needs to be set to `CertificateTypes.All` or `CertificateTypes.SelfSigned`.

ValidateCertificateUse

Default value: `true`

This check validates that the certificate presented by the client has the Client Authentication extended key use (EKU), or no EKUs at all. As the specifications say, if no EKU is specified, then all EKUs are deemed valid.

ValidateValidityPeriod

Default value: `true`

This check validates that the certificate is within its validity period. On each request, the handler ensures that a certificate that was valid when it was presented hasn't expired during its current session.

RevocationFlag

Default value: `X509RevocationFlag.ExcludeRoot`

A flag that specifies which certificates in the chain are checked for revocation.

Revocation checks are only performed when the certificate is chained to a root certificate.

RevocationMode

Default value: `X509RevocationMode.Online`

A flag that specifies how revocation checks are performed.

Specifying an online check can result in a long delay while the certificate authority is contacted.

Revocation checks are only performed when the certificate is chained to a root certificate.

Can I configure my app to require a certificate only on certain paths?

This isn't possible. Remember the certificate exchange is done that the start of the HTTPS conversation, it's done by the server before the first request is received on that connection so it's not possible to scope based on any request fields.

Handler events

The handler has two events:

- `OnAuthenticationFailed`: Called if an exception happens during authentication and allows you to react.
- `OnCertificateValidated`: Called after the certificate has been validated, passed validation and a default principal has been created. This event allows you to perform your own validation and augment or replace the principal. For examples include:
  - Determining if the certificate is known to your services.
  - Constructing your own principal. Consider the following example in `Startup.ConfigureServices`:

```
services.AddAuthentication(
    CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate(options =>
    {
        options.Events = new CertificateAuthenticationEvents
        {
            OnCertificateValidated = context =>
            {
                var claims = new[]
                {
                    new Claim(
                        ClaimTypes.NameIdentifier,
                        context.ClientCertificate.Subject,
                        ClaimValueTypes.String,
                        context.Options.ClaimsIssuer),
                    new Claim(ClaimTypes.Name,
                        context.ClientCertificate.Subject,
                        ClaimValueTypes.String,
                        context.Options.ClaimsIssuer)
                };

                context.Principal = new ClaimsPrincipal(
                    new ClaimsIdentity(claims, context.Scheme.Name));
                context.Success();

                return Task.CompletedTask;
            }
        };
    });
```

If you find the inbound certificate doesn't meet your extra validation, call `context.Fail("failure reason")` with a failure reason.

For real functionality, you'll probably want to call a service registered in dependency injection that connects to a database or other type of user store. Access your service by using the context passed into your delegate. Consider the following example in `Startup.ConfigureServices`:

```
services.AddAuthentication(
    CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate(options =>
    {
        options.Events = new CertificateAuthenticationEvents
        {
            OnCertificateValidated = context =>
            {
                var validationService =
                    context.HttpContext.RequestServices
                        .GetService<ICertificateValidationService>();

                if (validationService.ValidateCertificate(
                    context.ClientCertificate))
                {
                    var claims = new[]
                    {
                        new Claim(
                            ClaimTypes.NameIdentifier,
                            context.ClientCertificate.Subject,
                            ClaimValueTypes.String,
                            context.Options.ClaimsIssuer),
                        new Claim(
                            ClaimTypes.Name,
                            context.ClientCertificate.Subject,
                            ClaimValueTypes.String,
                            context.Options.ClaimsIssuer)
                    };

                    context.Principal = new ClaimsPrincipal(
                        new ClaimsIdentity(claims, context.Scheme.Name));
                    context.Success();
                }

                return Task.CompletedTask;
            }
        };
    });
```

Conceptually, the validation of the certificate is an authorization concern. Adding a check on, for example, an issuer or thumbprint in an authorization policy, rather than inside `OnCertificateValidated`, is perfectly acceptable.

Configure your server to require certificates

Kestrel

In *Program.cs*, configure Kestrel as follows:

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}

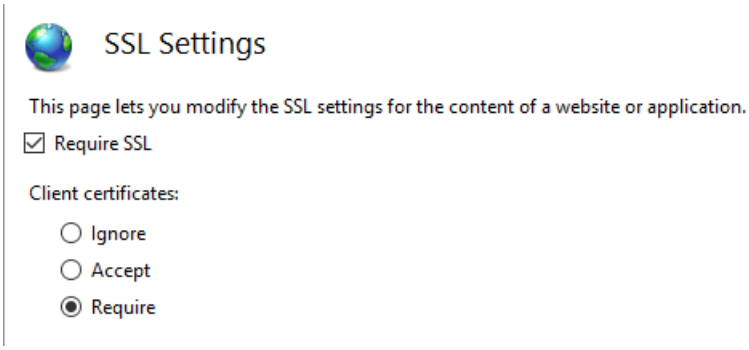
public static IHostBuilder CreateHostBuilder(string[] args)
{
    return Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
            webBuilder.ConfigureKestrel(o =>
            {
                o.ConfigureHttpsDefaults(o =>
      o.ClientCertificateMode =
          ClientCertificateMode.RequireCertificate);
            });
        });
}
```

> **NOTE**
>
> Endpoints created by calling Listen **before** calling ConfigureHttpsDefaults won't have the defaults applied.

IIS

Complete the following steps in IIS Manager:

1. Select your site from the **Connections** tab.
2. Double-click the **SSL Settings** option in the **Features View** window.
3. Check the **Require SSL** checkbox, and select the **Require** radio button in the **Client certificates** section.



Azure and custom web proxies

See the host and deploy documentation for how to configure the certificate forwarding middleware.

Use certificate authentication in Azure Web Apps

No forwarding configuration is required for Azure. This is already setup in the certificate forwarding middleware.

> **NOTE**
>
> This requires that the CertificateForwardingMiddleware is present.

Use certificate authentication in custom web proxies

The `AddCertificateForwarding` method is used to specify:

- The client header name.
- How the certificate is to be loaded (using the `HeaderConverter` property).

In custom web proxies, the certificate is passed as a custom request header, for example `X-SSL-CERT`. To use it, configure certificate forwarding in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCertificateForwarding(options =>
    {
        options.CertificateHeader = "X-SSL-CERT";
        options.HeaderConverter = (headerValue) =>
        {
            X509Certificate2 clientCertificate = null;

            if(!string.IsNullOrWhiteSpace(headerValue))
            {
                byte[] bytes = StringToByteArray(headerValue);
                clientCertificate = new X509Certificate2(bytes);
            }

            return clientCertificate;
        };
    });
}

private static byte[] StringToByteArray(string hex)
{
    int NumberChars = hex.Length;
    byte[] bytes = new byte[NumberChars / 2];

    for (int i = 0; i < NumberChars; i += 2)
    {
        bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);
    }

    return bytes;
}
```

The `Startup.Configure` method then adds the middleware. `UseCertificateForwarding` is called before the calls to `UseAuthentication` and `UseAuthorization`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    ...

    app.UseRouting();

    app.UseCertificateForwarding();
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

A separate class can be used to implement validation logic. Because the same self-signed certificate is used in this example, ensure that only your certificate can be used. Validate that the thumbprints of both the client certificate and the server certificate match, otherwise any certificate can be used and will be enough to authenticate. This would be used inside the `AddCertificate` method. You could also validate the subject or the issuer here if you're using intermediate or child certificates.

```
using System.IO;
using System.Security.Cryptography.X509Certificates;

namespace AspNetCoreCertificateAuthApi
{
    public class MyCertificateValidationService
    {
        public bool ValidateCertificate(X509Certificate2 clientCertificate)
        {
            // Do not hardcode passwords in production code
            // Use thumbprint or key vault
            var cert = new X509Certificate2(
                Path.Combine("sts_dev_cert.pfx"), "1234");

            if (clientCertificate.Thumbprint == cert.Thumbprint)
            {
                return true;
            }

            return false;
        }
    }
}
```

Implement an HttpClient using a certificate and the HttpClientHandler

The `HttpClientHandler` could be added directly in the constructor of the `HttpClient` class. Care should be taken when creating instances of the `HttpClient`. The `HttpClient` will then send the certificate with each request.

```
private async Task<JsonDocument> GetApiDataUsingHttpClientHandler()
{
    var cert = new X509Certificate2(Path.Combine(_environment.ContentRootPath, "sts_dev_cert.pfx"), "1234");
    var handler = new HttpClientHandler();
    handler.ClientCertificates.Add(cert);
    var client = new HttpClient(handler);

    var request = new HttpRequestMessage()
    {
        RequestUri = new Uri("https://localhost:44379/api/values"),
        Method = HttpMethod.Get,
    };
    var response = await client.SendAsync(request);
    if (response.IsSuccessStatusCode)
    {
        var responseContent = await response.Content.ReadAsStringAsync();
        var data = JsonDocument.Parse(responseContent);
        return data;
    }

    throw new ApplicationException($"Status code: {response.StatusCode}, Error: {response.ReasonPhrase}");
}
```

Implement an HttpClient using a certificate and a named HttpClient from IHttpClientFactory

In the following example, a client certificate is added to a `HttpClientHandler` using the `ClientCertificates` property from the handler. This handler can then be used in a named instance of an `HttpClient` using the `ConfigurePrimaryHttpMessageHandler` method. This is setup in `Startup.ConfigureServices`:

```
var clientCertificate =
    new X509Certificate2(
      Path.Combine(_environment.ContentRootPath, "sts_dev_cert.pfx"), "1234");

var handler = new HttpClientHandler();
handler.ClientCertificates.Add(clientCertificate);

services.AddHttpClient("namedClient", c =>
{
}).ConfigurePrimaryHttpMessageHandler(() => handler);
```

The `IHttpClientFactory` can then be used to get the named instance with the handler and the certificate. The `CreateClient` method with the name of the client defined in the `Startup` class is used to get the instance. The HTTP request can be sent using the client as required.

```
private readonly IHttpClientFactory _clientFactory;

public ApiService(IHttpClientFactory clientFactory)
{
    _clientFactory = clientFactory;
}

private async Task<JsonDocument> GetApiDataWithNamedClient()
{
    var client = _clientFactory.CreateClient("namedClient");

    var request = new HttpRequestMessage()
    {
        RequestUri = new Uri("https://localhost:44379/api/values"),
        Method = HttpMethod.Get,
    };
    var response = await client.SendAsync(request);
    if (response.IsSuccessStatusCode)
    {
        var responseContent = await response.Content.ReadAsStringAsync();
        var data = JsonDocument.Parse(responseContent);
        return data;
    }

    throw new ApplicationException($"Status code: {response.StatusCode}, Error: {response.ReasonPhrase}");
}
```

If the correct certificate is sent to the server, the data is returned. If no certificate or the wrong certificate is sent, an HTTP 403 status code is returned.

Create certificates in PowerShell

Creating the certificates is the hardest part in setting up this flow. A root certificate can be created using the `New-SelfSignedCertificate` PowerShell cmdlet. When creating the certificate, use a strong password. It's important to add the `KeyUsageProperty` parameter and the `KeyUsage` parameter as shown.

Create root CA

```
New-SelfSignedCertificate -DnsName "root_ca_dev_damienbod.com", "root_ca_dev_damienbod.com" -CertStoreLocation
"cert:\LocalMachine\My" -NotAfter (Get-Date).AddYears(20) -FriendlyName "root_ca_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-PfxCertificate -FilePath
C:\git\root_ca_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath root_ca_dev_damienbod.crt
```

NOTE

The `-DnsName` parameter value must match the deployment target of the app. For example, "localhost" for development.

Install in the trusted root

The root certificate needs to be trusted on your host system. A root certificate which was not created by a certificate authority won't be trusted by default. The following link explains how this can be accomplished on Windows:

https://social.msdn.microsoft.com/Forums/SqlServer/5ed119ef-1704-4be4-8a4f-ef11de7c8f34/a-certificate-chain-processed-but-terminated-in-a-root-certificate-which-is-not-trusted-by-the

Intermediate certificate

An intermediate certificate can now be created from the root certificate. This isn't required for all use cases, but you might need to create many certificates or need to activate or disable groups of certificates. The `TextExtension` parameter is required to set the path length in the basic constraints of the certificate.

The intermediate certificate can then be added to the trusted intermediate certificate in the Windows host system.

```
$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

$parentcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint of the root..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname "intermediate_dev_damienbod.com" -
Signer $parentcert -NotAfter (Get-Date).AddYears(20) -FriendlyName "intermediate_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature -TextExtension @("2.5.29.19=
{text}CA=1&pathlength=1")

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\intermediate_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath intermediate_dev_damienbod.crt
```

Create child certificate from intermediate certificate

A child certificate can be created from the intermediate certificate. This is the end entity and doesn't need to create more child certificates.

```
$parentcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint from the Intermediate
certificate..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname "child_a_dev_damienbod.com" -
Signer $parentcert -NotAfter (Get-Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com"

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath child_a_dev_damienbod.crt
```

Create child certificate from root certificate

A child certificate can also be created from the root certificate directly.

```
$rootcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint from the root cert..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname "child_a_dev_damienbod.com" -
Signer $rootcert -NotAfter (Get-Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com"

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath child_a_dev_damienbod.crt
```

Example root - intermediate certificate - certificate

```
$mypwdroot = ConvertTo-SecureString -String "1234" -Force -AsPlainText
$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

New-SelfSignedCertificate -DnsName "root_ca_dev_damienbod.com", "root_ca_dev_damienbod.com" -CertStoreLocation
"cert:\LocalMachine\My" -NotAfter (Get-Date).AddYears(20) -FriendlyName "root_ca_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature

Get-ChildItem -Path cert:\localMachine\my\0C89639E4E2998A93E423F919B36D4009A0F9991 | Export-PfxCertificate -
FilePath C:\git\root_ca_dev_damienbod.pfx -Password $mypwdroot

Export-Certificate -Cert cert:\localMachine\my\0C89639E4E2998A93E423F919B36D4009A0F9991 -FilePath
root_ca_dev_damienbod.crt

$rootcert = ( Get-ChildItem -Path cert:\LocalMachine\My\0C89639E4E2998A93E423F919B36D4009A0F9991 )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname "child_a_dev_damienbod.com" -
Signer $rootcert -NotAfter (Get-Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com" -KeyUsageProperty
All -KeyUsage CertSign, CRLSign, DigitalSignature -TextExtension @("2.5.29.19={text}CA=1&pathlength=1")

Get-ChildItem -Path cert:\localMachine\my\BA9BF91ED35538A01375EFC212A2F46104B33A44 | Export-PfxCertificate -
FilePath C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\BA9BF91ED35538A01375EFC212A2F46104B33A44 -FilePath
child_a_dev_damienbod.crt

$parentcert = ( Get-ChildItem -Path cert:\LocalMachine\My\BA9BF91ED35538A01375EFC212A2F46104B33A44 )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname "child_b_from_a_dev_damienbod.com"
-Signer $parentcert -NotAfter (Get-Date).AddYears(20) -FriendlyName "child_b_from_a_dev_damienbod.com"

Get-ChildItem -Path cert:\localMachine\my\141594A0AE38CBBECED7AF680F7945CD51D8F28A | Export-PfxCertificate -
FilePath C:\git\AspNetCoreCertificateAuth\Certs\child_b_from_a_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\141594A0AE38CBBECED7AF680F7945CD51D8F28A -FilePath
child_b_from_a_dev_damienbod.crt
```

When using the root, intermediate, or child certificates, the certificates can be validated using the Thumbprint or PublicKey as required.

```
using System.Collections.Generic;
using System.IO;
using System.Security.Cryptography.X509Certificates;

namespace AspNetCoreCertificateAuthApi
{
    public class MyCertificateValidationService
    {
        public bool ValidateCertificate(X509Certificate2 clientCertificate)
        {
            return CheckIfThumbprintIsValid(clientCertificate);
        }

        private bool CheckIfThumbprintIsValid(X509Certificate2 clientCertificate)
        {
            var listOfValidThumbprints = new List<string>
            {
                "141594A0AE38CBBECED7AF680F7945CD51D8F28A",
                "0C89639E4E2998A93E423F919B36D4009A0F9991",
                "BA9BF91ED35538A01375EFC212A2F46104B33A44"
            };

            if (listOfValidThumbprints.Contains(clientCertificate.Thumbprint))
            {
                return true;
            }

            return false;
        }
    }
}
```

Certificate validation caching

ASP.NET Core 5.0 and later versions support the ability to enable caching of validation results. The caching dramatically improves performance of certificate authentication, as validation is an expensive operation.

By default, certificate authentication disables caching. To enable caching, call `AddCertificateCache` in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
            .AddCertificate()
            .AddCertificateCache(options =>
            {
                options.CacheSize = 1024;
                options.CacheEntryExpiration = TimeSpan.FromMinutes(2);
            });
}
```

The default caching implementation stores results in memory. You can provide your own cache by implementing `ICertificateValidationCache` and registering it with dependency injection. For example, `services.AddSingleton<ICertificateValidationCache, YourCache>()`.

Optional client certificates

This section provides information for apps that must protect a subset of the app with a certificate. For example, a Razor Page or controller in the app might require client certificates. This presents challenges as client certificates:

- Are a TLS feature, not an HTTP feature.
- Are negotiated per-connection and must be be negotiated at the start of the connection before any HTTP data is available. At the start of the connection, only the Server Name Indication (SNI)† is known. The client and server certificates are negotiated prior to the first request on a connection and requests generally aren't able to renegotiate.

TLS renegotiation was an old way to implement optional client certificates. This is no longer recommended because:

- In HTTP/1.1, renegotiating during a POST request could cause a deadlock where the request body filled up the TCP window and the renegotiation packets can't be received.
- HTTP/2 explicitly prohibits renegotiation.
- TLS 1.3 has removed support for renegotiation.

ASP.NET Core 5 preview 7 and later adds more convenient support for optional client certificates. For more information, see the Optional certificates sample.

The following approach supports optional client certificates:

- Set up binding for the domain and subdomain:
  - For example, set up bindings on `contoso.com` and `myClient.contoso.com`. The `contoso.com` host doesn't require a client certificate but `myClient.contoso.com` does.
  - For more information, see:
    - Kestrel:
      - ListenOptions.UseHttps
      - ClientCertificateMode
      - Note Kestrel does not currently support multiple TLS configurations on one binding, you'll need two bindings with unique IPs or ports. See https://github.com/dotnet/runtime/issues/31097
    - IIS
      - Hosting IIS
      - Configure security on IIS
    - Http.Sys: Configure Windows Server
- For requests to the web app that require a client certificate and don't have one:
  - Redirect to the same page using the client certificate protected subdomain.
  - For example, redirect to `myClient.contoso.com/requestedPage`. Because the request to `myClient.contoso.com/requestedPage` is a different hostname than `contoso.com/requestedPage`, the client establishes a different connection and the client certificate is provided.
  - For more information, see Introduction to authorization in ASP.NET Core.

Leave questions, comments, and other feedback on optional client certificates in this GitHub discussion issue.

† Server Name Indication (SNI) is a TLS extension to include a virtual domain as a part of SSL negotiation. This effectively means the virtual domain name, or a hostname, can be used to identify the network end point.

Multi-factor authentication in ASP.NET Core

9/22/2020 • 11 minutes to read • Edit Online

By Damien Bowden

Multi-factor authentication (MFA) is a process in which a user is requested during a sign-in event for additional forms of identification. This prompt could be to enter a code from a cellphone, use a FIDO2 key, or to provide a fingerprint scan. When you require a second form of authentication, security is enhanced. The additional factor isn't easily obtained or duplicated by an attacker.

This article covers the following areas:

- What is MFA and what MFA flows are recommended
- Configure MFA for administration pages using ASP.NET Core Identity
- Send MFA sign-in requirement to OpenID Connect server
- Force ASP.NET Core OpenID Connect client to require MFA

MFA, 2FA

MFA requires at least two or more types of proof for an identity like something you know, something you possess, or biometric validation for the user to authenticate.

Two-factor authentication (2FA) is like a subset of MFA, but the difference being that MFA can require two or more factors to prove the identity.

MFA TOTP (Time-based One-time Password Algorithm)

MFA using TOTP is a supported implementation using ASP.NET Core Identity. This can be used together with any compliant authenticator app, including:

- Microsoft Authenticator App
- Google Authenticator App

See the following link for implementation details:

Enable QR Code generation for TOTP authenticator apps in ASP.NET Core

MFA FIDO2 or passwordless

FIDO2 is currently:

- The most secure way of achieving MFA.
- The only MFA flow that protects against phishing attacks.

At present, ASP.NET Core doesn't support FIDO2 directly. FIDO2 can be used for MFA or passwordless flows.

Azure Active Directory provides support for FIDO2 and passwordless flows. For more information, see Passwordless authentication options for Azure Active Directory.

MFA SMS

MFA with SMS increases security massively compared with password authentication (single factor). However, using SMS as a second factor is no longer recommended. Too many known attack vectors exist for this type of implementation.

NIST guidelines

Configure MFA for administration pages using ASP.NET Core Identity

MFA could be forced on users to access sensitive pages within an ASP.NET Core Identity app. This could be useful for apps where different levels of access exist for the different identities. For example, users might be able to view the profile data using a password login, but an administrator would be required to use MFA to access the administrative pages.

Extend the login with an MFA claim

The demo code is setup using ASP.NET Core with Identity and Razor Pages. The `AddIdentity` method is used instead of `AddDefaultIdentity` one, so an `IUserClaimsPrincipalFactory` implementation can be used to add claims to the identity after a successful login.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlite(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<IdentityUser, IdentityRole>(
            options => options.SignIn.RequireConfirmedAccount = false)
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddSingleton<IEmailSender, EmailSender>();
    services.AddScoped<IUserClaimsPrincipalFactory<IdentityUser>,
        AdditionalUserClaimsPrincipalFactory>();

    services.AddAuthorization(options =>
        options.AddPolicy("TwoFactorEnabled",
            x => x.RequireClaim("amr", "mfa")));

    services.AddRazorPages();
}
```

The `AdditionalUserClaimsPrincipalFactory` class adds the `amr` claim to the user claims only after a successful login. The claim's value is read from the database. The claim is added here because the user should only access the higher protected view if the identity has logged in with MFA. If the database view is read from the database directly instead of using the claim, it's possible to access the view without MFA directly after activating the MFA.

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using System.Collections.Generic;
using System.Security.Claims;
using System.Threading.Tasks;

namespace IdentityStandaloneMfa
{
    public class AdditionalUserClaimsPrincipalFactory :
        UserClaimsPrincipalFactory<IdentityUser, IdentityRole>
    {
        public AdditionalUserClaimsPrincipalFactory(
            UserManager<IdentityUser> userManager,
            RoleManager<IdentityRole> roleManager,
            IOptions<IdentityOptions> optionsAccessor)
            : base(userManager, roleManager, optionsAccessor)
        {
        }

        public async override Task<ClaimsPrincipal> CreateAsync(IdentityUser user)
        {
            var principal = await base.CreateAsync(user);
            var identity = (ClaimsIdentity)principal.Identity;

            var claims = new List<Claim>();

            if (user.TwoFactorEnabled)
            {
                claims.Add(new Claim("amr", "mfa"));
            }
            else
            {
                claims.Add(new Claim("amr", "pwd"));
            }

            identity.AddClaims(claims);
            return principal;
        }
    }
}
```

Because the Identity service setup changed in the `Startup` class, the layouts of the Identity need to be updated. Scaffold the Identity pages into the app. Define the layout in the *Identity/Account/Manage/_Layout.cshtml* file.

```
@{
    Layout = "/Pages/Shared/_Layout.cshtml";
}
```

Also assign the layout for all the manage pages from the Identity pages:

```
@{
    Layout = "_Layout.cshtml";
}
```

Validate the MFA requirement in the administration page

The administration Razor Page validates that the user has logged in using MFA. In the `OnGet` method, the identity is used to access the user claims. The `amr` claim is checked for the value `mfa`. If the identity is missing this claim or is `false`, the page redirects to the Enable MFA page. This is possible because the user has logged in already, but without MFA.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityStandaloneMfa
{
    public class AdminModel : PageModel
    {
        public IActionResult OnGet()
        {
            var claimTwoFactorEnabled =
                User.Claims.FirstOrDefault(t => t.Type == "amr");

            if (claimTwoFactorEnabled != null &&
                "mfa".Equals(claimTwoFactorEnabled.Value))
            {
                // You logged in with MFA, do the administrative stuff
            }
            else
            {
                return Redirect(
                    "/Identity/Account/Manage/TwoFactorAuthentication");
            }

            return Page();
        }
    }
}
```

UI logic to toggle user login information

An authorization policy was added at startup. The policy requires the `amr` claim with the value `mfa`.

```
services.AddAuthorization(options =>
    options.AddPolicy("TwoFactorEnabled",
        x => x.RequireClaim("amr", "mfa")));
```

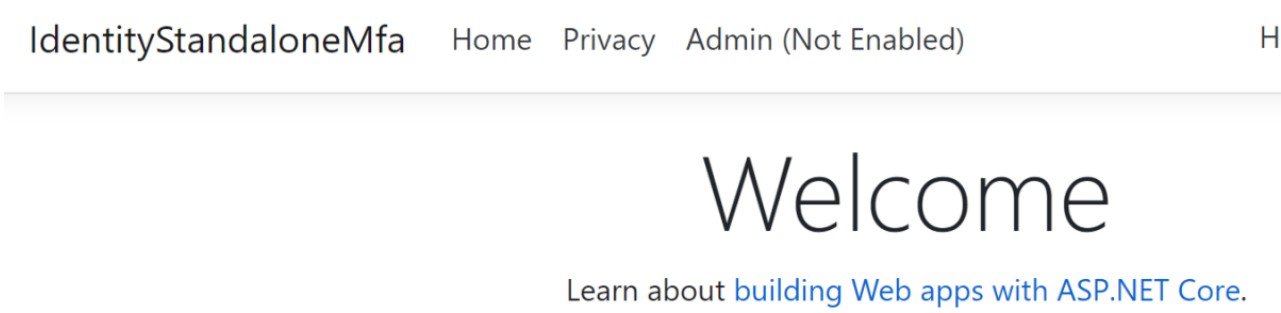
This policy can then be used in the `_Layout` view to show or hide the **Admin** menu with the warning:

```
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager
@inject IAuthorizationService AuthorizationService
```

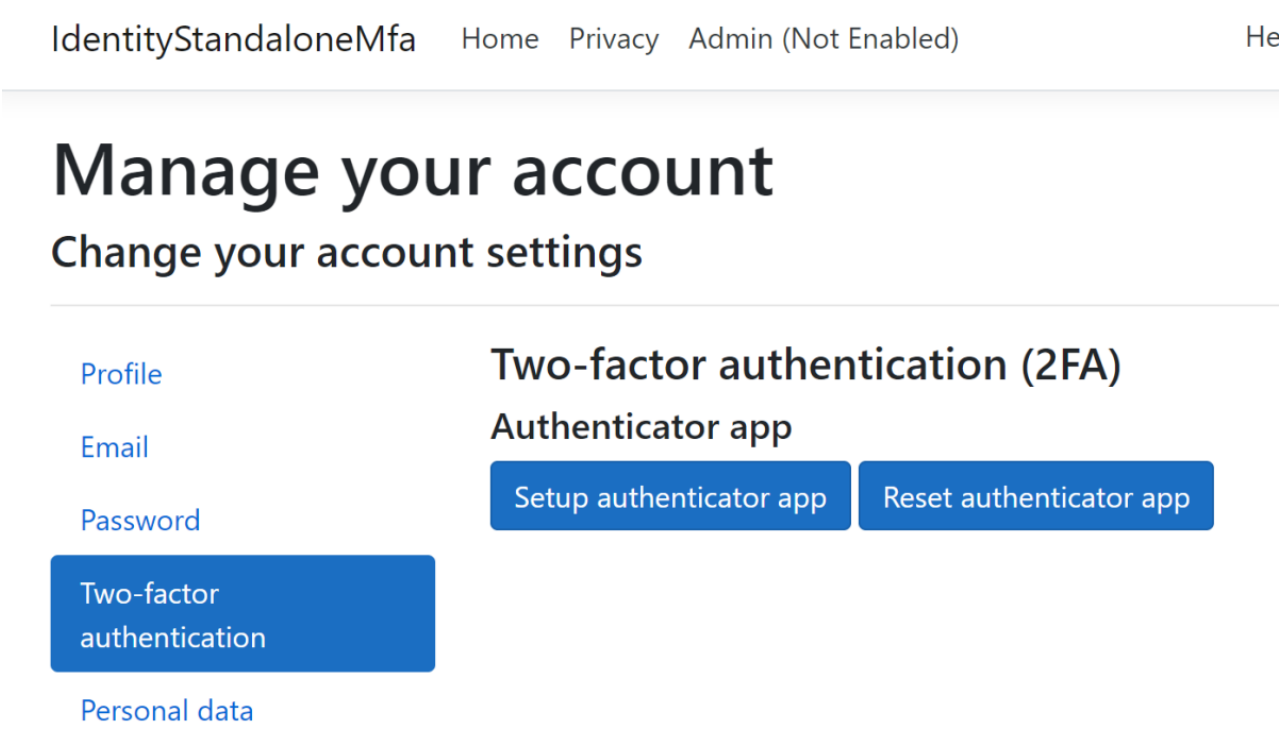
If the identity has logged in using MFA, the **Admin** menu is displayed without the tooltip warning. When the user has logged in without MFA, the **Admin (Not Enabled)** menu is displayed along with the tooltip that informs the user (explaining the warning).

```
@if (SignInManager.IsSignedIn(User))
{
    @if ((AuthorizationService.AuthorizeAsync(User, "TwoFactorEnabled")).Result.Succeeded)
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-page="/Admin">Admin</a>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-page="/Admin"
               id="tooltip-demo"
               data-toggle="tooltip"
               data-placement="bottom"
               title="MFA is NOT enabled. This is required for the Admin Page. If you have activated MFA, then
logout, login again.">
                Admin (Not Enabled)
            </a>
        </li>
    }
}
```

If the user logs in without MFA, the warning is displayed:



The user is redirected to the MFA enable view when clicking the **Admin** link:



Send MFA sign-in requirement to OpenID Connect server

The `acr_values` parameter can be used to pass the `mfa` required value from the client to the server in an authentication request.

> **NOTE**
> The `acr_values` parameter needs to be handled on the OpenID Connect server for this to work.

OpenID Connect ASP.NET Core client

The ASP.NET Core Razor Pages OpenID Connect client app uses the `AddOpenIdConnect` method to login to the OpenID Connect server. The `acr_values` parameter is set with the `mfa` value and sent with the authentication request. The `OpenIdConnectEvents` is used to add this.

For recommended `acr_values` parameter values, see Authentication Method Reference Values.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(options =>
    {
        options.DefaultScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
            OpenIdConnectDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddOpenIdConnect(options =>
    {
        options.SignInScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
        options.Authority = "<OpenID Connect server URL>";
        options.RequireHttpsMetadata = true;
        options.ClientId = "<OpenID Connect client ID>";
        options.ClientSecret = "<>";
        // Code with PKCE can also be used here
        options.ResponseType = "code id_token";
        options.Scope.Add("profile");
        options.Scope.Add("offline_access");
        options.SaveTokens = true;
        options.Events = new OpenIdConnectEvents
        {
            OnRedirectToIdentityProvider = context =>
            {
                context.ProtocolMessage.SetParameter("acr_values", "mfa");
                return Task.FromResult(0);
            }
        };
    });
```

Example OpenID Connect IdentityServer 4 server with ASP.NET Core Identity

On the OpenID Connect server, which is implemented using ASP.NET Core Identity with MVC views, a new view named *ErrorEnable2FA.cshtml* is created. The view:

- Displays if the Identity comes from an app that requires MFA but the user hasn't activated this in Identity.
- Informs the user and adds a link to activate this.

```
@{
    ViewData["Title"] = "ErrorEnable2FA";
}

<h1>The client application requires you to have MFA enabled. Enable this, try login again.</h1>

<br />

You can enable MFA to login here:

<br />

<a asp-controller="Manage" asp-action="TwoFactorAuthentication">Enable MFA</a>
```

In the `Login` method, the `IIdentityServerInteractionService` interface implementation `_interaction` is used to access the OpenID Connect request parameters. The `acr_values` parameter is accessed using the `AcrValues` property. As the client sent this with `mfa` set, this can then be checked.

If MFA is required, and the user in ASP.NET Core Identity has MFA enabled, then the login continues. When the user has no MFA enabled, the user is redirected to the custom view *ErrorEnable2FA.cshtml*. Then ASP.NET Core Identity signs the user in.

```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginInputModel model)
{
    var returnUrl = model.ReturnUrl;
    var context =
        await _interaction.GetAuthorizationContextAsync(returnUrl);
    var requires2Fa =
        context?.AcrValues.Count(t => t.Contains("mfa")) >= 1;

    var user = await _userManager.FindByNameAsync(model.Email);
    if (user != null && !user.TwoFactorEnabled && requires2Fa)
    {
        return RedirectToAction(nameof(ErrorEnable2FA));
    }

    // code omitted for brevity
```

The `ExternalLoginCallback` method works like the local Identity login. The `AcrValues` property is checked for the `mfa` value. If the `mfa` value is present, MFA is forced before the login completes (for example, redirected to the `ErrorEnable2FA` view).

```
//
// GET: /Account/ExternalLoginCallback
[HttpGet]
[AllowAnonymous]
public async Task<IActionResult> ExternalLoginCallback(
    string returnUrl = null,
    string remoteError = null)
{
    var context =
        await _interaction.GetAuthorizationContextAsync(returnUrl);
    var requires2Fa =
        context?.AcrValues.Count(t => t.Contains("mfa")) >= 1;

    if (remoteError != null)
    {
        ModelState.AddModelError(
            string.Empty,
            _sharedLocalizer["EXTERNAL_PROVIDER_ERROR",
            remoteError]);
        return View(nameof(Login));
    }
    var info = await _signInManager.GetExternalLoginInfoAsync();

    if (info == null)
    {
        return RedirectToAction(nameof(Login));
    }

    var email = info.Principal.FindFirstValue(ClaimTypes.Email);

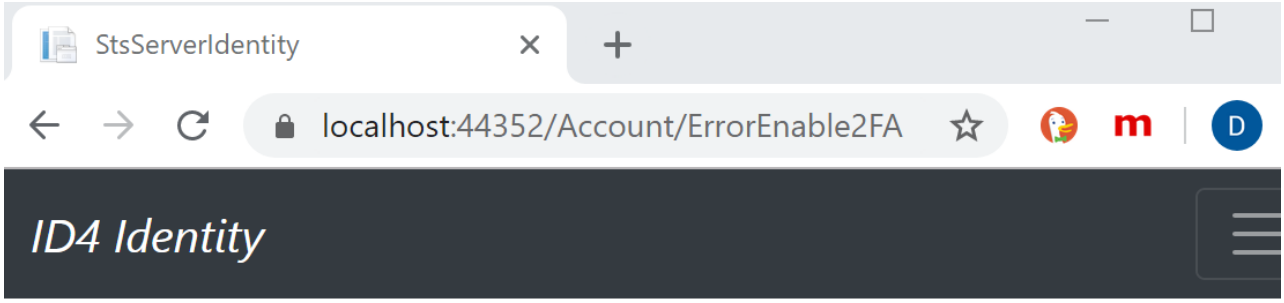
    if (!string.IsNullOrEmpty(email))
    {
        var user = await _userManager.FindByNameAsync(email);
        if (user != null && !user.TwoFactorEnabled && requires2Fa)
        {
            return RedirectToAction(nameof(ErrorEnable2FA));
        }
    }

    // Sign in the user with this external login provider if the user already has a login.
    var result = await _signInManager
        .ExternalLoginSignInAsync(
            info.LoginProvider,
            info.ProviderKey,
            isPersistent:
            false);

    // code omitted for brevity
```

If the user is already logged in, the client app:

- Still validates the `amr` claim.
- Can set up the MFA with a link to the ASP.NET Core Identity view.



Force ASP.NET Core OpenID Connect client to require MFA

This example shows how an ASP.NET Core Razor Page app, which uses OpenID Connect to sign in, can require that users have authenticated using MFA.

To validate the MFA requirement, an `IAuthorizationRequirement` requirement is created. This will be added to the pages using a policy that requires MFA.

```
using Microsoft.AspNetCore.Authorization;

namespace AspNetCoreRequireMfaOidc
{
    public class RequireMfa : IAuthorizationRequirement{}
}
```

An `AuthorizationHandler` is implemented that will use the `amr` claim and check for the value `mfa`. The `amr` is returned in the `id_token` of a successful authentication and can have many different values as defined in the Authentication Method Reference Values specification.

The returned value depends on how the identity authenticated and on the OpenID Connect server implementation.

The `AuthorizationHandler` uses the `RequireMfa` requirement and validates the `amr` claim. The OpenID Connect server can be implemented using IdentityServer4 with ASP.NET Core Identity. When a user logs in using TOTP, the `amr` claim is returned with an MFA value. If using a different OpenID Connect server implementation or a different MFA type, the `amr` claim will, or can, have a different value. The code must be extended to accept this as well.

```
using Microsoft.AspNetCore.Authorization;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace AspNetCoreRequireMfaOidc
{
    public class RequireMfaHandler : AuthorizationHandler<RequireMfa>
    {
        protected override Task HandleRequirementAsync(
            AuthorizationHandlerContext context,
            RequireMfa requirement)
        {
            if (context == null)
                throw new ArgumentNullException(nameof(context));
            if (requirement == null)
                throw new ArgumentNullException(nameof(requirement));

            var amrClaim =
                context.User.Claims.FirstOrDefault(t => t.Type == "amr");

            if (amrClaim != null && amrClaim.Value == Amr.Mfa)
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

In the `Startup.ConfigureServices` method, the `AddOpenIdConnect` method is used as the default challenge scheme. The authorization handler, which is used to check the `amr` claim, is added to the Inversion of Control container. A policy is then created which adds the `RequireMfa` requirement.

```
public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureApplicationCookie(options =>
        options.Cookie.SecurePolicy =
            CookieSecurePolicy.Always);

    services.AddSingleton<IAuthorizationHandler, RequireMfaHandler>();

    services.AddAuthentication(options =>
    {
        options.DefaultScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
            OpenIdConnectDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddOpenIdConnect(options =>
    {
        options.SignInScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
        options.Authority = "https://localhost:44352";
        options.RequireHttpsMetadata = true;
        options.ClientId = "AspNetCoreRequireMfaOidc";
        options.ClientSecret = "AspNetCoreRequireMfaOidcSecret";
        options.ResponseType = "code id_token";
        options.Scope.Add("profile");
        options.Scope.Add("offline_access");
        options.SaveTokens = true;
    });

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireMfa", policyIsAdminRequirement =>
        {
            policyIsAdminRequirement.Requirements.Add(new RequireMfa());
        });
    });

    services.AddRazorPages();
}
```

This policy is then used in the Razor page as required. The policy could be added globally for the entire app as well.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace AspNetCoreRequireMfaOidc.Pages
{
    [Authorize(Policy= "RequireMfa")]
    public class IndexModel : PageModel
    {
        private readonly ILogger<IndexModel> _logger;

        public IndexModel(ILogger<IndexModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

If the user authenticates without MFA, the `amr` claim will probably have a `pwd` value. The request won't be authorized to access the page. Using the default values, the user will be redirected to the *Account/AccessDenied* page. This behavior can be changed or you can implement your own custom logic here. In this example, a link is added so that the valid user can set up MFA for their account.

```
@page
@model AspNetCoreRequireMfaOidc.AccessDeniedModel
@{
    ViewData["Title"] = "AccessDenied";
    Layout = "~/Pages/Shared/_Layout.cshtml";
}

<h1>AccessDenied</h1>

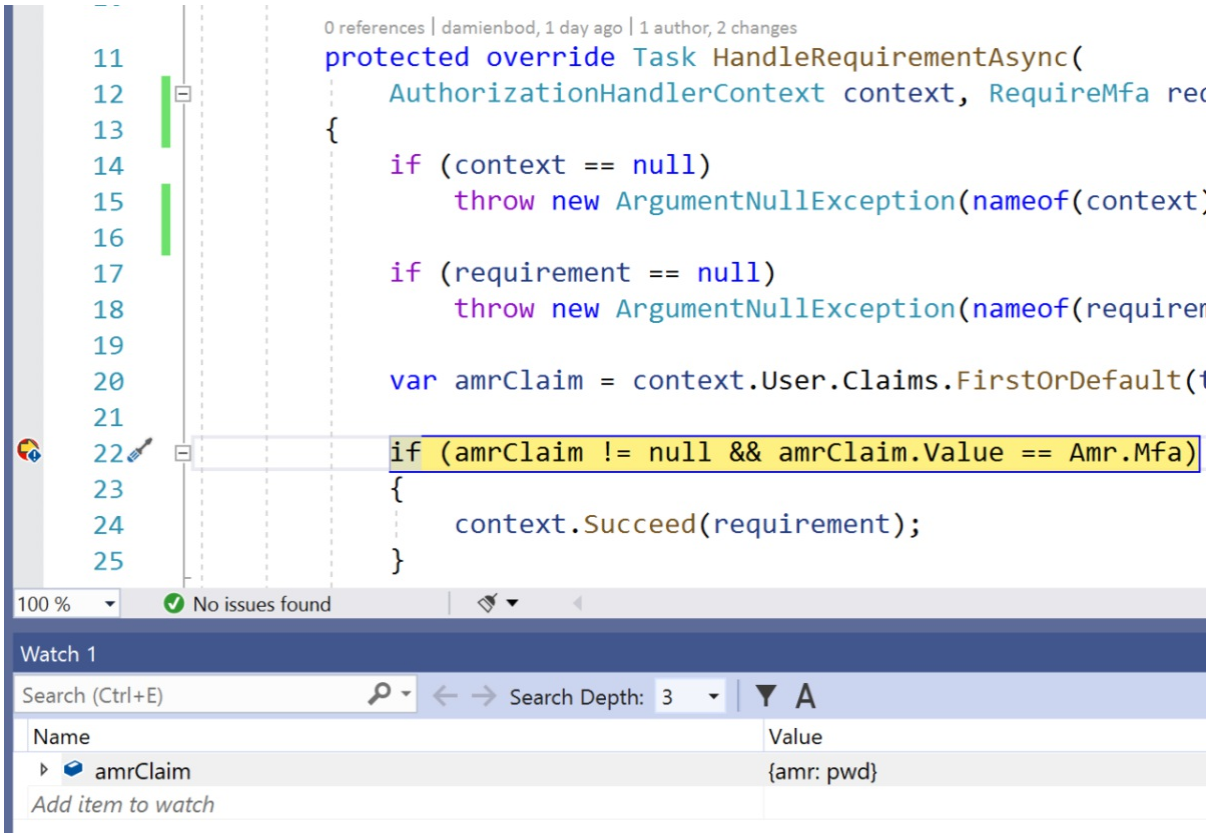
You require MFA to login here

<a href="https://localhost:44352/Manage/TwoFactorAuthentication">Enable MFA</a>
```

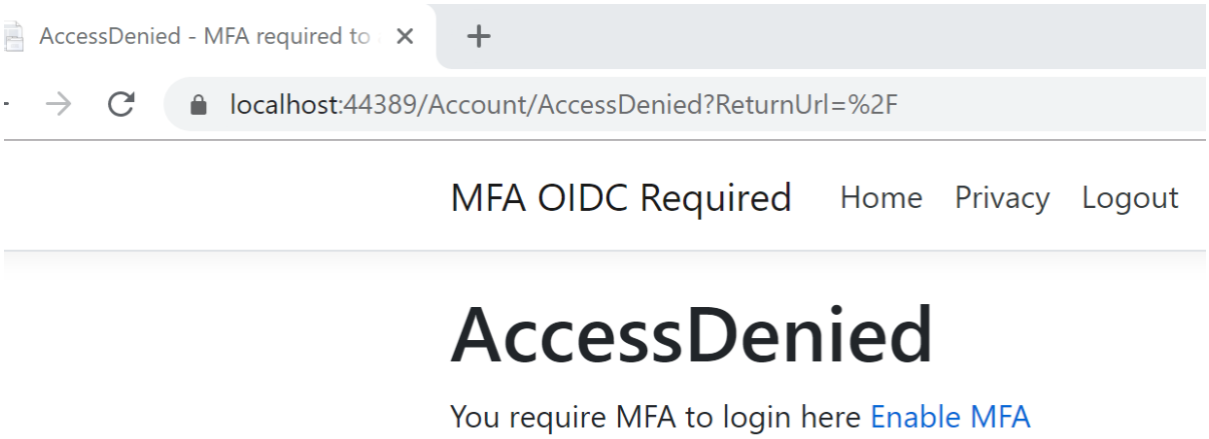
Now only users that authenticate with MFA can access the page or website. If different MFA types are used or if 2FA is okay, the `amr` claim will have different values and needs to be processed correctly. Different OpenID Connect servers also return different values for this claim and might not follow the Authentication Method Reference Values specification.

When logging in without MFA (for example, using just a password):

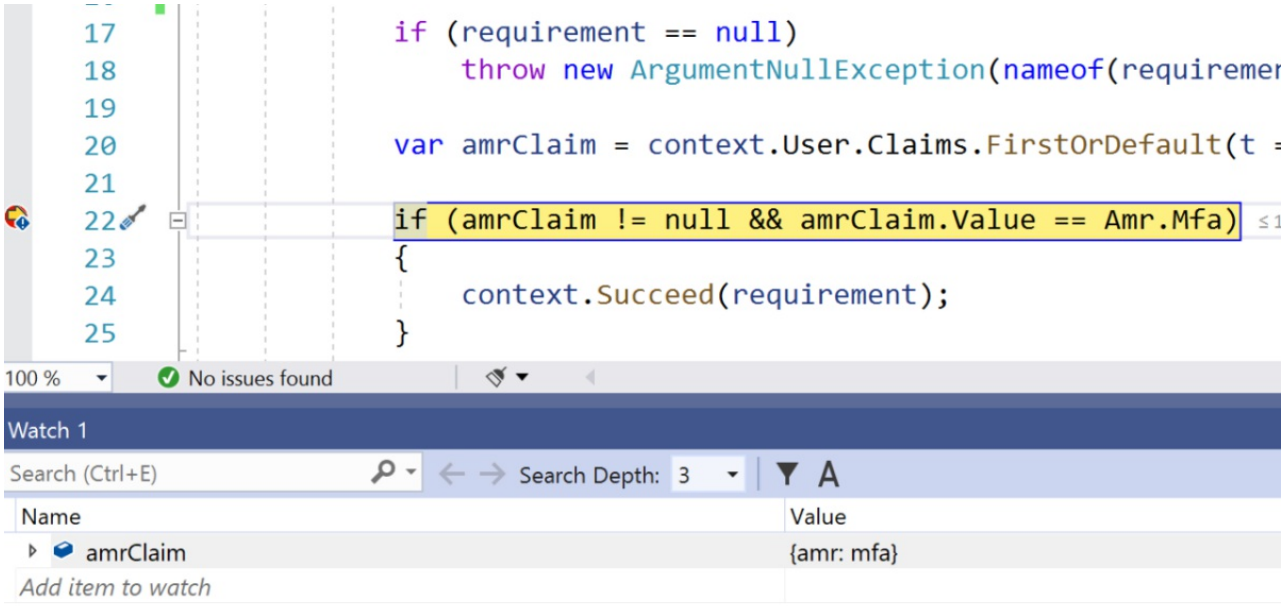
- The `amr` has the `pwd` value:



- Access is denied:



Alternatively, logging in using OTP with Identity:



Additional resources

- Enable QR Code generation for TOTP authenticator apps in ASP.NET Core
- Passwordless authentication options for Azure Active Directory
- FIDO2 .NET library for FIDO2 / WebAuthn Attestation and Assertion using .NET
- WebAuthn Awesome

Introduction to authorization in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create a document library, add documents, edit documents, and delete them. A non-administrative user working with the library is only authorized to read the documents.

Authorization is orthogonal and independent from authentication. However, authorization requires an authentication mechanism. Authentication is the process of ascertaining who a user is. Authentication may create one or more identities for the current user.

For more information about authentication in ASP.NET Core, see Overview of ASP.NET Core Authentication.

Authorization types

ASP.NET Core authorization provides a simple, declarative role and a rich policy-based model. Authorization is expressed in requirements, and handlers evaluate a user's claims against requirements. Imperative checks can be based on simple policies or policies which evaluate both the user identity and properties of the resource that the user is attempting to access.

Namespaces

Authorization components, including the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes, are found in the `Microsoft.AspNetCore.Authorization` namespace.

Consult the documentation on simple authorization.

Create an ASP.NET Core web app with user data protected by authorization

9/22/2020 • 39 minutes to read • Edit Online

By Rick Anderson and Joe Audette

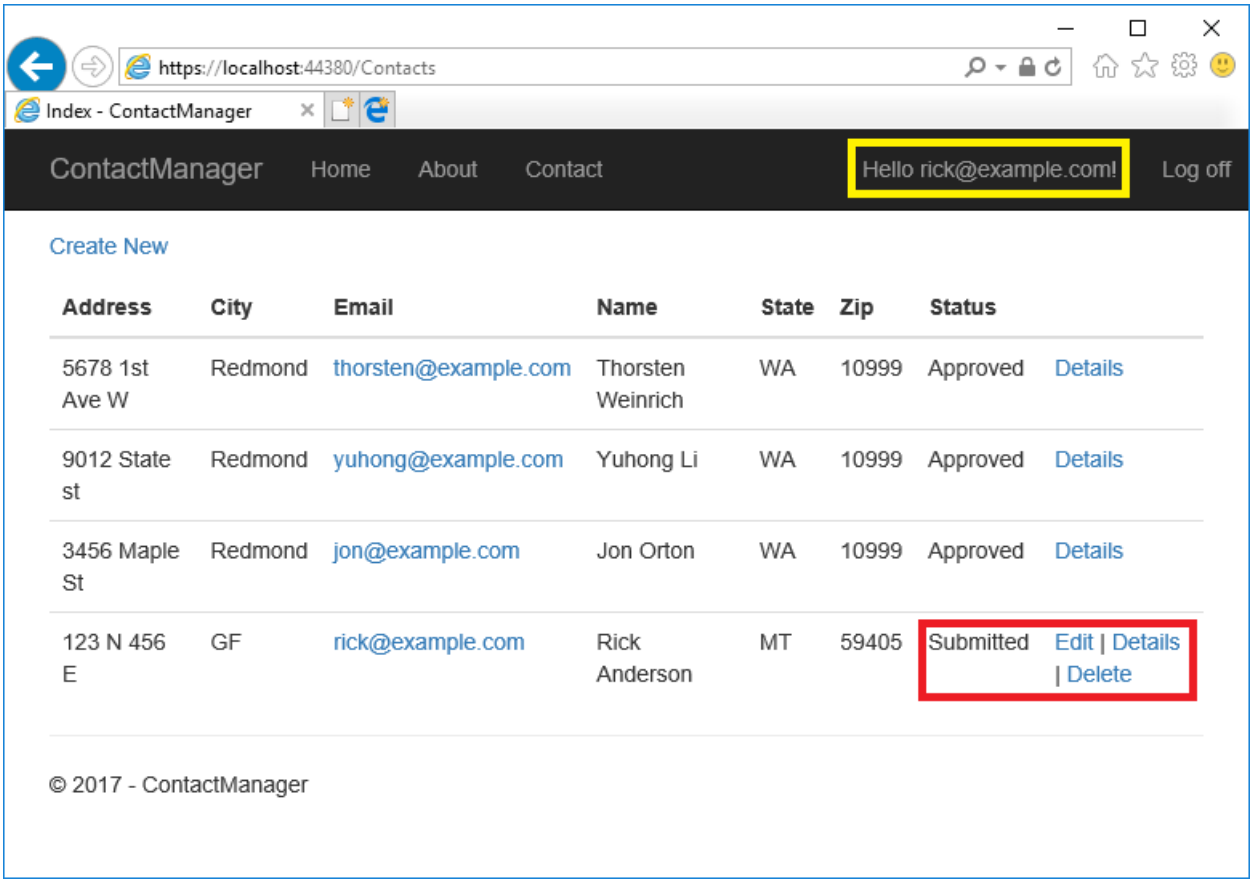
See this pdf

This tutorial shows how to create an ASP.NET Core web app with user data protected by authorization. It displays a list of contacts that authenticated (registered) users have created. There are three security groups:

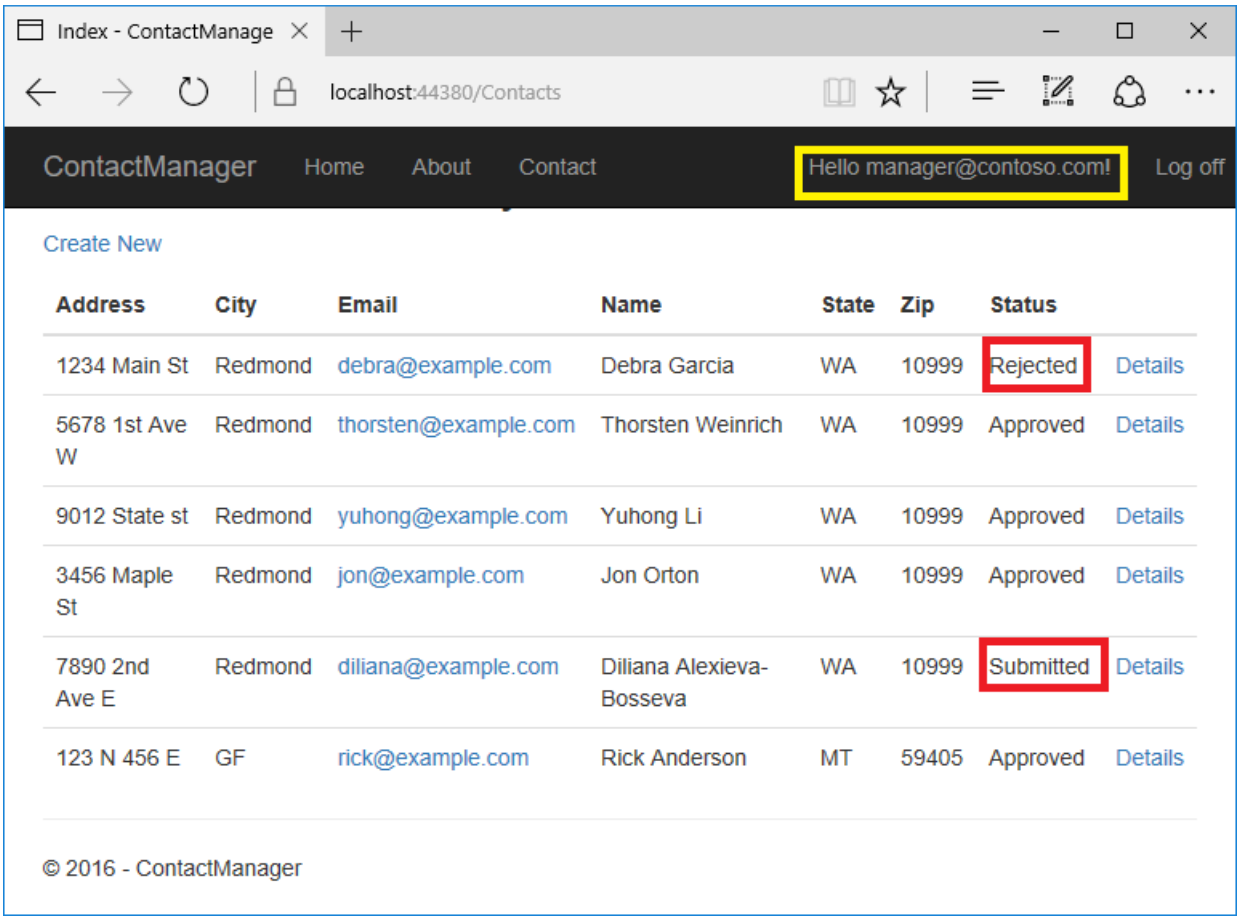
- **Registered users** can view all the approved data and can edit/delete their own data.
- **Managers** can approve or reject contact data. Only approved contacts are visible to users.
- **Administrators** can approve/reject and edit/delete any data.

The images in this document don't exactly match the latest templates.

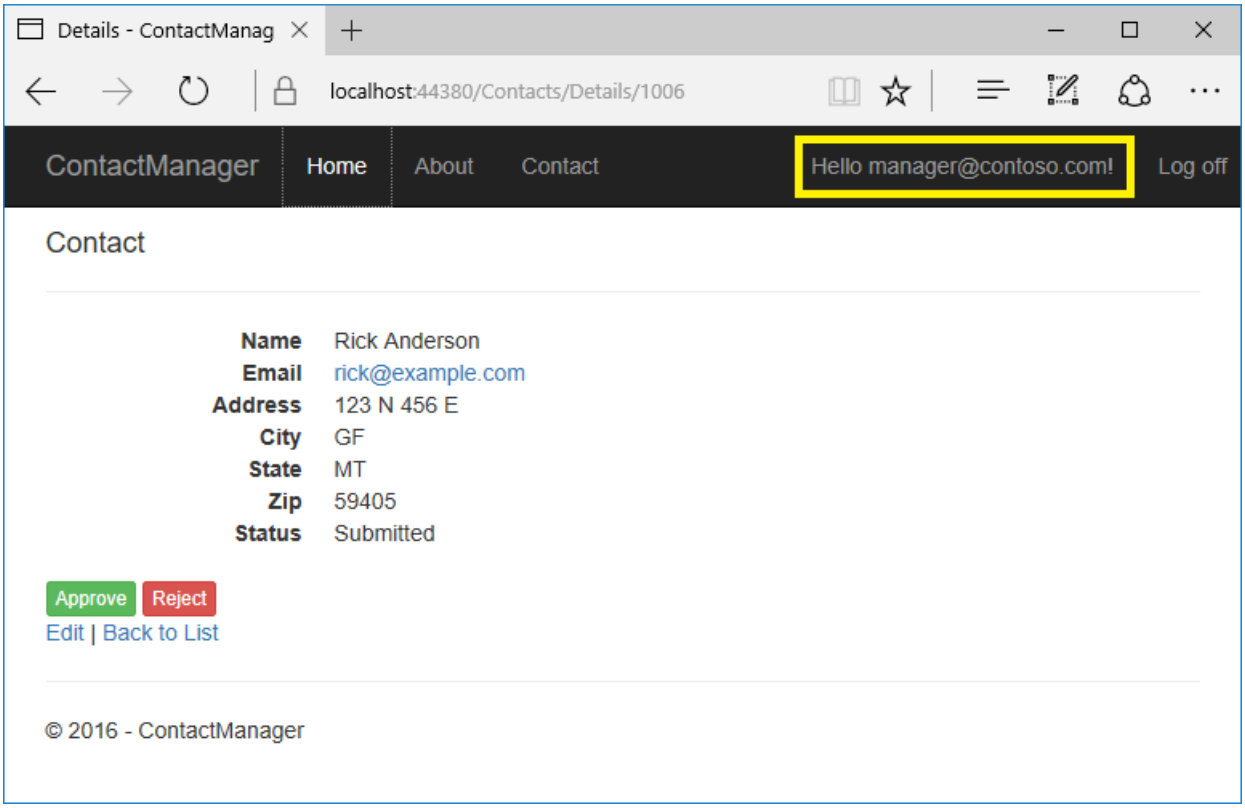
In the following image, user Rick (`rick@example.com`) is signed in. Rick can only view approved contacts and **Edit/Delete/Create New** links for his contacts. Only the last record, created by Rick, displays **Edit** and **Delete** links. Other users won't see the last record until a manager or administrator changes the status to "Approved".



In the following image, `manager@contoso.com` is signed in and in the manager's role:

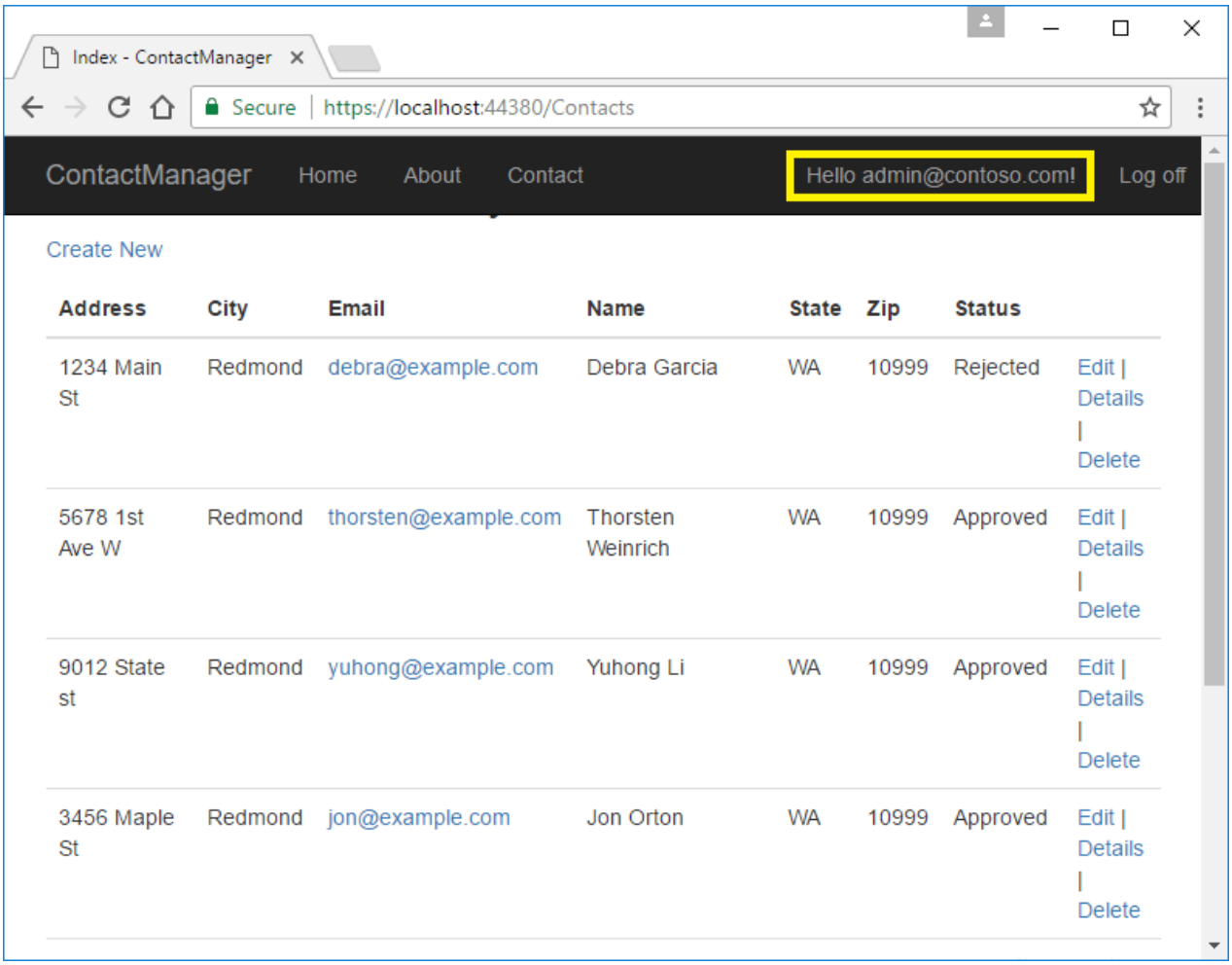


The following image shows the managers details view of a contact:



The **Approve** and **Reject** buttons are only displayed for managers and administrators.

In the following image, `admin@contoso.com` is signed in and in the administrator's role:



The administrator has all privileges. She can read/edit/delete any contact and change the status of contacts.

The app was created by scaffolding the following `Contact` model:

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

The sample contains the following authorization handlers:

- `ContactIsOwnerAuthorizationHandler`: Ensures that a user can only edit their data.
- `ContactManagerAuthorizationHandler`: Allows managers to approve or reject contacts.
- `ContactAdministratorsAuthorizationHandler`: Allows administrators to approve or reject contacts and to edit/delete contacts.

Prerequisites

This tutorial is advanced. You should be familiar with:

- ASP.NET Core
- Authentication

- Account Confirmation and Password Recovery
- Authorization
- Entity Framework Core

The starter and completed app

Download the completed app. Test the completed app so you become familiar with its security features.

The starter app

Download the starter app.

Run the app, tap the **ContactManager** link, and verify you can create, edit, and delete a contact.

Secure user data

The following sections have all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Tie the contact data to the user

Use the ASP.NET Identity user ID to ensure users can edit their data, but not other users data. Add `OwnerID` and `ContactStatus` to the `Contact` model:

```
public class Contact
{
    public int ContactId { get; set; }

    // user ID from AspNetUser table.
    public string OwnerID { get; set; }

    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

`OwnerID` is the user's ID from the `AspNetUser` table in the Identity database. The `Status` field determines if a contact is viewable by general users.

Create a new migration and update the database:

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

Add Role services to Identity

Append AddRoles to add Role services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

Require authenticated users

Set the fallback authentication policy to require users to be authenticated:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddRazorPages();

    services.AddAuthorization(options =>
    {
        options.FallbackPolicy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
    });
```

The preceding highlighted code sets the fallback authentication policy. The fallback authentication policy requires ***all*** users to be authenticated, except for Razor Pages, controllers, or action methods with an authentication attribute. For example, Razor Pages, controllers, or action methods with `[AllowAnonymous]` or `[Authorize(PolicyName="MyPolicy")]` use the applied authentication attribute rather than the fallback authentication policy.

The fallback authentication policy:

- Is applied to all requests that do not explicitly specify an authentication policy. For requests served by endpoint routing, this would include any endpoint that does not specify an authorization attribute. For requests served by other middleware after the authorization middleware, such as static files, this would apply the policy to all requests.

Setting the fallback authentication policy to require users to be authenticated protects newly added Razor Pages and controllers. Having authentication required by default is more secure than relying on new controllers and Razor Pages to include the `[Authorize]` attribute.

The AuthorizationOptions class also contains AuthorizationOptions.DefaultPolicy. The `DefaultPolicy` is the policy used with the `[Authorize]` attribute when no policy is specified. `[Authorize]` doesn't contain a named policy, unlike `[Authorize(PolicyName="MyPolicy")]`.

For more information on policies, see Policy-based authorization in ASP.NET Core.

An alternative way for MVC controllers and Razor Pages to require all users be authenticated is adding an authorization filter:

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddRazorPages();

    services.AddControllers(config =>
    {
        // using Microsoft.AspNetCore.Mvc.Authorization;
        // using Microsoft.AspNetCore.Authorization;
        var policy = new AuthorizationPolicyBuilder()
                         .RequireAuthenticatedUser()
                         .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    });
```

The preceding code uses an authorization filter, setting the fallback policy uses endpoint routing. Setting the fallback policy is the preferred way to require all users be authenticated.

Add AllowAnonymous to the `Index` and `Privacy` pages so anonymous users can get information about the site before they register:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace ContactManager.Pages
{
    [AllowAnonymous]
    public class IndexModel : PageModel
    {
        private readonly ILogger<IndexModel> _logger;

        public IndexModel(ILogger<IndexModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {

        }
    }
}
```

Configure the test account

The `SeedData` class creates two accounts: administrator and manager. Use the Secret Manager tool to set a password for these accounts. Set the password from the project directory (the directory containing *Program.cs*):

```
dotnet user-secrets set SeedUserPW <PW>
```

If a strong password is not specified, an exception is thrown when `SeedData.Initialize` is called.

Update `Main` to use the test password:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<ApplicationDbContext>();
                context.Database.Migrate();

                // requires using Microsoft.Extensions.Configuration;
                var config = host.Services.GetRequiredService<IConfiguration>();
                // Set password with the Secret Manager tool.
                // dotnet user-secrets set SeedUserPW <pw>

                var testUserPw = config["SeedUserPW"];

                SeedData.Initialize(services, testUserPw).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred seeding the DB.");
            }
        }

        host.Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Create the test accounts and update the contacts

Update the `Initialize` method in the `SeedData` class to create the test accounts:

```
public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
{
    using (var context = new ApplicationDbContext(
        serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
    {
        // For sample purposes seed both with the same password.
        // Password is set with the following:
        // dotnet user-secrets set SeedUserPW <pw>
        // The admin user can do anything

        var adminID = await EnsureUser(serviceProvider, testUserPw, "admin@contoso.com");
        await EnsureRole(serviceProvider, adminID, Constants.ContactAdministratorsRole);

        // allowed user can create and edit contacts that they create
        var managerID = await EnsureUser(serviceProvider, testUserPw, "manager@contoso.com");
        await EnsureRole(serviceProvider, managerID, Constants.ContactManagersRole);

        SeedDB(context, adminID);
    }
}
```

```
private static async Task<string> EnsureUser(IServiceProvider serviceProvider,
                                             string testUserPw, string UserName)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByNameAsync(UserName);
    if (user == null)
    {
        user = new IdentityUser {
            UserName = UserName,
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, testUserPw);
    }

    if (user == null)
    {
        throw new Exception("The password is probably not strong enough!");
    }

    return user.Id;
}

private static async Task<IdentityResult> EnsureRole(IServiceProvider serviceProvider,
                                                     string uid, string role)
{
    IdentityResult IR = null;
    var roleManager = serviceProvider.GetService<RoleManager<IdentityRole>>();

    if (roleManager == null)
    {
        throw new Exception("roleManager null");
    }

    if (!await roleManager.RoleExistsAsync(role))
    {
        IR = await roleManager.CreateAsync(new IdentityRole(role));
    }

    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByIdAsync(uid);

    if(user == null)
    {
        throw new Exception("The testUserPw password was probably not strong enough!");
    }

    IR = await userManager.AddToRoleAsync(user, role);

    return IR;
}
```

Add the administrator user ID and `ContactStatus` to the contacts. Make one of the contacts "Submitted" and one "Rejected". Add the user ID and status to all the contacts. Only one contact is shown:

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
    {
        return;   // DB has been seeded
    }

    context.Contact.AddRange(
        new Contact
        {
            Name = "Debra Garcia",
            Address = "1234 Main St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "debra@example.com",
            Status = ContactStatus.Approved,
            OwnerID = adminID
        },
```

Create owner, manager, and administrator authorization handlers

Create a `ContactIsOwnerAuthorizationHandler` class in the *Authorization* folder. The `ContactIsOwnerAuthorizationHandler` verifies that the user acting on a resource owns the resource.

```
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;
using System.Threading.Tasks;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
                : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        UserManager<IdentityUser> _userManager;

        public ContactIsOwnerAuthorizationHandler(UserManager<IdentityUser>
            userManager)
        {
            _userManager = userManager;
        }

        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for CRUD permission, return.

            if (requirement.Name != Constants.CreateOperationName &&
                requirement.Name != Constants.ReadOperationName   &&
                requirement.Name != Constants.UpdateOperationName &&
                requirement.Name != Constants.DeleteOperationName )
            {
                return Task.CompletedTask;
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

The `ContactIsOwnerAuthorizationHandler` calls context.Succeed if the current authenticated user is the contact owner. Authorization handlers generally:

- Return `context.Succeed` when the requirements are met.
- Return `Task.CompletedTask` when requirements aren't met. `Task.CompletedTask` is not success or failure—it allows other authorization handlers to run.

If you need to explicitly fail, return context.Fail.

The app allows contact owners to edit/delete/create their own data. `ContactIsOwnerAuthorizationHandler` doesn't need to check the operation passed in the requirement parameter.

Create a manager authorization handler

Create a `ContactManagerAuthorizationHandler` class in the *Authorization* folder. The

`ContactManagerAuthorizationHandler` verifies the user acting on the resource is a manager. Only managers can approve or reject content changes (new or changed).

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.CompletedTask;
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Create an administrator authorization handler

Create a `ContactAdministratorsAuthorizationHandler` class in the *Authorization* folder. The `ContactAdministratorsAuthorizationHandler` verifies the user acting on the resource is an administrator. Administrator can do all operations.

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
                    : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
                                            AuthorizationHandlerContext context,
                                    OperationAuthorizationRequirement requirement,
                                     Contact resource)
        {
            if (context.User == null)
            {
                return Task.CompletedTask;
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Register the authorization handlers

Services using Entity Framework Core must be registered for dependency injection using AddScoped. The `ContactIsOwnerAuthorizationHandler` uses ASP.NET Core Identity, which is built on Entity Framework Core. Register the handlers with the service collection so they're available to the `ContactsController` through dependency injection. Add the following code to the end of `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(
        options => options.SignIn.RequireConfirmedAccount = true)
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddRazorPages();

    services.AddAuthorization(options =>
    {
        options.FallbackPolicy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
    });

    // Authorization handlers.
    services.AddScoped<IAuthorizationHandler,
                          ContactIsOwnerAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
                          ContactAdministratorsAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
                          ContactManagerAuthorizationHandler>();
}
```

`ContactAdministratorsAuthorizationHandler` and `ContactManagerAuthorizationHandler` are added as singletons. They're singletons because they don't use EF and all the information needed is in the `Context` parameter of the `HandleRequirementAsync` method.

Support authorization

In this section, you update the Razor Pages and add an operations requirements class.

Review the contact operations requirements class

Review the `ContactOperations` class. This class contains the requirements the app supports:

```
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
          new OperationAuthorizationRequirement {Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
          new OperationAuthorizationRequirement {Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
          new OperationAuthorizationRequirement {Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
          new OperationAuthorizationRequirement {Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
          new OperationAuthorizationRequirement {Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
          new OperationAuthorizationRequirement {Name=Constants.RejectOperationName};
    }

    public class Constants
    {
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =
                                                          "ContactAdministrators";
        public static readonly string ContactManagersRole = "ContactManagers";
    }
}
```

Create a base class for the Contacts Razor Pages

Create a base class that contains the services used in the contacts Razor Pages. The base class puts the initialization code in one location:

```
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages.Contacts
{
    public class DI_BasePageModel : PageModel
    {
        protected ApplicationDbContext Context { get; }
        protected IAuthorizationService AuthorizationService { get; }
        protected UserManager<IdentityUser> UserManager { get; }

        public DI_BasePageModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<IdentityUser> userManager) : base()
        {
            Context = context;
            UserManager = userManager;
            AuthorizationService = authorizationService;
        }
    }
}
```

The preceding code:

- Adds the `IAuthorizationService` service to access to the authorization handlers.
- Adds the Identity `UserManager` service.
- Add the `ApplicationDbContext`.

Update the CreateModel

Update the create page model constructor to use the `DI_BasePageModel` base class:

```
public class CreateModel : DI_BasePageModel
{
    public CreateModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }
```

Update the `CreateModel.OnPostAsync` method to:

- Add the user ID to the `Contact` model.
- Call the authorization handler to verify the user has permission to create contacts.

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    Contact.OwnerID = UserManager.GetUserId(User);

    // requires using ContactManager.Authorization;
    var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                            User, Contact,
                                            ContactOperations.Create);
    if (!isAuthorized.Succeeded)
    {
        return Forbid();
    }

    Context.Contact.Add(Contact);
    await Context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Update the IndexModel

Update the `OnGetAsync` method so only approved contacts are shown to general users:

```
public class IndexModel : DI_BasePageModel
{
    public IndexModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public IList<Contact> Contact { get; set; }

    public async Task OnGetAsync()
    {
        var contacts = from c in Context.Contact
                       select c;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        // Only approved contacts are shown UNLESS you're authorized to see them
        // or you are the owner.
        if (!isAuthorized)
        {
            contacts = contacts.Where(c => c.Status == ContactStatus.Approved
                                      || c.OwnerID == currentUserId);
        }

        Contact = await contacts.ToListAsync();
    }
}
```

Update the EditModel

Add an authorization handler to verify the user owns the contact. Because resource authorization is being validated, the `[Authorize]` attribute is not enough. The app doesn't have access to the resource when attributes are evaluated. Resource-based authorization must be imperative. Checks must be performed once the app has access to the resource, either by loading it in the page model or by loading it within the handler itself. You frequently access the resource by passing in the resource key.

```
public class EditModel : DI_BasePageModel
{
    public EditModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
                                            m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }
```

```
        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                 User, Contact,
                                                 ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        // Fetch Contact from DB to get OwnerID.
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, contact,
                                                ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        Contact.OwnerID = contact.OwnerID;

        Context.Attach(Contact).State = EntityState.Modified;

        if (Contact.Status == ContactStatus.Approved)
        {
            // If the contact is updated after approval,
            // and the user cannot approve,
            // set the status back to submitted so the update can be
            // checked and approved.
            var canApprove = await AuthorizationService.AuthorizeAsync(User,
                                    Contact,
                                    ContactOperations.Approve);

            if (!canApprove.Succeeded)
            {
                Contact.Status = ContactStatus.Submitted;
            }
        }

        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Update the DeleteModel

Update the delete page model to use the authorization handler to verify the user has delete permission on the contact.

```
public class DeleteModel : DI_BasePageModel
{
    public DeleteModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
                                          m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                               User, Contact,
                                               ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                               User, contact,
                                               ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        Context.Contact.Remove(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Inject the authorization service into the views

Currently, the UI shows edit and delete links for contacts the user can't modify.

Inject the authorization service in the *Pages/_ViewImports.cshtml* file so it's available to all views:

```
@using Microsoft.AspNetCore.Identity
@using ContactManager
@using ContactManager.Data
@namespace ContactManager.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using ContactManager.Authorization;
@using Microsoft.AspNetCore.Authorization
@using ContactManager.Models
@inject IAuthorizationService AuthorizationService
```

The preceding markup adds several `using` statements.

Update the **Edit** and **Delete** links in *Pages/Contacts/Index.cshtml* so they're only rendered for users with the appropriate permissions:

```
@page
@model ContactManager.Pages.Contacts.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Address)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].City)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].State)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Zip)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Email)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Status)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Contact)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Address)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.City)
```

```
                @Html.DisplayFor(modelItem => item.City)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.State)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Email)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Status)
            </td>
            <td>
                @if ((await AuthorizationService.AuthorizeAsync(
                 User, item,
                 ContactOperations.Update)).Succeeded)
                {
                    <a asp-page="./Edit" asp-route-id="@item.ContactId">Edit</a>
                    <text> | </text>
                }

                <a asp-page="./Details" asp-route-id="@item.ContactId">Details</a>

                @if ((await AuthorizationService.AuthorizeAsync(
                 User, item,
                 ContactOperations.Delete)).Succeeded)
                {
                    <text> | </text>
                    <a asp-page="./Delete" asp-route-id="@item.ContactId">Delete</a>
                }
            </td>
        </tr>
    }
</tbody>
</table>
```

> **WARNING**
>
> Hiding links from users that don't have permission to change data doesn't secure the app. Hiding links makes the app more user-friendly by displaying only valid links. Users can hack the generated URLs to invoke edit and delete operations on data they don't own. The Razor Page or controller must enforce access checks to secure the data.

Update Details

Update the details view so managers can approve or reject contacts:

```
        @*Precedng markup omitted for brevity.*@
        <dt>
            @Html.DisplayNameFor(model => model.Contact.Email)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Email)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Contact.Status)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Status)
        </dd>
    </dl>
</div>

@if (Model.Contact.Status != ContactStatus.Approved)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Approve)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-success">Approve</button>
        </form>
    }
}

@if (Model.Contact.Status != ContactStatus.Rejected)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Reject)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-danger">Reject</button>
        </form>
    }
}

<div>
    @if ((await AuthorizationService.AuthorizeAsync(
         User, Model.Contact,
         ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-id="@Model.Contact.ContactId">Edit</a>
        <text> | </text>
    }
    <a asp-page="./Index">Back to List</a>
</div>
```

Update the details page model:

```
public class DetailsModel : DI_BasePageModel
{
    public DetailsModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            && currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id, ContactStatus status)
    {
        var contact = await Context.Contact.FirstOrDefaultAsync(
                                                m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var contactOperation = (status == ContactStatus.Approved)
                                                   ? ContactOperations.Approve
                                                   : ContactOperations.Reject;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(User, contact,
                                   contactOperation);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }
        contact.Status = status;
        Context.Contact.Update(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Add or remove a user to a role

See this issue for information on:

- Removing privileges from a user. For example, muting a user in a chat app.
- Adding privileges to a user.

Differences between Challenge and Forbid

This app sets the default policy to require authenticated users. The following code allows anonymous users. Anonymous users are allowed to show the differences between Challenge vs Forbid.

```
[AllowAnonymous]
public class Details2Model : DI_BasePageModel
{
    public Details2Model(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        if (!User.Identity.IsAuthenticated)
        {
            return Challenge();
        }

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            && currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return Forbid();
        }

        return Page();
    }
}
```

In the preceding code:

- When the user is **not** authenticated, a `ChallengeResult` is returned. When a `ChallengeResult` is returned, the user is redirected to the sign-in page.
- When the user is authenticated, but not authorized, a `ForbidResult` is returned. When a `ForbidResult` is returned, the user is redirected to the access denied page.

Test the completed app

If you haven't already set a password for seeded user accounts, use the Secret Manager tool to set a password:

- Choose a strong password: Use eight or more characters and at least one upper-case character, number, and symbol. For example, `Passw0rd!` meets the strong password requirements.
- Execute the following command from the project's folder, where `<PW>` is the password:

```
dotnet user-secrets set SeedUserPW <PW>
```

If the app has contacts:

- Delete all of the records in the `Contact` table.
- Restart the app to seed the database.

An easy way to test the completed app is to launch three different browsers (or incognito/InPrivate sessions). In one browser, register a new user (for example, `test@example.com`). Sign in to each browser with a different user. Verify the following operations:

- Registered users can view all of the approved contact data.
- Registered users can edit/delete their own data.
- Managers can approve/reject contact data. The `Details` view shows **Approve** and **Reject** buttons.
- Administrators can approve/reject and edit/delete all data.

USER	SEEDED BY THE APP	OPTIONS
test@example.com	No	Edit/delete the own data.
manager@contoso.com	Yes	Approve/reject and edit/delete own data.
admin@contoso.com	Yes	Approve/reject and edit/delete all data.

Create a contact in the administrator's browser. Copy the URL for delete and edit from the administrator contact. Paste these links into the test user's browser to verify the test user can't perform these operations.

Create the starter app

- Create a Razor Pages app named "ContactManager"
  - Create the app with **Individual User Accounts**.
  - Name it "ContactManager" so the namespace matches the namespace used in the sample.
  - `-uld` specifies LocalDB instead of SQLite

```
dotnet new webapp -o ContactManager -au Individual -uld
```

- Add *Models/Contact.cs*:

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

- Scaffold the `Contact` model.
- Create initial migration and update the database:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet aspnet-codegenerator razorpage -m Contact -udl -dc ApplicationDbContext -outDir Pages\Contacts --
referenceScriptLibraries
dotnet ef database drop -f
dotnet ef migrations add initial
dotnet ef database update
```

If you experience a bug with the `dotnet aspnet-codegenerator razorpage` command, see this GitHub issue.

- Update the **ContactManager** anchor in the *Pages/Shared/_Layout.cshtml* file:

```
<a class="navbar-brand" asp-area="" asp-page="/Contacts/Index">ContactManager</a>
```

- Test the app by creating, editing, and deleting a contact

Seed the database

Add the SeedData class to the *Data* folder:

```
using ContactManager.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Linq;
using System.Threading.Tasks;

// dotnet aspnet-codegenerator razorpage -m Contact -dc ApplicationDbContext -udl -outDir Pages\Contacts --
referenceScriptLibraries

namespace ContactManager.Data
{
    public static class SeedData
    {
        public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
        {
            using (var context = new ApplicationDbContext(
                serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
            {
                SeedDB(context, "0");
            }
        }

        public static void SeedDB(ApplicationDbContext context, string adminID)
        {
            if (context.Contact.Any())
```

```
            {
                return;   // DB has been seeded
            }

            context.Contact.AddRange(
                new Contact
                {
                    Name = "Debra Garcia",
                    Address = "1234 Main St",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "debra@example.com"
                },
                new Contact
                {
                    Name = "Thorsten Weinrich",
                    Address = "5678 1st Ave W",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "thorsten@example.com"
                },
             new Contact
             {
                 Name = "Yuhong Li",
                 Address = "9012 State st",
                 City = "Redmond",
                 State = "WA",
                 Zip = "10999",
                 Email = "yuhong@example.com"
             },
             new Contact
             {
                 Name = "Jon Orton",
                 Address = "3456 Maple St",
                 City = "Redmond",
                 State = "WA",
                 Zip = "10999",
                 Email = "jon@example.com"
             },
             new Contact
             {
                 Name = "Diliana Alexieva-Bosseva",
                 Address = "7890 2nd Ave E",
                 City = "Redmond",
                 State = "WA",
                 Zip = "10999",
                 Email = "diliana@example.com"
             }
             );
            context.SaveChanges();
        }

    }
}
```

Call `SeedData.Initialize` from `Main`:

```
using ContactManager.Data;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContactManager
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                    var context = services.GetRequiredService<ApplicationDbContext>();
                    context.Database.Migrate();
                    SeedData.Initialize(services, "not used");
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }

            host.Run();
        }

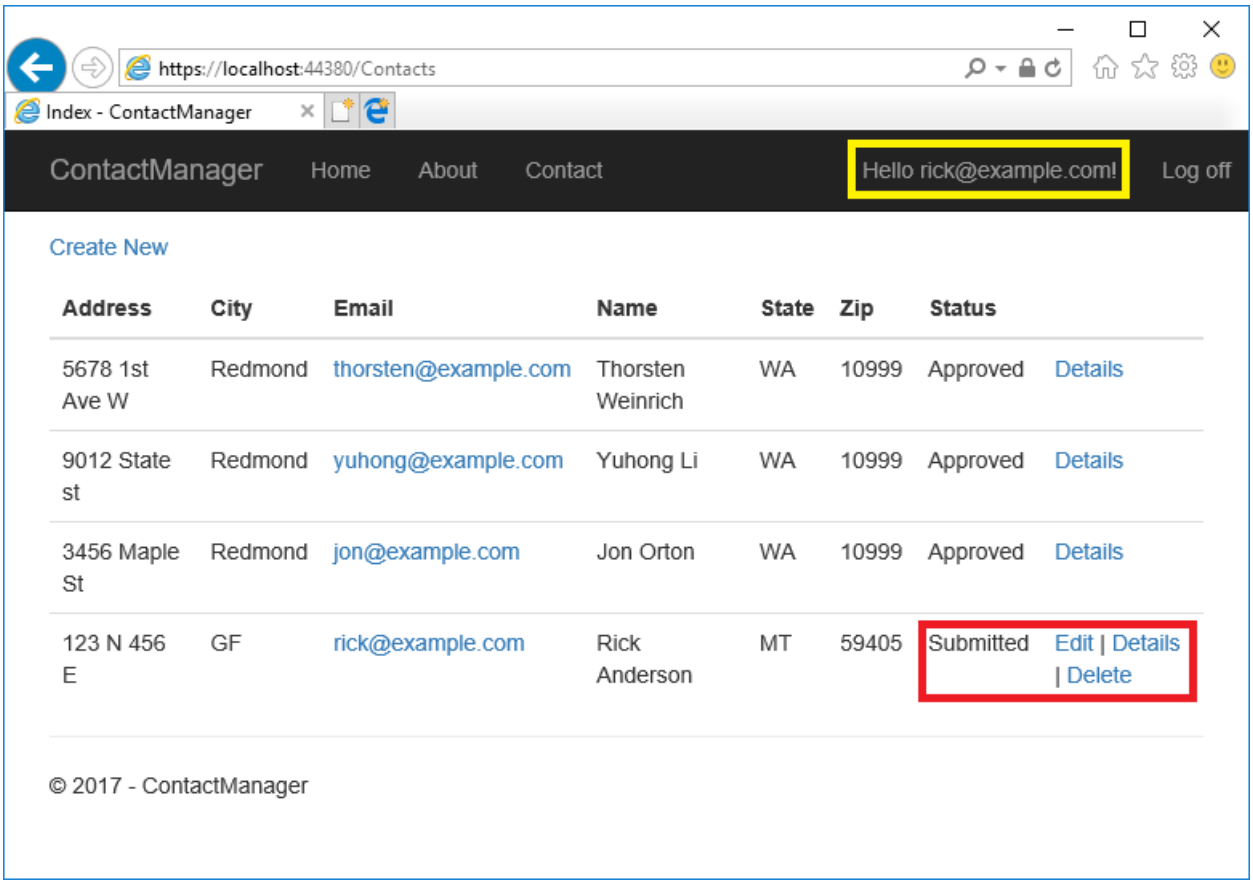
        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

Test that the app seeded the database. If there are any rows in the contact DB, the seed method doesn't run.

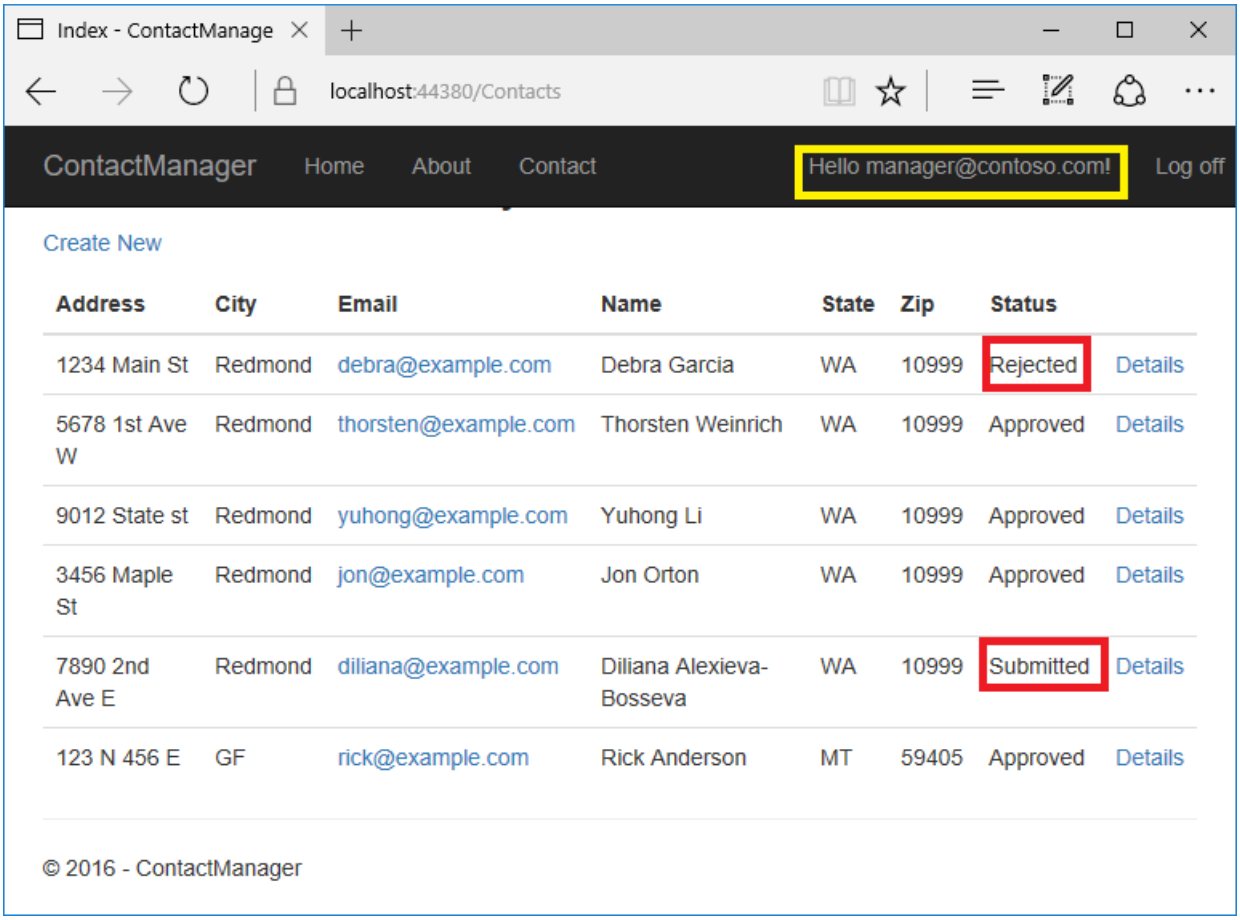
This tutorial shows how to create an ASP.NET Core web app with user data protected by authorization. It displays a list of contacts that authenticated (registered) users have created. There are three security groups:

- **Registered users** can view all the approved data and can edit/delete their own data.
- **Managers** can approve or reject contact data. Only approved contacts are visible to users.
- **Administrators** can approve/reject and edit/delete any data.

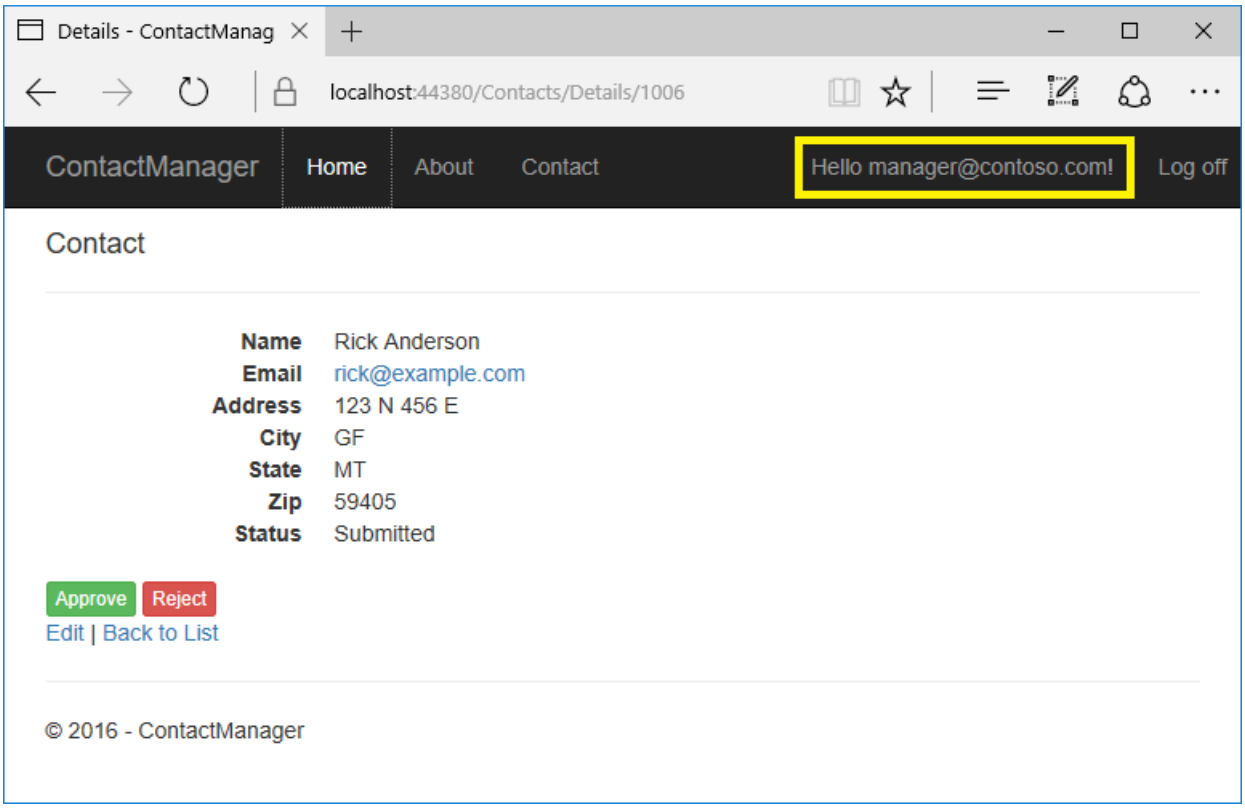
In the following image, user Rick (`rick@example.com`) is signed in. Rick can only view approved contacts and **Edit**/**Delete**/**Create New** links for his contacts. Only the last record, created by Rick, displays **Edit** and **Delete** links. Other users won't see the last record until a manager or administrator changes the status to "Approved".



In the following image, `manager@contoso.com` is signed in and in the manager's role:

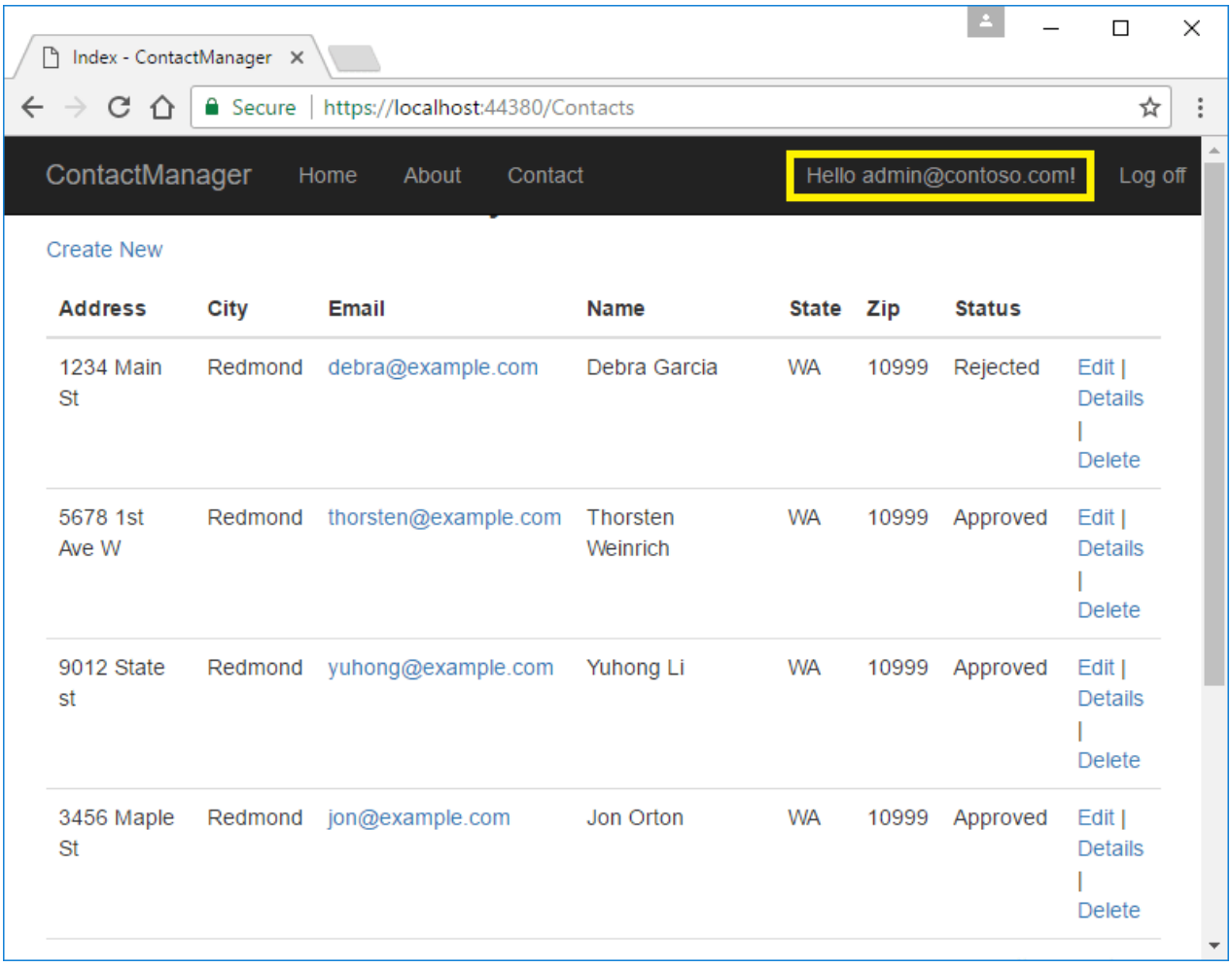


The following image shows the managers details view of a contact:



The **Approve** and **Reject** buttons are only displayed for managers and administrators.

In the following image, `admin@contoso.com` is signed in and in the administrator's role:



The administrator has all privileges. She can read/edit/delete any contact and change the status of contacts.

The app was created by scaffolding the following `Contact` model:

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

The sample contains the following authorization handlers:

- `ContactIsOwnerAuthorizationHandler`: Ensures that a user can only edit their data.
- `ContactManagerAuthorizationHandler`: Allows managers to approve or reject contacts.
- `ContactAdministratorsAuthorizationHandler`: Allows administrators to approve or reject contacts and to edit/delete contacts.

Prerequisites

This tutorial is advanced. You should be familiar with:

- ASP.NET Core
- Authentication
- Account Confirmation and Password Recovery
- Authorization
- Entity Framework Core

The starter and completed app

Download the completed app. Test the completed app so you become familiar with its security features.

The starter app

Download the starter app.

Run the app, tap the **ContactManager** link, and verify you can create, edit, and delete a contact.

Secure user data

The following sections have all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Tie the contact data to the user

Use the ASP.NET Identity user ID to ensure users can edit their data, but not other users data. Add `OwnerID` and `ContactStatus` to the `Contact` model:

```
public class Contact
{
    public int ContactId { get; set; }

    // user ID from AspNetUser table.
    public string OwnerID { get; set; }

    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

`OwnerID` is the user's ID from the `AspNetUser` table in the Identity database. The `Status` field determines if a contact is viewable by general users.

Create a new migration and update the database:

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

Add Role services to Identity

Append AddRoles to add Role services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>()
         .AddEntityFrameworkStores<ApplicationDbContext>();
```

Require authenticated users

Set the default authentication policy to require users to be authenticated:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>()
         .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddMvc(config =>
    {
        // using Microsoft.AspNetCore.Mvc.Authorization;
        // using Microsoft.AspNetCore.Authorization;
        var policy = new AuthorizationPolicyBuilder()
                         .RequireAuthenticatedUser()
                         .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    })
       .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

You can opt out of authentication at the Razor Page, controller, or action method level with the `[AllowAnonymous]` attribute. Setting the default authentication policy to require users to be authenticated protects newly added Razor Pages and controllers. Having authentication required by default is more secure than relying on new controllers and Razor Pages to include the `[Authorize]` attribute.

Add AllowAnonymous to the Index, About, and Contact pages so anonymous users can get information about the site before they register.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages
{
    [AllowAnonymous]
    public class IndexModel : PageModel
    {
        public void OnGet()
        {

        }
    }
}
```

Configure the test account

The `SeedData` class creates two accounts: administrator and manager. Use the Secret Manager tool to set a password for these accounts. Set the password from the project directory (the directory containing *Program.cs*):

```
dotnet user-secrets set SeedUserPW <PW>
```

If a strong password is not specified, an exception is thrown when `SeedData.Initialize` is called.

Update `Main` to use the test password:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            var context = services.GetRequiredService<ApplicationDbContext>();
            context.Database.Migrate();

            // requires using Microsoft.Extensions.Configuration;
            var config = host.Services.GetRequiredService<IConfiguration>();
            // Set password with the Secret Manager tool.
            // dotnet user-secrets set SeedUserPW <pw>

            var testUserPw = config["SeedUserPW"];
            try
            {
                SeedData.Initialize(services, testUserPw).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex.Message, "An error occurred seeding the DB.");
            }
        }

        host.Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Create the test accounts and update the contacts

Update the `Initialize` method in the `SeedData` class to create the test accounts:

```
public static async Task Initialize(IServiceProvider serviceProvider, string testUserPw)
{
    using (var context = new ApplicationDbContext(
        serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))
    {
        // For sample purposes seed both with the same password.
        // Password is set with the following:
        // dotnet user-secrets set SeedUserPW <pw>
        // The admin user can do anything

        var adminID = await EnsureUser(serviceProvider, testUserPw, "admin@contoso.com");
        await EnsureRole(serviceProvider, adminID, Constants.ContactAdministratorsRole);

        // allowed user can create and edit contacts that they create
        var managerID = await EnsureUser(serviceProvider, testUserPw, "manager@contoso.com");
        await EnsureRole(serviceProvider, managerID, Constants.ContactManagersRole);

        SeedDB(context, adminID);
    }
}

private static async Task<string> EnsureUser(IServiceProvider serviceProvider,
                                            string testUserPw, string UserName)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByNameAsync(UserName);
    if (user == null)
    {
        user = new IdentityUser { UserName = UserName };
        await userManager.CreateAsync(user, testUserPw);
    }

    return user.Id;
}

private static async Task<IdentityResult> EnsureRole(IServiceProvider serviceProvider,
                                                      string uid, string role)
{
    IdentityResult IR = null;
    var roleManager = serviceProvider.GetService<RoleManager<IdentityRole>>();

    if (roleManager == null)
    {
        throw new Exception("roleManager null");
    }

    if (!await roleManager.RoleExistsAsync(role))
    {
        IR = await roleManager.CreateAsync(new IdentityRole(role));
    }

    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>();

    var user = await userManager.FindByIdAsync(uid);

    if(user == null)
    {
        throw new Exception("The testUserPw password was probably not strong enough!");
    }

    IR = await userManager.AddToRoleAsync(user, role);

    return IR;
}
```

Add the administrator user ID and `ContactStatus` to the contacts. Make one of the contacts "Submitted" and one "Rejected". Add the user ID and status to all the contacts. Only one contact is shown:

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
    {
        return;   // DB has been seeded
    }

    context.Contact.AddRange(
        new Contact
        {
            Name = "Debra Garcia",
            Address = "1234 Main St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "debra@example.com",
            Status = ContactStatus.Approved,
            OwnerID = adminID
        },
```

Create owner, manager, and administrator authorization handlers

Create an *Authorization* folder and create a `ContactIsOwnerAuthorizationHandler` class in it. The `ContactIsOwnerAuthorizationHandler` verifies that the user acting on a resource owns the resource.

```
using System.Threading.Tasks;
using ContactManager.Data;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
                : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        UserManager<IdentityUser> _userManager;

        public ContactIsOwnerAuthorizationHandler(UserManager<IdentityUser>
            userManager)
        {
            _userManager = userManager;
        }

        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                // Return Task.FromResult(0) if targeting a version of
                // .NET Framework older than 4.6:
                return Task.CompletedTask;
            }

            // If we're not asking for CRUD permission, return.

            if (requirement.Name != Constants.CreateOperationName &&
                requirement.Name != Constants.ReadOperationName   &&
                requirement.Name != Constants.UpdateOperationName &&
                requirement.Name != Constants.DeleteOperationName )
            {
                return Task.CompletedTask;
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

The `ContactIsOwnerAuthorizationHandler` calls context.Succeed if the current authenticated user is the contact owner. Authorization handlers generally:

- Return `context.Succeed` when the requirements are met.
- Return `Task.CompletedTask` when requirements aren't met. `Task.CompletedTask` is not success or failure—it allows other authorization handlers to run.

If you need to explicitly fail, return context.Fail.

The app allows contact owners to edit/delete/create their own data. `ContactIsOwnerAuthorizationHandler` doesn't need to check the operation passed in the requirement parameter.

Create a manager authorization handler

Create a `ContactManagerAuthorizationHandler` class in the *Authorization* folder. The `ContactManagerAuthorizationHandler` verifies the user acting on the resource is a manager. Only managers can approve or reject content changes (new or changed).

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.CompletedTask;
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Create an administrator authorization handler

Create a `ContactAdministratorsAuthorizationHandler` class in the *Authorization* folder. The `ContactAdministratorsAuthorizationHandler` verifies the user acting on the resource is an administrator. Administrator can do all operations.

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
                    : AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
                                            AuthorizationHandlerContext context,
                                    OperationAuthorizationRequirement requirement,
                                     Contact resource)
        {
            if (context.User == null)
            {
                return Task.CompletedTask;
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Register the authorization handlers

Services using Entity Framework Core must be registered for dependency injection using AddScoped. The `ContactIsOwnerAuthorizationHandler` uses ASP.NET Core Identity, which is built on Entity Framework Core. Register the handlers with the service collection so they're available to the `ContactsController` through dependency injection. Add the following code to the end of `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>()
         .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddMvc(config =>
    {
        // using Microsoft.AspNetCore.Mvc.Authorization;
        // using Microsoft.AspNetCore.Authorization;
        var policy = new AuthorizationPolicyBuilder()
                         .RequireAuthenticatedUser()
                         .Build();
        config.Filters.Add(new AuthorizeFilter(policy));
    })
       .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    // Authorization handlers.
    services.AddScoped<IAuthorizationHandler,
                          ContactIsOwnerAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
                          ContactAdministratorsAuthorizationHandler>();

    services.AddSingleton<IAuthorizationHandler,
                          ContactManagerAuthorizationHandler>();
}
```

`ContactAdministratorsAuthorizationHandler` and `ContactManagerAuthorizationHandler` are added as singletons. They're singletons because they don't use EF and all the information needed is in the `Context` parameter of the `HandleRequirementAsync` method.

Support authorization

In this section, you update the Razor Pages and add an operations requirements class.

Review the contact operations requirements class

Review the `ContactOperations` class. This class contains the requirements the app supports:

```
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
          new OperationAuthorizationRequirement {Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
          new OperationAuthorizationRequirement {Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
          new OperationAuthorizationRequirement {Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
          new OperationAuthorizationRequirement {Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
          new OperationAuthorizationRequirement {Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
          new OperationAuthorizationRequirement {Name=Constants.RejectOperationName};
    }

    public class Constants
    {
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =
                                                          "ContactAdministrators";
        public static readonly string ContactManagersRole = "ContactManagers";
    }
}
```

Create a base class for the Contacts Razor Pages

Create a base class that contains the services used in the contacts Razor Pages. The base class puts the initialization code in one location:

```
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages.Contacts
{
    public class DI_BasePageModel : PageModel
    {
        protected ApplicationDbContext Context { get; }
        protected IAuthorizationService AuthorizationService { get; }
        protected UserManager<IdentityUser> UserManager { get; }

        public DI_BasePageModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<IdentityUser> userManager) : base()
        {
            Context = context;
            UserManager = userManager;
            AuthorizationService = authorizationService;
        }
    }
}
```

The preceding code:

- Adds the `IAuthorizationService` service to access to the authorization handlers.
- Adds the Identity `UserManager` service.
- Add the `ApplicationDbContext`.

Update the CreateModel

Update the create page model constructor to use the `DI_BasePageModel` base class:

```
public class CreateModel : DI_BasePageModel
{
    public CreateModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }
```

Update the `CreateModel.OnPostAsync` method to:

- Add the user ID to the `Contact` model.
- Call the authorization handler to verify the user has permission to create contacts.

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    Contact.OwnerID = UserManager.GetUserId(User);

    // requires using ContactManager.Authorization;
    var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                            User, Contact,
                                            ContactOperations.Create);
    if (!isAuthorized.Succeeded)
    {
        return new ChallengeResult();
    }

    Context.Contact.Add(Contact);
    await Context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Update the IndexModel

Update the `OnGetAsync` method so only approved contacts are shown to general users:

```
public class IndexModel : DI_BasePageModel
{
    public IndexModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public IList<Contact> Contact { get; set; }

    public async Task OnGetAsync()
    {
        var contacts = from c in Context.Contact
                       select c;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        // Only approved contacts are shown UNLESS you're authorized to see them
        // or you are the owner.
        if (!isAuthorized)
        {
            contacts = contacts.Where(c => c.Status == ContactStatus.Approved
                                      || c.OwnerID == currentUserId);
        }

        Contact = await contacts.ToListAsync();
    }
}
```

Update the EditModel

Add an authorization handler to verify the user owns the contact. Because resource authorization is being validated, the `[Authorize]` attribute is not enough. The app doesn't have access to the resource when attributes are evaluated. Resource-based authorization must be imperative. Checks must be performed once the app has access to the resource, either by loading it in the page model or by loading it within the handler itself. You frequently access the resource by passing in the resource key.

```
public class EditModel : DI_BasePageModel
{
    public EditModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
                                            m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }
```

```
        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                 User, Contact,
                                                 ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        // Fetch Contact from DB to get OwnerID.
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, contact,
                                                ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        Contact.OwnerID = contact.OwnerID;

        Context.Attach(Contact).State = EntityState.Modified;

        if (contact.Status == ContactStatus.Approved)
        {
            // If the contact is updated after approval,
            // and the user cannot approve,
            // set the status back to submitted so the update can be
            // checked and approved.
            var canApprove = await AuthorizationService.AuthorizeAsync(User,
                                    contact,
                                    ContactOperations.Approve);

            if (!canApprove.Succeeded)
            {
                contact.Status = ContactStatus.Submitted;
            }
        }

        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }

    private bool ContactExists(int id)
    {
        return Context.Contact.Any(e => e.ContactId == id);
    }
}
```

Update the DeleteModel

Update the delete page model to use the authorization handler to verify the user has delete permission on the contact.

```
public class DeleteModel : DI_BasePageModel
{
    public DeleteModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(
                                          m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, Contact,
                                                ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        Contact = await Context.Contact.FindAsync(id);

        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, contact,
                                                ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }

        Context.Contact.Remove(Contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Inject the authorization service into the views

Currently, the UI shows edit and delete links for contacts the user can't modify.

Inject the authorization service in the *Views/_ViewImports.cshtml* file so it's available to all views:

```
@using Microsoft.AspNetCore.Identity
@using ContactManager
@using ContactManager.Data
@namespace ContactManager.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using ContactManager.Authorization;
@using Microsoft.AspNetCore.Authorization
@using ContactManager.Models
@inject IAuthorizationService AuthorizationService
```

The preceding markup adds several `using` statements.

Update the **Edit** and **Delete** links in *Pages/Contacts/Index.cshtml* so they're only rendered for users with the appropriate permissions:

```
@page
@model ContactManager.Pages.Contacts.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Address)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].City)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].State)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Zip)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Email)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Status)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Contact)
        {
            <tr>
                <td>
```

```
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Address)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.City)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.State)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Zip)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Email)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Status)
                </td>
                <td>
                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Update)).Succeeded)
                    {
                        <a asp-page="./Edit" asp-route-id="@item.ContactId">Edit</a>
                        <text> | </text>
                    }

                    <a asp-page="./Details" asp-route-id="@item.ContactId">Details</a>

                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Delete)).Succeeded)
                    {
                        <text> | </text>
                        <a asp-page="./Delete" asp-route-id="@item.ContactId">Delete</a>
                    }
                </td>
            </tr>
        }
    </tbody>
</table>
```

WARNING

Hiding links from users that don't have permission to change data doesn't secure the app. Hiding links makes the app more user-friendly by displaying only valid links. Users can hack the generated URLs to invoke edit and delete operations on data they don't own. The Razor Page or controller must enforce access checks to secure the data.

Update Details

Update the details view so managers can approve or reject contacts:

```
        @*Precedng markup omitted for brevity.*@
        <dt>
            @Html.DisplayNameFor(model => model.Contact.Email)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Email)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Contact.Status)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Status)
        </dd>
    </dl>
</div>

@if (Model.Contact.Status != ContactStatus.Approved)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Approve)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-success">Approve</button>
        </form>
    }
}

@if (Model.Contact.Status != ContactStatus.Rejected)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Reject)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId" />
            <input type="hidden" name="status" value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-success">Reject</button>
        </form>
    }
}

<div>
    @if ((await AuthorizationService.AuthorizeAsync(
         User, Model.Contact,
         ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-id="@Model.Contact.ContactId">Edit</a>
        <text> | </text>
    }
    <a asp-page="./Index">Back to List</a>
</div>
```

Update the details page model:

```
public class DetailsModel : DI_BasePageModel
{
    public DetailsModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact = await Context.Contact.FirstOrDefaultAsync(m => m.ContactId == id);

        if (Contact == null)
        {
            return NotFound();
        }

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||
                           User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            &&  currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return new ChallengeResult();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id, ContactStatus status)
    {
        var contact = await Context.Contact.FirstOrDefaultAsync(
                                                m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var contactOperation = (status == ContactStatus.Approved)
                                                   ? ContactOperations.Approve
                                                   : ContactOperations.Reject;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(User, contact,
                                   contactOperation);
        if (!isAuthorized.Succeeded)
        {
            return new ChallengeResult();
        }
        contact.Status = status;
        Context.Contact.Update(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Add or remove a user to a role

See this issue for information on:

- Removing privileges from a user. For example, muting a user in a chat app.
- Adding privileges to a user.

Test the completed app

If you haven't already set a password for seeded user accounts, use the Secret Manager tool to set a password:

- Choose a strong password: Use eight or more characters and at least one upper-case character, number, and symbol. For example, `Passw0rd!` meets the strong password requirements.
- Execute the following command from the project's folder, where `<PW>` is the password:

```
dotnet user-secrets set SeedUserPW <PW>
```

- Drop and update the Database

```
dotnet ef database drop -f
dotnet ef database update
```

- Restart the app to seed the database.

An easy way to test the completed app is to launch three different browsers (or incognito/InPrivate sessions). In one browser, register a new user (for example, `test@example.com`). Sign in to each browser with a different user. Verify the following operations:

- Registered users can view all of the approved contact data.
- Registered users can edit/delete their own data.
- Managers can approve/reject contact data. The `Details` view shows **Approve** and **Reject** buttons.
- Administrators can approve/reject and edit/delete all data.

USER	SEEDED BY THE APP	OPTIONS
test@example.com	No	Edit/delete the own data.
manager@contoso.com	Yes	Approve/reject and edit/delete own data.
admin@contoso.com	Yes	Approve/reject and edit/delete all data.

Create a contact in the administrator's browser. Copy the URL for delete and edit from the administrator contact. Paste these links into the test user's browser to verify the test user can't perform these operations.

Create the starter app

- Create a Razor Pages app named "ContactManager"
  - Create the app with **Individual User Accounts**.
  - Name it "ContactManager" so the namespace matches the namespace used in the sample.
  - `-uld` specifies LocalDB instead of SQLite

```
dotnet new webapp -o ContactManager -au Individual -uld
```

- Add *Models/Contact.cs*:

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

- Scaffold the `Contact` model.
- Create initial migration and update the database:

```
dotnet aspnet-codegenerator razorpage -m Contact -udl -dc ApplicationDbContext -outDir Pages\Contacts -
-referenceScriptLibraries
dotnet ef database drop -f
dotnet ef migrations add initial
dotnet ef database update
```

- Update the **ContactManager** anchor in the *Pages/_Layout.cshtml* file:

```
<a asp-page="/Contacts/Index" class="navbar-brand">ContactManager</a>
```

- Test the app by creating, editing, and deleting a contact

Seed the database

Add the SeedData class to the *Data* folder.

Call `SeedData.Initialize` from `Main`:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateWebHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<ApplicationDbContext>();
                context.Database.Migrate();
                SeedData.Initialize(services, "not used");
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>();
                logger.LogError(ex, "An error occurred seeding the DB.");
            }
        }

        host.Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Test that the app seeded the database. If there are any rows in the contact DB, the seed method doesn't run.

Additional resources

- Build a .NET Core and SQL Database web app in Azure App Service
- ASP.NET Core Authorization Lab. This lab goes into more detail on the security features introduced in this tutorial.
- Introduction to authorization in ASP.NET Core
- Custom policy-based authorization

Razor Pages authorization conventions in ASP.NET Core

9/22/2020 • 6 minutes to read • Edit Online

One way to control access in your Razor Pages app is to use authorization conventions at startup. These conventions allow you to authorize users and allow anonymous users to access individual pages or folders of pages. The conventions described in this topic automatically apply authorization filters to control access.

View or download sample code (how to download)

The sample app uses cookie authentication without ASP.NET Core Identity. The concepts and examples shown in this topic apply equally to apps that use ASP.NET Core Identity. To use ASP.NET Core Identity, follow the guidance in Introduction to Identity on ASP.NET Core.

Require authorization to access a page

Use the AuthorizePage convention to add an AuthorizeFilter to the page at the specified path:

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

To specify an authorization policy, use an AuthorizePage overload:

```
options.Conventions.AuthorizePage("/Contact", "AtLeast21");
```

NOTE

An AuthorizeFilter can be applied to a page model class with the `[Authorize]` filter attribute. For more information, see Authorize filter attribute.

Require authorization to access a folder of pages

Use the AuthorizeFolder convention to add an AuthorizeFilter to all of the pages in a folder at the specified path:

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

To specify an authorization policy, use an AuthorizeFolder overload:

```
options.Conventions.AuthorizeFolder("/Private", "AtLeast21");
```

Require authorization to access an area page

Use the AuthorizeAreaPage convention to add an AuthorizeFilter to the area page at the specified path:

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts");
```

The page name is the path of the file without an extension relative to the pages root directory for the specified area. For example, the page name for the file *Areas/Identity/Pages/Manage/Accounts.cshtml* is */Manage/Accounts*.

To specify an authorization policy, use an AuthorizeAreaPage overload:

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts", "AtLeast21");
```

Require authorization to access a folder of areas

Use the AuthorizeAreaFolder convention to add an AuthorizeFilter to all of the areas in a folder at the specified path:

```
options.Conventions.AuthorizeAreaFolder("Identity", "/Manage");
```

The folder path is the path of the folder relative to the pages root directory for the specified area. For example, the folder path for the files under *Areas/Identity/Pages/Manage/* is */Manage*.

To specify an authorization policy, use an AuthorizeAreaFolder overload:

```
options.Conventions.AuthorizeAreaFolder("Identity", "/Manage", "AtLeast21");
```

Allow anonymous access to a page

Use the AllowAnonymousToPage convention to add an AllowAnonymousFilter to a page at the specified path:

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

Allow anonymous access to a folder of pages

Use the AllowAnonymousToFolder convention to add an AllowAnonymousFilter to all of the pages in a folder at the specified path:

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

Note on combining authorized and anonymous access

It's valid to specify that a folder of pages requires authorization and then specify that a page within that folder allows anonymous access:

```
// This works.
.AuthorizeFolder("/Private").AllowAnonymousToPage("/Private/Public")
```

The reverse, however, isn't valid. You can't declare a folder of pages for anonymous access and then specify a page within that folder that requires authorization:

```
// This doesn't work!
.AllowAnonymousToFolder("/Public").AuthorizePage("/Public/Private")
```

Requiring authorization on the Private page fails. When both the AllowAnonymousFilter and AuthorizeFilter are applied to the page, the AllowAnonymousFilter takes precedence and controls access.

Additional resources

- Razor Pages route and app conventions in ASP.NET Core
- PageConventionCollection

One way to control access in your Razor Pages app is to use authorization conventions at startup. These conventions allow you to authorize users and allow anonymous users to access individual pages or folders of pages. The conventions described in this topic automatically apply authorization filters to control access.

View or download sample code (how to download)

The sample app uses cookie authentication without ASP.NET Core Identity. The concepts and examples shown in this topic apply equally to apps that use ASP.NET Core Identity. To use ASP.NET Core Identity, follow the guidance in Introduction to Identity on ASP.NET Core.

Require authorization to access a page

Use the AuthorizePage convention via AddRazorPagesOptions to add an AuthorizeFilter to the page at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

To specify an authorization policy, use an AuthorizePage overload:

```
options.Conventions.AuthorizePage("/Contact", "AtLeast21");
```

> **NOTE**
> An AuthorizeFilter can be applied to a page model class with the `[Authorize]` filter attribute. For more information, see Authorize filter attribute.

Require authorization to access a folder of pages

Use the AuthorizeFolder convention via AddRazorPagesOptions to add an AuthorizeFilter to all of the pages in a folder at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

To specify an authorization policy, use an AuthorizeFolder overload:

```
options.Conventions.AuthorizeFolder("/Private", "AtLeast21");
```

Require authorization to access an area page

Use the AuthorizeAreaPage convention via AddRazorPagesOptions to add an AuthorizeFilter to the area page at the specified path:

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts");
```

The page name is the path of the file without an extension relative to the pages root directory for the specified area. For example, the page name for the file *Areas/Identity/Pages/Manage/Accounts.cshtml* is */Manage/Accounts*.

To specify an authorization policy, use an AuthorizeAreaPage overload:

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts", "AtLeast21");
```

Require authorization to access a folder of areas

Use the AuthorizeAreaFolder convention via AddRazorPagesOptions to add an AuthorizeFilter to all of the areas in a folder at the specified path:

```
options.Conventions.AuthorizeAreaFolder("Identity", "/Manage");
```

The folder path is the path of the folder relative to the pages root directory for the specified area. For example, the folder path for the files under *Areas/Identity/Pages/Manage/* is */Manage*.

To specify an authorization policy, use an AuthorizeAreaFolder overload:

```
options.Conventions.AuthorizeAreaFolder("Identity", "/Manage", "AtLeast21");
```

Allow anonymous access to a page

Use the AllowAnonymousToPage convention via AddRazorPagesOptions to add an AllowAnonymousFilter to a page at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

Allow anonymous access to a folder of pages

Use the AllowAnonymousToFolder convention via AddRazorPagesOptions to add an AllowAnonymousFilter to all of the pages in a folder at the specified path:

```
services.AddMvc()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/Contact");
        options.Conventions.AuthorizeFolder("/Private");
        options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
        options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

Note on combining authorized and anonymous access

It's valid to specify that a folder of pages that require authorization and than specify that a page within that folder allows anonymous access:

```
// This works.
.AuthorizeFolder("/Private").AllowAnonymousToPage("/Private/Public")
```

The reverse, however, isn't valid. You can't declare a folder of pages for anonymous access and then specify a page within that folder that requires authorization:

```
// This doesn't work!
.AllowAnonymousToFolder("/Public").AuthorizePage("/Public/Private")
```

Requiring authorization on the Private page fails. When both the AllowAnonymousFilter and AuthorizeFilter are applied to the page, the AllowAnonymousFilter takes precedence and controls access.

Additional resources

- Razor Pages route and app conventions in ASP.NET Core
- PageConventionCollection

Simple authorization in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

Authorization in ASP.NET Core is controlled with AuthorizeAttribute and its various parameters. In its simplest form, applying the `[Authorize]` attribute to a controller, action, or Razor Page, limits access to that component to any authenticated user.

For example, the following code limits access to the `AccountController` to any authenticated user.

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

If you want to apply authorization to an action rather than the controller, apply the `AuthorizeAttribute` attribute to the action itself:

```
public class AccountController : Controller
{
   public ActionResult Login()
   {
   }

   [Authorize]
   public ActionResult Logout()
   {
   }
}
```

Now only authenticated users can access the `Logout` function.

You can also use the `AllowAnonymous` attribute to allow access by non-authenticated users to individual actions. For example:

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

This would allow only authenticated users to the `AccountController`, except for the `Login` action, which is

accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

> **WARNING**
> `[AllowAnonymous]` bypasses all authorization statements. If you combine `[AllowAnonymous]` and any `[Authorize]` attribute, the `[Authorize]` attributes are ignored. For example if you apply `[AllowAnonymous]` at the controller level, any `[Authorize]` attributes on the same controller (or on any action within it) is ignored.

For information on how to globally require all users to be authenticated, see Require authenticated users.

Authorize attribute and Razor Pages

The AuthorizeAttribute can ***not*** be applied to Razor Page handlers. For example, `[Authorize]` can't be applied to `OnGet`, `OnPost`, or any other page handler. Consider using an ASP.NET Core MVC controller for pages with different authorization requirements for different handlers.

The following two approaches can be used to apply authorization to Razor Page handler methods:

- Use separate pages for page handlers requiring different authorization. Moved shared content into one or more partial views. When possible, this is the recommended approach.
- For content that must share a common page, write a filter that performs authorization as part of IAsyncPageFilter.OnPageHandlerSelectionAsync. The PageHandlerAuth GitHub project demonstrates this approach:
  - The AuthorizeIndexPageHandlerFilter implements the authorization filter:

```
[TypeFilter(typeof(AuthorizeIndexPageHandlerFilter))]
public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {

    }

    public void OnPost()
    {

    }

    [AuthorizePageHandler]
    public void OnPostAuthorized()
    {

    }
}
```

  - The [AuthorizePageHandler] attribute is applied to the `OnGet` page handler:

```
public class AuthorizeIndexPageHandlerFilter : IAsyncPageFilter, IOrderedFilter
{
    private readonly IAuthorizationPolicyProvider policyProvider;
    private readonly IPolicyEvaluator policyEvaluator;
```

```
    public AuthorizeIndexPageHandlerFilter(
        IAuthorizationPolicyProvider policyProvider,
        IPolicyEvaluator policyEvaluator)
    {
        this.policyProvider = policyProvider;
        this.policyEvaluator = policyEvaluator;
    }

    // Run late in the selection pipeline
    public int Order => 10000;

    public Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,
PageHandlerExecutionDelegate next) => next();

    public async Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
        var attribute =
context.HandlerMethod?.MethodInfo?.GetCustomAttribute<AuthorizePageHandlerAttribute>();
        if (attribute is null)
        {
            return;
        }

        var policy = await AuthorizationPolicy.CombineAsync(policyProvider, new[] { attribute });
        if (policy is null)
        {
            return;
        }

        await AuthorizeAsync(context, policy);
    }

    #region AuthZ - do not change
    private async Task AuthorizeAsync(ActionContext actionContext, AuthorizationPolicy policy)
    {
        var httpContext = actionContext.HttpContext;
        var authenticateResult = await policyEvaluator.AuthenticateAsync(policy, httpContext);
        var authorizeResult = await policyEvaluator.AuthorizeAsync(policy, authenticateResult,
httpContext, actionContext.ActionDescriptor);
        if (authorizeResult.Challenged)
        {
            if (policy.AuthenticationSchemes.Count > 0)
            {
                foreach (var scheme in policy.AuthenticationSchemes)
                {
                    await httpContext.ChallengeAsync(scheme);
                }
            }
            else
            {
                await httpContext.ChallengeAsync();
            }

            return;
        }
        else if (authorizeResult.Forbidden)
        {
            if (policy.AuthenticationSchemes.Count > 0)
            {
                foreach (var scheme in policy.AuthenticationSchemes)
                {
                    await httpContext.ForbidAsync(scheme);
                }
            }
            else
            {
                await httpContext.ForbidAsync();
            }
```

```
            return;
        }
    }
```

WARNING

The PageHandlerAuth sample approach does ***not***:

- Compose with authorization attributes applied to the page, page model, or globally. Composing authorization attributes results in authentication and authorization executing multiple times when you have one more `AuthorizeAttribute` or `AuthorizeFilter` instances also applied to the page.
- Work in conjunction with the rest of ASP.NET Core authentication and authorization system. You must verify using this approach works correctly for your application.

There are no plans to support the `AuthorizeAttribute` on Razor Page handlers.

Role-based authorization in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

When an identity is created it may belong to one or more roles. For example, Tracy may belong to the Administrator and User roles whilst Scott may only belong to the User role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the IsInRole method on the ClaimsPrincipal class.

Adding role checks

Role-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying roles which the current user must be a member of to access the requested resource.

For example, the following code limits access to any actions on the `AdministrationController` to users who are a member of the `Administrator` role:

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
}
```

You can specify multiple roles as a comma separated list:

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
}
```

This controller would be only accessible by users who are members of the `HRManager` role or the `Finance` role.

If you apply multiple attributes then an accessing user must be a member of all the roles specified; the following sample requires that a user must be a member of both the `PowerUser` and `ControlPanelUser` role.

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

You can further limit access by applying additional role authorization attributes at the action level:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

In the previous code snippet members of the `Administrator` role or the `PowerUser` role can access the controller and the `SetTime` action, but only members of the `Administrator` role can access the `ShutDown` action.

You can also lock down a controller but allow anonymous, unauthenticated access to individual actions.

```
[Authorize]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [AllowAnonymous]
    public ActionResult Login()
    {
    }
}
```

For Razor Pages, the `AuthorizeAttribute` can be applied by either:

- Using a convention, or
- Applying the `AuthorizeAttribute` to the `PageModel` instance:

```
[Authorize(Policy = "RequireAdministratorRole")]
public class UpdateModel : PageModel
{
    public ActionResult OnPost()
    {
    }
}
```

IMPORTANT

Filter attributes, including `AuthorizeAttribute`, can only be applied to PageModel and cannot be applied to specific page handler methods.

Policy based role checks

Role requirements can also be expressed using the new Policy syntax, where a developer registers a policy at startup as part of the Authorization service configuration. This normally occurs in `ConfigureServices()` in your *Startup.cs* file.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole",
             policy => policy.RequireRole("Administrator"));
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdministratorRole",
             policy => policy.RequireRole("Administrator"));
    });
}
```

Policies are applied using the `Policy` property on the `AuthorizeAttribute` attribute:

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

If you want to specify multiple allowed roles in a requirement then you can specify them as parameters to the `RequireRole` method:

```
options.AddPolicy("ElevatedRights", policy =>
                  policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));
```

This example authorizes users who belong to the `Administrator`, `PowerUser` or `BackupAdministrator` roles.

Add Role services to Identity

Append AddRoles to add Role services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddRoles<IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>()
        .AddRoles<IdentityRole>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Claims-based authorization in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name value pair that represents what the subject is, not what the subject can do. For example, you may have a driver's license, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be `DateOfBirth`, the claim value would be your date of birth, for example `8th June 1970` and the issuer would be the driving license authority. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example if you want access to a night club the authorization process might be:

The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

Claim based authorization checks are declarative - the developer embeds them within their code, against a controller or an action within a controller, specifying claims which the current user must possess, and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based, the developer must build and register a policy expressing the claims requirements.

The simplest type of claim policy looks for the presence of a claim and doesn't check the value.

First you need to build and register the policy. This takes place as part of the Authorization service configuration, which normally takes part in `ConfigureServices()` in your *Startup.cs* file.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));
    });
}
```

In this case the `EmployeeOnly` policy checks for the presence of an `EmployeeNumber` claim on the current identity.

You then apply the policy using the `Policy` property on the `AuthorizeAttribute` attribute to specify the policy name;

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

The `AuthorizeAttribute` attribute can be applied to an entire controller, in this instance only identities matching the policy will be allowed access to any Action on the controller.

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }
}
```

If you have a controller that's protected by the `AuthorizeAttribute` attribute, but want to allow anonymous access to particular actions you apply the `AllowAnonymousAttribute` attribute.

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public ActionResult VacationBalance()
    {
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
    }
}
```

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4 or 5.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Founders", policy =>
                          policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("Founders", policy =>
                          policy.RequireClaim("EmployeeNumber", "1", "2", "3", "4", "5"));
    });
}
```

Add a generic claim check

If the claim value isn't a single value or a transformation is required, use RequireAssertion. For more information, see Use a func to fulfill a policy.

Multiple Policy Evaluation

If you apply multiple policies to a controller or action, then all policies must pass before access is granted. For example:

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public ActionResult Payslip()
    {
    }

    [Authorize(Policy = "HumanResources")]
    public ActionResult UpdateSalary()
    {
    }
}
```

In the above example any identity which fulfills the `EmployeeOnly` policy can access the `Payslip` action as that policy is enforced on the controller. However in order to call the `UpdateSalary` action the identity must fulfill *both* the `EmployeeOnly` policy and the `HumanResources` policy.

If you want more complicated policies, such as taking a date of birth claim, calculating an age from it then checking the age is 21 or older then you need to write custom policy handlers.

Policy-based authorization in ASP.NET Core

9/22/2020 • 20 minutes to read • Edit Online

Underneath the covers, role-based authorization and claims-based authorization use a requirement, a requirement handler, and a pre-configured policy. These building blocks support the expression of authorization evaluations in code. The result is a richer, reusable, testable authorization structure.

An authorization policy consists of one or more requirements. It's registered as part of the authorization service configuration, in the `Startup.ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();
    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
}
```

In the preceding example, an "AtLeast21" policy is created. It has a single requirement—that of a minimum age, which is supplied as a parameter to the requirement.

IAuthorizationService

The primary service that determines if authorization is successful is IAuthorizationService:

```
/// <summary>
/// Checks policy based permissions for a user
/// </summary>
public interface IAuthorizationService
{
    /// <summary>
    /// Checks if a user meets a specific set of requirements for the specified resource
    /// </summary>
    /// <param name="user">The user to evaluate the requirements against.</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass null as the value
    /// </param>
    /// <param name="requirements">The requirements to evaluate.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// This value is <value>true</value> when the user fulfills the policy;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource,
                                     IEnumerable<IAuthorizationRequirement> requirements);

    /// <summary>
    /// Checks if a user meets a specific authorization policy
    /// </summary>
    /// <param name="user">The user to check the policy against.</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass null as the value
    /// </param>
    /// <param name="policyName">The name of the policy to check against a specific
    /// context.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// Returns a flag indicating whether the user, and optional resource has fulfilled
    /// the policy.
    /// <value>true</value> when the policy has been fulfilled;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(
                                ClaimsPrincipal user, object resource, string policyName);
}
```

The preceding code highlights the two methods of the IAuthorizationService.

IAuthorizationRequirement is a marker service with no methods, and the mechanism for tracking whether authorization is successful.

Each IAuthorizationHandler is responsible for checking if requirements are met:

```
/// <summary>
/// Classes implementing this interface are able to make a decision if authorization
/// is allowed.
/// </summary>
public interface IAuthorizationHandler
{
    /// <summary>
    /// Makes a decision if authorization is allowed.
    /// </summary>
    /// <param name="context">The authorization information.</param>
    Task HandleAsync(AuthorizationHandlerContext context);
}
```

The AuthorizationHandlerContext class is what the handler uses to mark whether requirements have been met:

```
 context.Succeed(requirement)
```

The following code shows the simplified (and annotated with comments) default implementation of the authorization service:

```
public async Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
             object resource, IEnumerable<IAuthorizationRequirement> requirements)
{
    // Create a tracking context from the authorization inputs.
    var authContext = _contextFactory.CreateContext(requirements, user, resource);

    // By default this returns an IEnumerable<IAuthorizationHandlers> from DI.
    var handlers = await _handlers.GetHandlersAsync(authContext);

    // Invoke all handlers.
    foreach (var handler in handlers)
    {
        await handler.HandleAsync(authContext);
    }

    // Check the context, by default success is when all requirements have been met.
    return _evaluator.Evaluate(authContext);
}
```

The following code shows a typical `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add all of your handlers to DI.
    services.AddSingleton<IAuthorizationHandler, MyHandler1>();
    // MyHandler2, ...

    services.AddSingleton<IAuthorizationHandler, MyHandlerN>();

    // Configure your policies
    services.AddAuthorization(options =>
          options.AddPolicy("Something",
          policy => policy.RequireClaim("Permission", "CanViewPage", "CanViewAnything")));


    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

Use IAuthorizationService or `[Authorize(Policy = "Something")]` for authorization.

Apply policies to MVC controllers

If you're using Razor Pages, see Apply policies to Razor Pages in this document.

Policies are applied to controllers by using the `[Authorize]` attribute with the policy name. For example:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseController : Controller
{
    public IActionResult Index() => View();
}
```

Apply policies to Razor Pages

Policies are applied to Razor Pages by using the `[Authorize]` attribute with the policy name. For example:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseModel : PageModel
{
}
```

Policies can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

Policies can be applied to Razor Pages by using an authorization convention.

Requirements

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user principal. In our "AtLeast21" policy, the requirement is a single parameter—the minimum age. A requirement implements IAuthorizationRequirement, which is an empty marker interface. A parameterized minimum age requirement could be implemented as follows:

```
using Microsoft.AspNetCore.Authorization;

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; }

    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
}
```

If an authorization policy contains multiple authorization requirements, all requirements must pass in order for the policy evaluation to succeed. In other words, multiple authorization requirements added to a single authorization policy are treated on an **AND** basis.

> **NOTE**
>
> A requirement doesn't need to have data or properties.

Authorization handlers

An authorization handler is responsible for the evaluation of a requirement's properties. The authorization handler evaluates the requirements against a provided AuthorizationHandlerContext to determine if access is allowed.

A requirement can have multiple handlers. A handler may inherit AuthorizationHandler<TRequirement>, where `TRequirement` is the requirement to be handled. Alternatively, a handler may implement IAuthorizationHandler to handle more than one type of requirement.

Use a handler for one requirement

The following is an example of a one-to-one relationship in which a minimum age handler utilizes a single requirement:

```
using System;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   MinimumAgeRequirement requirement)
    {
        if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&
                                        c.Issuer == "http://contoso.com"))
        {
            //TODO: Use the following if targeting a version of
            //.NET Framework older than 4.6:
            //      return Task.FromResult(0);
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(
            context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth &&
                                        c.Issuer == "http://contoso.com").Value);

        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

The preceding code determines if the current user principal has a date of birth claim which has been issued

by a known and trusted Issuer. Authorization can't occur when the claim is missing, in which case a completed task is returned. When a claim is present, the user's age is calculated. If the user meets the minimum age defined by the requirement, authorization is deemed successful. When authorization is successful, `context.Succeed` is invoked with the satisfied requirement as its sole parameter.

Use a handler for multiple requirements

The following is an example of a one-to-many relationship in which a permission handler can handle three different types of requirements:

```
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class PermissionHandler : IAuthorizationHandler
{
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        var pendingRequirements = context.PendingRequirements.ToList();

        foreach (var requirement in pendingRequirements)
        {
            if (requirement is ReadPermission)
            {
                if (IsOwner(context.User, context.Resource) ||
                    IsSponsor(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
            else if (requirement is EditPermission ||
                     requirement is DeletePermission)
            {
                if (IsOwner(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }

    private bool IsOwner(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }

    private bool IsSponsor(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }
}
```

The preceding code traverses PendingRequirements—a property containing requirements not marked as successful. For a `ReadPermission` requirement, the user must be either an owner or a sponsor to access the

requested resource. In the case of an `EditPermission` or `DeletePermission` requirement, he or she must be an owner to access the requested resource.

Handler registration

Handlers are registered in the services collection during configuration. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();
    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

The preceding code registers `MinimumAgeHandler` as a singleton by invoking `services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();`. Handlers can be registered using any of the built-in service lifetimes.

What should a handler return?

Note that the `Handle` method in the handler example returns no value. How is a status of either success or failure indicated?

- A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.
- A handler doesn't need to handle failures generally, as other handlers for the same requirement may succeed.
- To guarantee failure, even if other requirement handlers succeed, call `context.Fail`.

If a handler calls `context.Succeed` or `context.Fail`, all other handlers are still called. This allows requirements to produce side effects, such as logging, which takes place even if another handler has successfully validated or failed a requirement. When set to `false`, the InvokeHandlersAfterFailure property (available in ASP.NET Core 1.1 and later) short-circuits the execution of handlers when `context.Fail` is called. `InvokeHandlersAfterFailure` defaults to `true`, in which case all handlers are called.

> **NOTE**
> Authorization handlers are called even if authentication fails.

Why would I want multiple handlers for a requirement?

In cases where you want evaluation to be on an **OR** basis, implement multiple handlers for a single requirement. For example, Microsoft has doors which only open with key cards. If you leave your key card at home, the receptionist prints a temporary sticker and opens the door for you. In this scenario, you'd have a single requirement, *BuildingEntry*, but multiple handlers, each one examining a single requirement.

BuildingEntryRequirement.cs

```
using Microsoft.AspNetCore.Authorization;

public class BuildingEntryRequirement : IAuthorizationRequirement
{
}
```

BadgeEntryHandler.cs

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class BadgeEntryHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == "BadgeId" &&
                                       c.Issuer == "http://microsoftsecurity"))
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

TemporaryStickerHandler.cs

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class TemporaryStickerHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == "TemporaryBadgeId" &&
                                       c.Issuer == "https://microsoftsecurity"))
        {
            // We'd also check the expiration date on the sticker.
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

Ensure that both handlers are registered. If either handler succeeds when a policy evaluates the `BuildingEntryRequirement`, the policy evaluation succeeds.

Use a func to fulfill a policy

There may be situations in which fulfilling a policy is simple to express in code. It's possible to supply a `Func<AuthorizationHandlerContext, bool>` when configuring your policy with the `RequireAssertion` policy builder.

For example, the previous `BadgeEntryHandler` could be rewritten as follows:

```
services.AddAuthorization(options =>
{
     options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context =>
            context.User.HasClaim(c =>
                (c.Type == "BadgeId" ||
                 c.Type == "TemporaryBadgeId") &&
                 c.Issuer == "https://microsoftsecurity")));
});
```

Access MVC request context in handlers

The `HandleRequirementAsync` method you implement in an authorization handler has two parameters: an `AuthorizationHandlerContext` and the `TRequirement` you are handling. Frameworks such as MVC or SignalR are free to add any object to the `Resource` property on the `AuthorizationHandlerContext` to pass extra information.

When using endpoint routing, authorization is typically handled by the Authorization Middleware. In this case, the `Resource` property is an instance of Endpoint. The endpoint can be used to probe the underlying resource to which you're routing. For example:

```
if (context.Resource is Endpoint endpoint)
{
   var actionDescriptor = endpoint.Metadata.GetMetadata<ControllerActionDescriptor>();
   ...
}
```

The endpoint doesn't provide access to the current `HttpContext`. When using endpoint routing, use `IHttpContextAcessor` to access `HttpContext` inside of an authorization handler. For more information, see Use HttpContext from custom components.

With traditional routing, or when authorization happens as part of MVC's authorization filter, the value of `Resource` is an AuthorizationFilterContext instance. This property provides access to `HttpContext`, `RouteData`, and everything else provided by MVC and Razor Pages.

The use of the `Resource` property is framework specific. Using information in the `Resource` property limits your authorization policies to particular frameworks. You should cast the `Resource` property using the `is` keyword, and then confirm the cast has succeeded to ensure your code doesn't crash with an `InvalidCastException` when run on other frameworks:

```
// Requires the following import:
//     using Microsoft.AspNetCore.Mvc.Filters;
if (context.Resource is AuthorizationFilterContext mvcContext)
{
    // Examine MVC-specific things like routing data.
}
```

Globally require all users to be authenticated

For information on how to globally require all users to be authenticated, see Require authenticated users.

Underneath the covers, role-based authorization and claims-based authorization use a requirement, a requirement handler, and a pre-configured policy. These building blocks support the expression of authorization evaluations in code. The result is a richer, reusable, testable authorization structure.

An authorization policy consists of one or more requirements. It's registered as part of the authorization service configuration, in the `Startup.ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
}
```

In the preceding example, an "AtLeast21" policy is created. It has a single requirement—that of a minimum age, which is supplied as a parameter to the requirement.

IAuthorizationService

The primary service that determines if authorization is successful is IAuthorizationService:

```
/// <summary>
/// Checks policy based permissions for a user
/// </summary>
public interface IAuthorizationService
{
    /// <summary>
    /// Checks if a user meets a specific set of requirements for the specified resource
    /// </summary>
    /// <param name="user">The user to evaluate the requirements against.</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass null as the value
    /// </param>
    /// <param name="requirements">The requirements to evaluate.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// This value is <value>true</value> when the user fulfills the policy;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object resource,
                                    IEnumerable<IAuthorizationRequirement> requirements);

    /// <summary>
    /// Checks if a user meets a specific authorization policy
    /// </summary>
    /// <param name="user">The user to check the policy against.</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass null as the value
    /// </param>
    /// <param name="policyName">The name of the policy to check against a specific
    /// context.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// Returns a flag indicating whether the user, and optional resource has fulfilled
    /// the policy.
    /// <value>true</value> when the policy has been fulfilled;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(
                                ClaimsPrincipal user, object resource, string policyName);
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding code highlights the two methods of the IAuthorizationService.

IAuthorizationRequirement is a marker service with no methods, and the mechanism for tracking whether authorization is successful.

Each IAuthorizationHandler is responsible for checking if requirements are met:

```
/// <summary>
/// Classes implementing this interface are able to make a decision if authorization
/// is allowed.
/// </summary>
public interface IAuthorizationHandler
{
    /// <summary>
    /// Makes a decision if authorization is allowed.
    /// </summary>
    /// <param name="context">The authorization information.</param>
    Task HandleAsync(AuthorizationHandlerContext context);
}
```

The AuthorizationHandlerContext class is what the handler uses to mark whether requirements have been met:

```
 context.Succeed(requirement)
```

The following code shows the simplified (and annotated with comments) default implementation of the authorization service:

```
public async Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
             object resource, IEnumerable<IAuthorizationRequirement> requirements)
{
    // Create a tracking context from the authorization inputs.
    var authContext = _contextFactory.CreateContext(requirements, user, resource);

    // By default this returns an IEnumerable<IAuthorizationHandlers> from DI.
    var handlers = await _handlers.GetHandlersAsync(authContext);

    // Invoke all handlers.
    foreach (var handler in handlers)
    {
        await handler.HandleAsync(authContext);
    }

    // Check the context, by default success is when all requirements have been met.
    return _evaluator.Evaluate(authContext);
}
```

The following code shows a typical `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add all of your handlers to DI.
    services.AddSingleton<IAuthorizationHandler, MyHandler1>();
    // MyHandler2, ...

    services.AddSingleton<IAuthorizationHandler, MyHandlerN>();

    // Configure your policies
    services.AddAuthorization(options =>
          options.AddPolicy("Something",
          policy => policy.RequireClaim("Permission", "CanViewPage", "CanViewAnything")));


    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Use IAuthorizationService or `[Authorize(Policy = "Something")]` for authorization.

Apply policies to MVC controllers

If you're using Razor Pages, see Apply policies to Razor Pages in this document.

Policies are applied to controllers by using the `[Authorize]` attribute with the policy name. For example:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseController : Controller
{
    public IActionResult Index() => View();
}
```

Apply policies to Razor Pages

Policies are applied to Razor Pages by using the `[Authorize]` attribute with the policy name. For example:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseModel : PageModel
{
}
```

Policies can also be applied to Razor Pages by using an authorization convention.

Requirements

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user principal. In our "AtLeast21" policy, the requirement is a single parameter—the minimum age. A requirement implements IAuthorizationRequirement, which is an empty marker interface. A parameterized minimum age requirement could be implemented as follows:

```
using Microsoft.AspNetCore.Authorization;

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; }

    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
}
```

If an authorization policy contains multiple authorization requirements, all requirements must pass in order for the policy evaluation to succeed. In other words, multiple authorization requirements added to a single authorization policy are treated on an **AND** basis.

> **NOTE**
>
> A requirement doesn't need to have data or properties.

Authorization handlers

An authorization handler is responsible for the evaluation of a requirement's properties. The authorization handler evaluates the requirements against a provided AuthorizationHandlerContext to determine if access is allowed.

A requirement can have multiple handlers. A handler may inherit AuthorizationHandler<TRequirement>, where `TRequirement` is the requirement to be handled. Alternatively, a handler may implement IAuthorizationHandler to handle more than one type of requirement.

Use a handler for one requirement

The following is an example of a one-to-one relationship in which a minimum age handler utilizes a single requirement:

```
using System;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   MinimumAgeRequirement requirement)
    {
        if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth &&
                                        c.Issuer == "http://contoso.com"))
        {
            //TODO: Use the following if targeting a version of
            //.NET Framework older than 4.6:
            //      return Task.FromResult(0);
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(
            context.User.FindFirst(c => c.Type == ClaimTypes.DateOfBirth &&
                                        c.Issuer == "http://contoso.com").Value);

        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

The preceding code determines if the current user principal has a date of birth claim which has been issued by a known and trusted Issuer. Authorization can't occur when the claim is missing, in which case a completed task is returned. When a claim is present, the user's age is calculated. If the user meets the minimum age defined by the requirement, authorization is deemed successful. When authorization is successful, `context.Succeed` is invoked with the satisfied requirement as its sole parameter.

Use a handler for multiple requirements

The following is an example of a one-to-many relationship in which a permission handler can handle three different types of requirements:

```
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class PermissionHandler : IAuthorizationHandler
{
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        var pendingRequirements = context.PendingRequirements.ToList();

        foreach (var requirement in pendingRequirements)
        {
            if (requirement is ReadPermission)
            {
                if (IsOwner(context.User, context.Resource) ||
                    IsSponsor(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
            else if (requirement is EditPermission ||
                     requirement is DeletePermission)
            {
                if (IsOwner(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }

    private bool IsOwner(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }

    private bool IsSponsor(ClaimsPrincipal user, object resource)
    {
        // Code omitted for brevity

        return true;
    }
}
```

The preceding code traverses PendingRequirements—a property containing requirements not marked as successful. For a `ReadPermission` requirement, the user must be either an owner or a sponsor to access the requested resource. In the case of an `EditPermission` or `DeletePermission` requirement, he or she must be an owner to access the requested resource.

Handler registration

Handlers are registered in the services collection during configuration. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

The preceding code registers `MinimumAgeHandler` as a singleton by invoking `services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();`. Handlers can be registered using any of the built-in service lifetimes.

What should a handler return?

Note that the `Handle` method in the handler example returns no value. How is a status of either success or failure indicated?

- A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.
- A handler doesn't need to handle failures generally, as other handlers for the same requirement may succeed.
- To guarantee failure, even if other requirement handlers succeed, call `context.Fail`.

If a handler calls `context.Succeed` or `context.Fail`, all other handlers are still called. This allows requirements to produce side effects, such as logging, which takes place even if another handler has successfully validated or failed a requirement. When set to `false`, the InvokeHandlersAfterFailure property (available in ASP.NET Core 1.1 and later) short-circuits the execution of handlers when `context.Fail` is called. `InvokeHandlersAfterFailure` defaults to `true`, in which case all handlers are called.

> **NOTE**
> Authorization handlers are called even if authentication fails.

Why would I want multiple handlers for a requirement?

In cases where you want evaluation to be on an **OR** basis, implement multiple handlers for a single requirement. For example, Microsoft has doors which only open with key cards. If you leave your key card at home, the receptionist prints a temporary sticker and opens the door for you. In this scenario, you'd have a single requirement, *BuildingEntry*, but multiple handlers, each one examining a single requirement.

BuildingEntryRequirement.cs

```
using Microsoft.AspNetCore.Authorization;

public class BuildingEntryRequirement : IAuthorizationRequirement
{
}
```

BadgeEntryHandler.cs

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class BadgeEntryHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == "BadgeId" &&
                                       c.Issuer == "http://microsoftsecurity"))
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

TemporaryStickerHandler.cs

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using PoliciesAuthApp1.Services.Requirements;

public class TemporaryStickerHandler : AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   BuildingEntryRequirement requirement)
    {
        if (context.User.HasClaim(c => c.Type == "TemporaryBadgeId" &&
                                       c.Issuer == "https://microsoftsecurity"))
        {
            // We'd also check the expiration date on the sticker.
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

Ensure that both handlers are registered. If either handler succeeds when a policy evaluates the `BuildingEntryRequirement`, the policy evaluation succeeds.

Use a func to fulfill a policy

There may be situations in which fulfilling a policy is simple to express in code. It's possible to supply a `Func<AuthorizationHandlerContext, bool>` when configuring your policy with the `RequireAssertion` policy builder.

For example, the previous `BadgeEntryHandler` could be rewritten as follows:

```
services.AddAuthorization(options =>
{
     options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context =>
            context.User.HasClaim(c =>
                (c.Type == "BadgeId" ||
                 c.Type == "TemporaryBadgeId") &&
                 c.Issuer == "https://microsoftsecurity")));
});
```

Access MVC request context in handlers

The `HandleRequirementAsync` method you implement in an authorization handler has two parameters: an `AuthorizationHandlerContext` and the `TRequirement` you are handling. Frameworks such as MVC or SignalR are free to add any object to the `Resource` property on the `AuthorizationHandlerContext` to pass extra information.

For example, MVC passes an instance of AuthorizationFilterContext in the `Resource` property. This property provides access to `HttpContext`, `RouteData`, and everything else provided by MVC and Razor Pages.

The use of the `Resource` property is framework specific. Using information in the `Resource` property limits your authorization policies to particular frameworks. You should cast the `Resource` property using the `is` keyword, and then confirm the cast has succeeded to ensure your code doesn't crash with an `InvalidCastException` when run on other frameworks:

```
// Requires the following import:
//     using Microsoft.AspNetCore.Mvc.Filters;
if (context.Resource is AuthorizationFilterContext mvcContext)
{
    // Examine MVC-specific things like routing data.
}
```

Custom Authorization Policy Providers using IAuthorizationPolicyProvider in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

By Mike Rousos

Typically when using policy-based authorization, policies are registered by calling `AuthorizationOptions.AddPolicy` as part of authorization service configuration. In some scenarios, it may not be possible (or desirable) to register all authorization policies in this way. In those cases, you can use a custom `IAuthorizationPolicyProvider` to control how authorization policies are supplied.

Examples of scenarios where a custom IAuthorizationPolicyProvider may be useful include:

- Using an external service to provide policy evaluation.
- Using a large range of policies (for different room numbers or ages, for example), so it doesn't make sense to add each individual authorization policy with an `AuthorizationOptions.AddPolicy` call.
- Creating policies at runtime based on information in an external data source (like a database) or determining authorization requirements dynamically through another mechanism.

View or download sample code from the AspNetCore GitHub repository. Download the dotnet/AspNetCore repository ZIP file. Unzip the file. Navigate to the *src/Security/samples/CustomPolicyProvider* project folder.

Customize policy retrieval

ASP.NET Core apps use an implementation of the `IAuthorizationPolicyProvider` interface to retrieve authorization policies. By default, DefaultAuthorizationPolicyProvider is registered and used. `DefaultAuthorizationPolicyProvider` returns policies from the `AuthorizationOptions` provided in an `IServiceCollection.AddAuthorization` call.

Customize this behavior by registering a different `IAuthorizationPolicyProvider` implementation in the app's dependency injection container.

The `IAuthorizationPolicyProvider` interface contains three APIs:

- GetPolicyAsync returns an authorization policy for a given name.
- GetDefaultPolicyAsync returns the default authorization policy (the policy used for `[Authorize]` attributes without a policy specified).
- GetFallbackPolicyAsync returns the fallback authorization policy (the policy used by the Authorization Middleware when no policy is specified).

By implementing these APIs, you can customize how authorization policies are provided.

Parameterized authorize attribute example

One scenario where `IAuthorizationPolicyProvider` is useful is enabling custom `[Authorize]` attributes whose requirements depend on a parameter. For example, in policy-based authorization documentation, an age-based ("AtLeast21") policy was used as a sample. If different controller actions in an app should be made available to users of *different* ages, it might be useful to have many different age-based policies. Instead of registering all the different age-based policies that the application will need in `AuthorizationOptions`, you can generate the policies dynamically with a custom `IAuthorizationPolicyProvider`. To make using the policies easier, you can annotate actions with custom authorization attribute like `[MinimumAgeAuthorize(20)]`.

Custom Authorization attributes

Authorization policies are identified by their names. The custom `MinimumAgeAuthorizeAttribute` described previously needs to map arguments into a string that can be used to retrieve the corresponding authorization policy. You can do this by deriving from `AuthorizeAttribute` and making the `Age` property wrap the `AuthorizeAttribute.Policy` property.

```
internal class MinimumAgeAuthorizeAttribute : AuthorizeAttribute
{
    const string POLICY_PREFIX = "MinimumAge";

    public MinimumAgeAuthorizeAttribute(int age) => Age = age;

    // Get or set the Age property by manipulating the underlying Policy property
    public int Age
    {
        get
        {
            if (int.TryParse(Policy.Substring(POLICY_PREFIX.Length), out var age))
            {
                return age;
            }
            return default(int);
        }
        set
        {
            Policy = $"{POLICY_PREFIX}{value.ToString()}";
        }
    }
}
```

This attribute type has a `Policy` string based on the hard-coded prefix (`"MinimumAge"`) and an integer passed in via the constructor.

You can apply it to actions in the same way as other `Authorize` attributes except that it takes an integer as a parameter.

```
[MinimumAgeAuthorize(10)]
public IActionResult RequiresMinimumAge10()
```

Custom IAuthorizationPolicyProvider

The custom `MinimumAgeAuthorizeAttribute` makes it easy to request authorization policies for any minimum age desired. The next problem to solve is making sure that authorization policies are available for all of those different ages. This is where an `IAuthorizationPolicyProvider` is useful.

When using `MinimumAgeAuthorizationAttribute` , the authorization policy names will follow the pattern `"MinimumAge" + Age` , so the custom `IAuthorizationPolicyProvider` should generate authorization policies by:

- Parsing the age from the policy name.
- Using `AuthorizationPolicyBuilder` to create a new `AuthorizationPolicy`
- In this and following examples it will be assumed that the user is authenticated via a cookie. The `AuthorizationPolicyBuilder` should either be constructed with at least one authorization scheme name or always succeed. Otherwise there is no information on how to provide a challenge to the user and an exception will be thrown.
- Adding requirements to the policy based on the age with `AuthorizationPolicyBuilder.AddRequirements` . In other scenarios, you might use `RequireClaim` , `RequireRole` , or `RequireUserName` instead.

```
internal class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";

    // Policies are looked up by string name, so expect 'parameters' (like age)
    // to be embedded in the policy names. This is abstracted away from developers
    // by the more strongly-typed attributes derived from AuthorizeAttribute
    // (like [MinimumAgeAuthorize()] in this sample)
    public Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        if (policyName.StartsWith(POLICY_PREFIX, StringComparison.OrdinalIgnoreCase) &&
            int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var age))
        {
            var policy = new AuthorizationPolicyBuilder(CookieAuthenticationDefaults.AuthenticationScheme);
            policy.AddRequirements(new MinimumAgeRequirement(age));
            return Task.FromResult(policy.Build());
        }

        return Task.FromResult<AuthorizationPolicy>(null);
    }
}
```

Multiple authorization policy providers

When using custom `IAuthorizationPolicyProvider` implementations, keep in mind that ASP.NET Core only uses one instance of `IAuthorizationPolicyProvider`. If a custom provider isn't able to provide authorization policies for all policy names that will be used, it should defer to a backup provider.

For example, consider an application that needs both custom age policies and more traditional role-based policy retrieval. Such an app could use a custom authorization policy provider that:

- Attempts to parse policy names.
- Calls into a different policy provider (like `DefaultAuthorizationPolicyProvider`) if the policy name doesn't contain an age.

The example `IAuthorizationPolicyProvider` implementation shown above can be updated to use the `DefaultAuthorizationPolicyProvider` by creating a backup policy provider in its constructor (to be used in case the policy name doesn't match its expected pattern of 'MinimumAge' + age).

```
private DefaultAuthorizationPolicyProvider BackupPolicyProvider { get; }

public MinimumAgePolicyProvider(IOptions<AuthorizationOptions> options)
{
    // ASP.NET Core only uses one authorization policy provider, so if the custom implementation
    // doesn't handle all policies it should fall back to an alternate provider.
    BackupPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
}
```

Then, the `GetPolicyAsync` method can be updated to use the `BackupPolicyProvider` instead of returning null:

```
...
return BackupPolicyProvider.GetPolicyAsync(policyName);
```

Default policy

In addition to providing named authorization policies, a custom `IAuthorizationPolicyProvider` needs to implement `GetDefaultPolicyAsync` to provide an authorization policy for `[Authorize]` attributes without a policy name

specified.

In many cases, this authorization attribute only requires an authenticated user, so you can make the necessary policy with a call to `RequireAuthenticatedUser`:

```
public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
    Task.FromResult(new
AuthorizationPolicyBuilder(CookieAuthenticationDefaults.AuthenticationScheme).RequireAuthenticatedUser().Build(
));
```

As with all aspects of a custom `IAuthorizationPolicyProvider`, you can customize this, as needed. In some cases, it may be desirable to retrieve the default policy from a fallback `IAuthorizationPolicyProvider`.

Fallback policy

A custom `IAuthorizationPolicyProvider` can optionally implement `GetFallbackPolicyAsync` to provide a policy that's used when combining policies and when no policies are specified. If `GetFallbackPolicyAsync` returns a non-null policy, the returned policy is used by the Authorization Middleware when no policies are specified for the request.

If no fallback policy is required, the provider can return `null` or defer to the fallback provider:

```
public Task<AuthorizationPolicy> GetFallbackPolicyAsync() =>
    Task.FromResult<AuthorizationPolicy>(null);
```

Use a custom IAuthorizationPolicyProvider

To use custom policies from an `IAuthorizationPolicyProvider`, you ***must***:

- Register the appropriate `AuthorizationHandler` types with dependency injection (described in policy-based authorization), as with all policy-based authorization scenarios.
- Register the custom `IAuthorizationPolicyProvider` type in the app's dependency injection service collection in `Startup.ConfigureServices` to replace the default policy provider.

  ```
  services.AddSingleton<IAuthorizationPolicyProvider, MinimumAgePolicyProvider>();
  ```

A complete custom `IAuthorizationPolicyProvider` sample is available in the dotnet/aspnetcore GitHub repository.

Dependency injection in requirement handlers in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Authorization handlers must be registered in the service collection during configuration (using dependency injection).

Suppose you had a repository of rules you wanted to evaluate inside an authorization handler and that repository was registered in the service collection. Authorization will resolve and inject that into your constructor.

For example, if you wanted to use ASP.NET's logging infrastructure you would want to inject `ILoggerFactory` into your handler. Such a handler might look like:

```
public class LoggingAuthorizationHandler : AuthorizationHandler<MyRequirement>
   {
       ILogger _logger;

       public LoggingAuthorizationHandler(ILoggerFactory loggerFactory)
       {
           _logger = loggerFactory.CreateLogger(this.GetType().FullName);
       }

       protected override Task HandleRequirementAsync(AuthorizationHandlerContext context, MyRequirement
requirement)
       {
           _logger.LogInformation("Inside my handler");
           // Check if the requirement is fulfilled.
           return Task.CompletedTask;
       }
   }
```

You would register the handler with `services.AddSingleton()`:

```
services.AddSingleton<IAuthorizationHandler, LoggingAuthorizationHandler>();
```

An instance of the handler will be created when your application starts, and DI will inject the registered `ILoggerFactory` into your constructor.

> **NOTE**
> Handlers that use Entity Framework shouldn't be registered as singletons.

Resource-based authorization in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

Authorization strategy depends upon the resource being accessed. Consider a document that has an author property. Only the author is allowed to update the document. Consequently, the document must be retrieved from the data store before authorization evaluation can occur.

Attribute evaluation occurs before data binding and before execution of the page handler or action that loads the document. For these reasons, declarative authorization with an `[Authorize]` attribute doesn't suffice. Instead, you can invoke a custom authorization method—a style known as *imperative authorization*.

View or download sample code (how to download).

View or download sample code (how to download).

View or download sample code (how to download).

Create an ASP.NET Core app with user data protected by authorization contains a sample app that uses resource-based authorization.

Use imperative authorization

Authorization is implemented as an IAuthorizationService service and is registered in the service collection within the `Startup` class. The service is made available via dependency injection to page handlers or actions.

```
public class DocumentController : Controller
{
    private readonly IAuthorizationService _authorizationService;
    private readonly IDocumentRepository _documentRepository;

    public DocumentController(IAuthorizationService authorizationService,
                              IDocumentRepository documentRepository)
    {
        _authorizationService = authorizationService;
        _documentRepository = documentRepository;
    }
```

`IAuthorizationService` has two `AuthorizeAsync` method overloads: one accepting the resource and the policy name and the other accepting the resource and a list of requirements to evaluate.

```
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          IEnumerable<IAuthorizationRequirement> requirements);
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          string policyName);
```

```
Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          IEnumerable<IAuthorizationRequirement> requirements);
Task<bool> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          string policyName);
```

In the following example, the resource to be secured is loaded into a custom `Document` object. An `AuthorizeAsync` overload is invoked to determine whether the current user is allowed to edit the provided document. A custom "EditPolicy" authorization policy is factored into the decision. See Custom policy-based authorization for more on creating authorization policies.

> **NOTE**
> The following code samples assume authentication has run and set the `User` property.

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, "EditPolicy");

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

```
[HttpGet]
public async Task<IActionResult> Edit(Guid documentId)
{
    Document document = _documentRepository.Find(documentId);

    if (document == null)
    {
        return new NotFoundResult();
    }

    if (await _authorizationService
        .AuthorizeAsync(User, document, "EditPolicy"))
    {
        return View(document);
    }
    else
    {
        return new ChallengeResult();
    }
}
```

Write a resource-based handler

Writing a handler for resource-based authorization isn't much different than writing a plain requirements handler. Create a custom requirement class, and implement a requirement handler class. For more information on creating

a requirement class, see Requirements.

The handler class specifies both the requirement and resource type. For example, a handler utilizing a `SameAuthorRequirement` and a `Document` resource follows:

```
public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   SameAuthorRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }
```

```
public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   SameAuthorRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }
```

In the preceding example, imagine that `SameAuthorRequirement` is a special case of a more generic `SpecificAuthorRequirement` class. The `SpecificAuthorRequirement` class (not shown) contains a `Name` property representing the name of the author. The `Name` property could be set to the current user.

Register the requirement and handler in `Startup.ConfigureServices`:

```
services.AddControllersWithViews();
services.AddRazorPages();

services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationCrudHandler>();
services.AddScoped<IDocumentRepository, DocumentRepository>();
```

```
services.AddMvc();

services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationCrudHandler>();
services.AddScoped<IDocumentRepository, DocumentRepository>();
```

```
services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationHandler>();
services.AddSingleton<IAuthorizationHandler, DocumentAuthorizationCrudHandler>();
services.AddScoped<IDocumentRepository, DocumentRepository>();
```

Operational requirements

If you're making decisions based on the outcomes of CRUD (Create, Read, Update, Delete) operations, use the OperationAuthorizationRequirement helper class. This class enables you to write a single handler instead of an individual class for each operation type. To use it, provide some operation names:

```
public static class Operations
{
    public static OperationAuthorizationRequirement Create =
        new OperationAuthorizationRequirement { Name = nameof(Create) };
    public static OperationAuthorizationRequirement Read =
        new OperationAuthorizationRequirement { Name = nameof(Read) };
    public static OperationAuthorizationRequirement Update =
        new OperationAuthorizationRequirement { Name = nameof(Update) };
    public static OperationAuthorizationRequirement Delete =
        new OperationAuthorizationRequirement { Name = nameof(Delete) };
}
```

The handler is implemented as follows, using an `OperationAuthorizationRequirement` requirement and a `Document` resource:

```
public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   OperationAuthorizationRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

```
public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                   OperationAuthorizationRequirement requirement,
                                                   Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
        {
            context.Succeed(requirement);
        }

        //TODO: Use the following if targeting a version of
        //.NET Framework older than 4.6:
        //      return Task.FromResult(0);
        return Task.CompletedTask;
    }
}
```

The preceding handler validates the operation using the resource, the user's identity, and the requirement's `Name` property.

Challenge and forbid with an operational resource handler

This section shows how the challenge and forbid action results are processed and how challenge and forbid differ.

To call an operational resource handler, specify the operation when invoking `AuthorizeAsync` in your page handler or action. The following example determines whether the authenticated user is permitted to view the provided document.

> **NOTE**
> The following code samples assume authentication has run and set the `User` property.

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, Operations.Read);

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

If authorization succeeds, the page for viewing the document is returned. If authorization fails but the user is authenticated, returning `ForbidResult` informs any authentication middleware that authorization failed. A `ChallengeResult` is returned when authentication must be performed. For interactive browser clients, it may be appropriate to redirect the user to a login page.

```
[HttpGet]
public async Task<IActionResult> View(Guid documentId)
{
    Document document = _documentRepository.Find(documentId);

    if (document == null)
    {
        return new NotFoundResult();
    }

    if (await _authorizationService
        .AuthorizeAsync(User, document, Operations.Read))
    {
        return View(document);
    }
    else
    {
        return new ChallengeResult();
    }
}
```

If authorization succeeds, the view for the document is returned. If authorization fails, returning `ChallengeResult` informs any authentication middleware that authorization failed, and the middleware can take the appropriate response. An appropriate response could be returning a 401 or 403 status code. For interactive browser clients, it could mean redirecting the user to a login page.

View-based authorization in ASP.NET Core MVC

9/22/2020 • 2 minutes to read • Edit Online

A developer often wants to show, hide, or otherwise modify a UI based on the current user identity. You can access the authorization service within MVC views via dependency injection. To inject the authorization service into a Razor view, use the `@inject` directive:

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
```

If you want the authorization service in every view, place the `@inject` directive into the *_ViewImports.cshtml* file of the *Views* directory. For more information, see Dependency injection into views.

Use the injected authorization service to invoke `AuthorizeAsync` in exactly the same way you would check during resource-based authorization:

```
@if ((await AuthorizationService.AuthorizeAsync(User, "PolicyName")).Succeeded)
{
    <p>This paragraph is displayed because you fulfilled PolicyName.</p>
}
```

In some cases, the resource will be your view model. Invoke `AuthorizeAsync` in exactly the same way you would check during resource-based authorization:

```
@if ((await AuthorizationService.AuthorizeAsync(User, Model, Operations.Edit)).Succeeded)
{
    <p><a class="btn btn-default" role="button"
        href="@Url.Action("Edit", "Document", new { id = Model.Id })">Edit</a></p>
}
```

In the preceding code, the model is passed as a resource the policy evaluation should take into consideration.

WARNING

Don't rely on toggling visibility of your app's UI elements as the sole authorization check. Hiding a UI element may not completely prevent access to its associated controller action. For example, consider the button in the preceding code snippet. A user can invoke the `Edit` action method if he or she knows the relative resource URL is */Document/Edit/1*. For this reason, the `Edit` action method should perform its own authorization check.

Authorize with a specific scheme in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

In some scenarios, such as Single Page Applications (SPAs), it's common to use multiple authentication methods. For example, the app may use cookie-based authentication to log in and JWT bearer authentication for JavaScript requests. In some cases, the app may have multiple instances of an authentication handler. For example, two cookie handlers where one contains a basic identity and one is created when a multi-factor authentication (MFA) has been triggered. MFA may be triggered because the user requested an operation that requires extra security. For more information on enforcing MFA when a user requests a resource that requires MFA, see the GitHub issue Protect section with MFA.

An authentication scheme is named when the authentication service is configured during authentication. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthentication()
        .AddCookie(options => {
            options.LoginPath = "/Account/Unauthorized/";
            options.AccessDeniedPath = "/Account/Forbidden/";
        })
        .AddJwtBearer(options => {
            options.Audience = "http://localhost:5001/";
            options.Authority = "http://localhost:5000/";
        });
```

In the preceding code, two authentication handlers have been added: one for cookies and one for bearer.

> **NOTE**
> Specifying the default scheme results in the `HttpContext.User` property being set to that identity. If that behavior isn't desired, disable it by invoking the parameterless form of `AddAuthentication`.

Selecting the scheme with the Authorize attribute

At the point of authorization, the app indicates the handler to be used. Select the handler with which the app will authorize by passing a comma-delimited list of authentication schemes to `[Authorize]`. The `[Authorize]` attribute specifies the authentication scheme or schemes to use regardless of whether a default is configured. For example:

```
[Authorize(AuthenticationSchemes = AuthSchemes)]
public class MixedController : Controller
    // Requires the following imports:
    // using Microsoft.AspNetCore.Authentication.Cookies;
    // using Microsoft.AspNetCore.Authentication.JwtBearer;
    private const string AuthSchemes =
        CookieAuthenticationDefaults.AuthenticationScheme + "," +
        JwtBearerDefaults.AuthenticationScheme;
```

In the preceding example, both the cookie and bearer handlers run and have a chance to create and append an

identity for the current user. By specifying a single scheme only, the corresponding handler runs.

```
[Authorize(AuthenticationSchemes =
    JwtBearerDefaults.AuthenticationScheme)]
public class MixedController : Controller
```

In the preceding code, only the handler with the "Bearer" scheme runs. Any cookie-based identities are ignored.

Selecting the scheme with policies

If you prefer to specify the desired schemes in policy, you can set the `AuthenticationSchemes` collection when adding your policy:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy =>
    {
        policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new MinimumAgeRequirement());
    });
});
```

In the preceding example, the "Over18" policy only runs against the identity created by the "Bearer" handler. Use the policy by setting the `[Authorize]` attribute's `Policy` property:

```
[Authorize(Policy = "Over18")]
public class RegistrationController : Controller
```

Use multiple authentication schemes

Some apps may need to support multiple types of authentication. For example, your app might authenticate users from Azure Active Directory and from a users database. Another example is an app that authenticates users from both Active Directory Federation Services and Azure Active Directory B2C. In this case, the app should accept a JWT bearer token from several issuers.

Add all authentication schemes you'd like to accept. For example, the following code in `Startup.ConfigureServices` adds two JWT bearer authentication schemes with different issuers:

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://localhost:5000/identity/";
        })
        .AddJwtBearer("AzureAD", options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://login.microsoftonline.com/eb971100-6f99-4bdc-8611-1bc8edd7f436/";
        });
}
```

NOTE

Only one JWT bearer authentication is registered with the default authentication scheme `JwtBearerDefaults.AuthenticationScheme`. Additional authentication has to be registered with a unique authentication scheme.

The next step is to update the default authorization policy to accept both authentication schemes. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    // Code omitted for brevity

    services.AddAuthorization(options =>
    {
        var defaultAuthorizationPolicyBuilder = new AuthorizationPolicyBuilder(
            JwtBearerDefaults.AuthenticationScheme,
            "AzureAD");
        defaultAuthorizationPolicyBuilder =
            defaultAuthorizationPolicyBuilder.RequireAuthenticatedUser();
        options.DefaultPolicy = defaultAuthorizationPolicyBuilder.Build();
    });
}
```

As the default authorization policy is overridden, it's possible to use the `[Authorize]` attribute in controllers. The controller then accepts requests with JWT issued by the first or second issuer.

ASP.NET Core Data Protection

9/22/2020 • 5 minutes to read • Edit Online

Web applications often need to store security-sensitive data. Windows provides DPAPI for desktop applications but this is unsuitable for web applications. The ASP.NET Core data protection stack provide a simple, easy to use cryptographic API a developer can use to protect data, including key management and rotation.

The ASP.NET Core data protection stack is designed to serve as the long-term replacement for the <machineKey> element in ASP.NET 1.x - 4.x. It was designed to address many of the shortcomings of the old cryptographic stack while providing an out-of-the-box solution for the majority of use cases modern applications are likely to encounter.

Problem statement

The overall problem statement can be succinctly stated in a single sentence: I need to persist trusted information for later retrieval, but I don't trust the persistence mechanism. In web terms, this might be written as "I need to round-trip trusted state via an untrusted client."

The canonical example of this is an authentication cookie or bearer token. The server generates an "I am Groot and have xyz permissions" token and hands it to the client. At some future date the client will present that token back to the server, but the server needs some kind of assurance that the client hasn't forged the token. Thus the first requirement: authenticity (a.k.a. integrity, tamper-proofing).

Since the persisted state is trusted by the server, we anticipate that this state might contain information that's specific to the operating environment. This could be in the form of a file path, a permission, a handle or other indirect reference, or some other piece of server-specific data. Such information should generally not be disclosed to an untrusted client. Thus the second requirement: confidentiality.

Finally, since modern applications are componentized, what we've seen is that individual components will want to take advantage of this system without regard to other components in the system. For instance, if a bearer token component is using this stack, it should operate without interference from an anti-CSRF mechanism that might also be using the same stack. Thus the final requirement: isolation.

We can provide further constraints in order to narrow the scope of our requirements. We assume that all services operating within the cryptosystem are equally trusted and that the data doesn't need to be generated or consumed outside of the services under our direct control. Furthermore, we require that operations are as fast as possible since each request to the web service might go through the cryptosystem one or more times. This makes symmetric cryptography ideal for our scenario, and we can discount asymmetric cryptography until such a time that it's needed.

Design philosophy

We started by identifying problems with the existing stack. Once we had that, we surveyed the landscape of existing solutions and concluded that no existing solution quite had the capabilities we sought. We then engineered a solution based on several guiding principles.

- The system should offer simplicity of configuration. Ideally the system would be zero-configuration and developers could hit the ground running. In situations where developers need to configure a specific aspect (such as the key repository), consideration should be given to making those specific configurations simple.

- Offer a simple consumer-facing API. The APIs should be easy to use correctly and difficult to use incorrectly.
- Developers shouldn't learn key management principles. The system should handle algorithm selection and key lifetime on the developer's behalf. Ideally the developer should never even have access to the raw key material.
- Keys should be protected at rest when possible. The system should figure out an appropriate default protection mechanism and apply it automatically.

With these principles in mind we developed a simple, easy to use data protection stack.

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there's nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data.

Audience

The data protection system is divided into five main packages. Various aspects of these APIs target three main audiences;

1. The Consumer APIs Overview target application and framework developers.

   "I don't want to learn about how the stack operates or about how it's configured. I simply want to perform some operation in as simple a manner as possible with high probability of using the APIs successfully."

2. The configuration APIs target application developers and system administrators.

   "I need to tell the data protection system that my environment requires non-default paths or settings."

3. The extensibility APIs target developers in charge of implementing custom policy. Usage of these APIs would be limited to rare situations and experienced, security aware developers.

   "I need to replace an entire component within the system because I have truly unique behavioral requirements. I am willing to learn uncommonly-used parts of the API surface in order to build a plugin that fulfills my requirements."

Package layout

The data protection stack consists of five packages.

- Microsoft.AspNetCore.DataProtection.Abstractions contains the IDataProtectionProvider and IDataProtector interfaces to create data protection services. It also contains useful extension methods for working with these types (for example, IDataProtector.Protect). If the data protection system is instantiated elsewhere and you're consuming the API, reference `Microsoft.AspNetCore.DataProtection.Abstractions`.
- Microsoft.AspNetCore.DataProtection contains the core implementation of the data protection system, including core cryptographic operations, key management, configuration, and extensibility. To instantiate the data protection system (for example, adding it to an IServiceCollection) or modifying or extending its behavior, reference `Microsoft.AspNetCore.DataProtection`.
- Microsoft.AspNetCore.DataProtection.Extensions contains additional APIs which developers might find useful but which don't belong in the core package. For instance, this package contains factory methods to instantiate the data protection system to store keys at a location on the file system without

dependency injection (see DataProtectionProvider). It also contains extension methods for limiting the lifetime of protected payloads (see ITimeLimitedDataProtector).

- Microsoft.AspNetCore.DataProtection.SystemWeb can be installed into an existing ASP.NET 4.x app to redirect its `<machineKey>` operations to use the new ASP.NET Core data protection stack. For more information, see Replace the ASP.NET machineKey in ASP.NET Core.
- Microsoft.AspNetCore.Cryptography.KeyDerivation provides an implementation of the PBKDF2 password hashing routine and can be used by systems that must handle user passwords securely. For more information, see Hash passwords in ASP.NET Core.

Additional resources

Host ASP.NET Core in a web farm

Get started with the Data Protection APIs in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

At its simplest, protecting data consists of the following steps:

1. Create a data protector from a data protection provider.
2. Call the `Protect` method with the data you want to protect.
3. Call the `Unprotect` method with the data you want to turn back into plain text.

Most frameworks and app models, such as ASP.NET Core or SignalR, already configure the data protection system and add it to a service container you access via dependency injection. The following sample demonstrates configuring a service container for dependency injection and registering the data protection stack, receiving the data protection provider via DI, creating a protector and protecting then unprotecting data.

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

When you create a protector you must provide one or more Purpose Strings. A purpose string provides isolation between consumers. For example, a protector created with a purpose string of "green" wouldn't be able to unprotect data provided by a protector with a purpose of "purple".

TIP

Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It's intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`.

A call to `Unprotect` will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Consumer APIs overview for ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

The `IDataProtectionProvider` and `IDataProtector` interfaces are the basic interfaces through which consumers use the data protection system. They're located in the Microsoft.AspNetCore.DataProtection.Abstractions package.

IDataProtectionProvider

The provider interface represents the root of the data protection system. It cannot directly be used to protect or unprotect data. Instead, the consumer must get a reference to an `IDataProtector` by calling `IDataProtectionProvider.CreateProtector(purpose)`, where purpose is a string that describes the intended consumer use case. See Purpose Strings for much more information on the intent of this parameter and how to choose an appropriate value.

IDataProtector

The protector interface is returned by a call to `CreateProtector`, and it's this interface which consumers can use to perform protect and unprotect operations.

To protect a piece of data, pass the data to the `Protect` method. The basic interface defines a method which converts byte[] -> byte[], but there's also an overload (provided as an extension method) which converts string -> string. The security offered by the two methods is identical; the developer should choose whichever overload is most convenient for their use case. Irrespective of the overload chosen, the value returned by the Protect method is now protected (enciphered and tamper-proofed), and the application can send it to an untrusted client.

To unprotect a previously-protected piece of data, pass the protected data to the `Unprotect` method. (There are byte[]-based and string-based overloads for developer convenience.) If the protected payload was generated by an earlier call to `Protect` on this same `IDataProtector`, the `Unprotect` method will return the original unprotected payload. If the protected payload has been tampered with or was produced by a different `IDataProtector`, the `Unprotect` method will throw CryptographicException.

The concept of same vs. different `IDataProtector` ties back to the concept of purpose. If two `IDataProtector` instances were generated from the same root `IDataProtectionProvider` but via different purpose strings in the call to `IDataProtectionProvider.CreateProtector`, then they're considered different protectors, and one won't be able to unprotect payloads generated by the other.

Consuming these interfaces

For a DI-aware component, the intended usage is that the component takes an `IDataProtectionProvider` parameter in its constructor and that the DI system automatically provides this service when the component is instantiated.

> **NOTE**
>
> Some applications (such as console applications or ASP.NET 4.x applications) might not be DI-aware so cannot use the mechanism described here. For these scenarios consult the Non DI Aware Scenarios document for more information on getting an instance of an `IDataProtection` provider without going through DI.

The following sample demonstrates three concepts:

1. Add the data protection system to the service container,

2. Using DI to receive an instance of an `IDataProtectionProvider`, and
3. Creating an `IDataProtector` from an `IDataProtectionProvider` and using it to protect and unprotect data.

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload = _protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

The package Microsoft.AspNetCore.DataProtection.Abstractions contains an extension method `IServiceProvider.GetDataProtector` as a developer convenience. It encapsulates as a single operation both retrieving an `IDataProtectionProvider` from the service provider and calling `IDataProtectionProvider.CreateProtector`. The following sample demonstrates its usage.

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // get an IDataProtector from the IServiceProvider
        var protector = services.GetDataProtector("Contoso.Example.v2");
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
    }
}
```

TIP

Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It's intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`. A call to `Unprotect` will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Purpose strings in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

Components which consume `IDataProtectionProvider` must pass a unique *purposes* parameter to the `CreateProtector` method. The purposes *parameter* is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root cryptographic keys are the same.

When a consumer specifies a purpose, the purpose string is used along with the root cryptographic keys to derive cryptographic subkeys unique to that consumer. This isolates the consumer from all other cryptographic consumers in the application: no other component can read its payloads, and it cannot read any other component's payloads. This isolation also renders infeasible entire categories of attack against the component.

In the diagram above, `IDataProtector` instances A and B **cannot** read each other's payloads, only their own.

The purpose string doesn't have to be secret. It should simply be unique in the sense that no other well-behaved component will ever provide the same purpose string.

> **TIP**
>
> Using the namespace and type name of the component consuming the data protection APIs is a good rule of thumb, as in practice this information will never conflict.
>
> A Contoso-authored component which is responsible for minting bearer tokens might use Contoso.Security.BearerToken as its purpose string. Or - even better - it might use Contoso.Security.BearerToken.v1 as its purpose string. Appending the version number allows a future version to use Contoso.Security.BearerToken.v2 as its purpose, and the different versions would be completely isolated from one another as far as payloads go.

Since the purposes parameter to `CreateProtector` is a string array, the above could've been instead specified as `[ "Contoso.Security.BearerToken", "v1" ]`. This allows establishing a hierarchy of purposes and opens up the possibility of multi-tenancy scenarios with the data protection system.

WARNING

Components shouldn't allow untrusted user input to be the sole source of input for the purposes chain.

For example, consider a component Contoso.Messaging.SecureMessage which is responsible for storing secure messages. If the secure messaging component were to call `CreateProtector([ username ])`, then a malicious user might create an account with username "Contoso.Security.BearerToken" in an attempt to get the component to call `CreateProtector([ "Contoso.Security.BearerToken" ])`, thus inadvertently causing the secure messaging system to mint payloads that could be perceived as authentication tokens.

A better purposes chain for the messaging component would be `CreateProtector([ "Contoso.Messaging.SecureMessage", "User: username" ])`, which provides proper isolation.

The isolation provided by and behaviors of `IDataProtectionProvider`, `IDataProtector`, and purposes are as follows:

- For a given `IDataProtectionProvider` object, the `CreateProtector` method will create an `IDataProtector` object uniquely tied to both the `IDataProtectionProvider` object which created it and the purposes parameter which was passed into the method.
- The purpose parameter must not be null. (If purposes is specified as an array, this means that the array must not be of zero length and all elements of the array must be non-null.) An empty string purpose is technically allowed but is discouraged.
- Two purposes arguments are equivalent if and only if they contain the same strings (using an ordinal comparer) in the same order. A single purpose argument is equivalent to the corresponding single-element purposes array.
- Two `IDataProtector` objects are equivalent if and only if they're created from equivalent `IDataProtectionProvider` objects with equivalent purposes parameters.
- For a given `IDataProtector` object, a call to `Unprotect(protectedData)` will return the original `unprotectedData` if and only if `protectedData := Protect(unprotectedData)` for an equivalent `IDataProtector` object.

NOTE

We're not considering the case where some component intentionally chooses a purpose string which is known to conflict with another component. Such a component would essentially be considered malicious, and this system isn't intended to provide security guarantees in the event that malicious code is already running inside of the worker process.

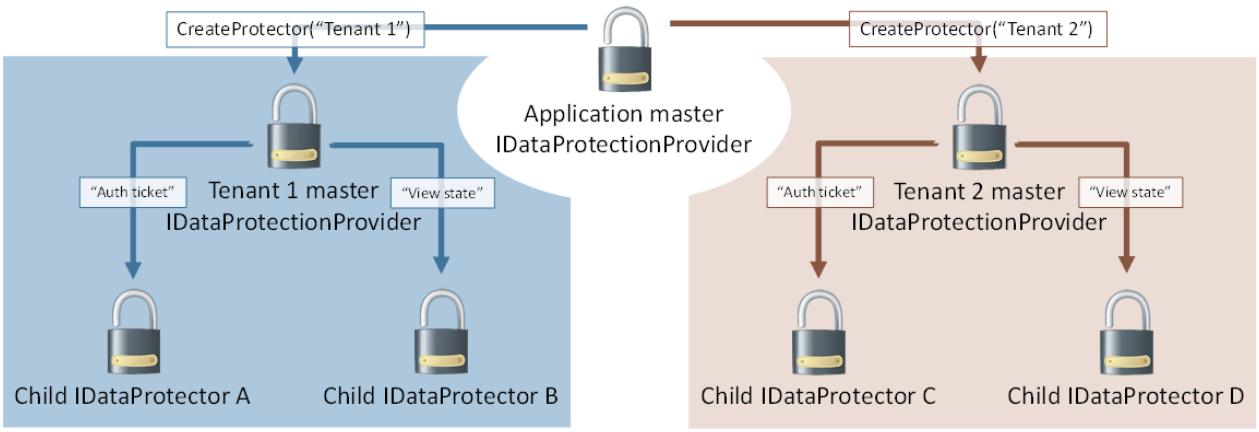
Purpose hierarchy and multi-tenancy in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Since an `IDataProtector` is also implicitly an `IDataProtectionProvider`, purposes can be chained together. In this sense, `provider.CreateProtector([ "purpose1", "purpose2" ])` is equivalent to `provider.CreateProtector("purpose1").CreateProtector("purpose2")`.

This allows for some interesting hierarchical relationships through the data protection system. In the earlier example of Contoso.Messaging.SecureMessage, the SecureMessage component can call `provider.CreateProtector("Contoso.Messaging.SecureMessage")` once up-front and cache the result into a private `_myProvider` field. Future protectors can then be created via calls to `_myProvider.CreateProtector("User: username")`, and these protectors would be used for securing the individual messages.

This can also be flipped. Consider a single logical application which hosts multiple tenants (a CMS seems reasonable), and each tenant can be configured with its own authentication and state management system. The umbrella application has a single master provider, and it calls `provider.CreateProtector("Tenant 1")` and `provider.CreateProtector("Tenant 2")` to give each tenant its own isolated slice of the data protection system. The tenants could then derive their own individual protectors based on their own needs, but no matter how hard they try they cannot create protectors which collide with any other tenant in the system. Graphically, this is represented as below.



WARNING

This assumes the umbrella application controls which APIs are available to individual tenants and that tenants cannot execute arbitrary code on the server. If a tenant can execute arbitrary code, they could perform private reflection to break the isolation guarantees, or they could just read the master keying material directly and derive whatever subkeys they desire.

The data protection system actually uses a sort of multi-tenancy in its default out-of-the-box configuration. By default master keying material is stored in the worker process account's user profile folder (or the registry, for IIS application pool identities). But it's actually fairly common to use a single account to run multiple applications, and thus all these applications would end up sharing the master keying material. To solve this, the data protection system automatically inserts a unique-per-application identifier as the first element in the overall purpose chain. This implicit purpose serves to isolate individual applications from one another by effectively treating each application as a unique tenant within the system, and the protector creation process looks identical to the image above.

Hash passwords in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

The data protection code base includes a package *Microsoft.AspNetCore.Cryptography.KeyDerivation* which contains cryptographic key derivation functions. This package is a standalone component and has no dependencies on the rest of the data protection system. It can be used completely independently. The source exists alongside the data protection code base as a convenience.

The package currently offers a method `KeyDerivation.Pbkdf2` which allows hashing a password using the PBKDF2 algorithm. This API is very similar to the .NET Framework's existing Rfc2898DeriveBytes type, but there are three important distinctions:

1. The `KeyDerivation.Pbkdf2` method supports consuming multiple PRFs (currently `HMACSHA1`, `HMACSHA256`, and `HMACSHA512`), whereas the `Rfc2898DeriveBytes` type only supports `HMACSHA1`.
2. The `KeyDerivation.Pbkdf2` method detects the current operating system and attempts to choose the most optimized implementation of the routine, providing much better performance in certain cases. (On Windows 8, it offers around 10x the throughput of `Rfc2898DeriveBytes`.)
3. The `KeyDerivation.Pbkdf2` method requires the caller to specify all parameters (salt, PRF, and iteration count). The `Rfc2898DeriveBytes` type provides default values for these.

```
using System;
using System.Security.Cryptography;
using Microsoft.AspNetCore.Cryptography.KeyDerivation;

public class Program
{
    public static void Main(string[] args)
    {
        Console.Write("Enter a password: ");
        string password = Console.ReadLine();

        // generate a 128-bit salt using a secure PRNG
        byte[] salt = new byte[128 / 8];
        using (var rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(salt);
        }
        Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");

        // derive a 256-bit subkey (use HMACSHA1 with 10,000 iterations)
        string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
            password: password,
            salt: salt,
            prf: KeyDerivationPrf.HMACSHA1,
            iterationCount: 10000,
            numBytesRequested: 256 / 8));
        Console.WriteLine($"Hashed: {hashed}");
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter a password: Xtw9NMgx
 * Salt: NZsP6NnmfBuYeJrrAKNuVQ==
 * Hashed: /OOoOer10+tGwTRDTrQSoeCxVTFr6dtYly7d0cPxIak=
 */
```

See the source code for ASP.NET Core Identity's `PasswordHasher` type for a real-world use case.

Limit the lifetime of protected payloads in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

There are scenarios where the application developer wants to create a protected payload that expires after a set period of time. For instance, the protected payload might represent a password reset token that should only be valid for one hour. It's certainly possible for the developer to create their own payload format that contains an embedded expiration date, and advanced developers may wish to do this anyway, but for the majority of developers managing these expirations can grow tedious.

To make this easier for our developer audience, the package Microsoft.AspNetCore.DataProtection.Extensions contains utility APIs for creating payloads that automatically expire after a set period of time. These APIs hang off of the `ITimeLimitedDataProtector` type.

API usage

The `ITimeLimitedDataProtector` interface is the core interface for protecting and unprotecting time-limited / self-expiring payloads. To create an instance of an `ITimeLimitedDataProtector`, you'll first need an instance of a regular IDataProtector constructed with a specific purpose. Once the `IDataProtector` instance is available, call the `IDataProtector.ToTimeLimitedDataProtector` extension method to get back a protector with built-in expiration capabilities.

`ITimeLimitedDataProtector` exposes the following API surface and extension methods:

- CreateProtector(string purpose) : ITimeLimitedDataProtector - This API is similar to the existing `IDataProtectionProvider.CreateProtector` in that it can be used to create purpose chains from a root time-limited protector.
- Protect(byte[] plaintext, DateTimeOffset expiration) : byte[]
- Protect(byte[] plaintext, TimeSpan lifetime) : byte[]
- Protect(byte[] plaintext) : byte[]
- Protect(string plaintext, DateTimeOffset expiration) : string
- Protect(string plaintext, TimeSpan lifetime) : string
- Protect(string plaintext) : string

In addition to the core `Protect` methods which take only the plaintext, there are new overloads which allow specifying the payload's expiration date. The expiration date can be specified as an absolute date (via a `DateTimeOffset`) or as a relative time (from the current system time, via a `TimeSpan`). If an overload which doesn't take an expiration is called, the payload is assumed never to expire.

- Unprotect(byte[] protectedData, out DateTimeOffset expiration) : byte[]
- Unprotect(byte[] protectedData) : byte[]
- Unprotect(string protectedData, out DateTimeOffset expiration) : string
- Unprotect(string protectedData) : string

The `Unprotect` methods return the original unprotected data. If the payload hasn't yet expired, the absolute

expiration is returned as an optional out parameter along with the original unprotected data. If the payload is expired, all overloads of the Unprotect method will throw CryptographicException.

> **WARNING**
> It's not advised to use these APIs to protect payloads which require long-term or indefinite persistence. "Can I afford for the protected payloads to be permanently unrecoverable after a month?" can serve as a good rule of thumb; if the answer is no then developers should consider alternative APIs.

The sample below uses the non-DI code paths for instantiating the data protection system. To run this sample, ensure that you have first added a reference to the Microsoft.AspNetCore.DataProtection.Extensions package.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // create a protector for my application

        var provider = DataProtectionProvider.Create(new DirectoryInfo(@"c:\myapp-keys\"));
        var baseProtector = provider.CreateProtector("Contoso.TimeLimitedSample");

        // convert the normal protector into a time-limited protector
        var timeLimitedProtector = baseProtector.ToTimeLimitedDataProtector();

        // get some input and protect it for five seconds
        Console.Write("Enter input: ");
        string input = Console.ReadLine();
        string protectedData = timeLimitedProtector.Protect(input, lifetime: TimeSpan.FromSeconds(5));
        Console.WriteLine($"Protected data: {protectedData}");

        // unprotect it to demonstrate that round-tripping works properly
        string roundtripped = timeLimitedProtector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped data: {roundtripped}");

        // wait 6 seconds and perform another unprotect, demonstrating that the payload self-expires
        Console.WriteLine("Waiting 6 seconds...");
        Thread.Sleep(6000);
        timeLimitedProtector.Unprotect(protectedData);
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protected data: CfDJ8Hu5z0zwxn...nLk7Ok
 * Round-tripped data: Hello!
 * Waiting 6 seconds...
 * <<throws CryptographicException with message 'The payload expired at ...'>>

 */
```

Unprotect payloads whose keys have been revoked in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there's nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data. Keys are never removed from the key ring, so `IDataProtector.Unprotect` can always recover existing payloads as long as the keys are available and valid.

However, an issue arises when the developer tries to unprotect data that has been protected with a revoked key, as `IDataProtector.Unprotect` will throw an exception in this case. This might be fine for short-lived or transient payloads (like authentication tokens), as these kinds of payloads can easily be recreated by the system, and at worst the site visitor might be required to log in again. But for persisted payloads, having `Unprotect` throw could lead to unacceptable data loss.

IPersistedDataProtector

To support the scenario of allowing payloads to be unprotected even in the face of revoked keys, the data protection system contains an `IPersistedDataProtector` type. To get an instance of `IPersistedDataProtector`, simply get an instance of `IDataProtector` in the normal fashion and try casting the `IDataProtector` to `IPersistedDataProtector`.

> **NOTE**
> Not all `IDataProtector` instances can be cast to `IPersistedDataProtector`. Developers should use the C# as operator or similar to avoid runtime exceptions caused by invalid casts, and they should be prepared to handle the failure case appropriately.

`IPersistedDataProtector` exposes the following API surface:

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
     out bool requiresMigration, out bool wasRevoked) : byte[]
```

This API takes the protected payload (as a byte array) and returns the unprotected payload. There's no string-based overload. The two out parameters are as follows.

- `requiresMigration`: will be set to true if the key used to protect this payload is no longer the active default key, e.g., the key used to protect this payload is old and a key rolling operation has since taken place. The caller may wish to consider reprotecting the payload depending on their business needs.
- `wasRevoked`: will be set to true if the key used to protect this payload was revoked.

WARNING

Exercise extreme caution when passing `ignoreRevocationErrors: true` to the `DangerousUnprotect` method. If after calling this method the `wasRevoked` value is true, then the key used to protect this payload was revoked, and the payload's authenticity should be treated as suspect. In this case, only continue operating on the unprotected payload if you have some separate assurance that it's authentic, e.g. that it's coming from a secure database rather than being sent by an untrusted web client.

```
using System;
using System.IO;
using System.Text;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // get a protector and perform a protect operation
        var protector = services.GetDataProtector("Sample.DangerousUnprotect");
        Console.Write("Input: ");
        byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
        var protectedData = protector.Protect(input);
        Console.WriteLine($"Protected payload: {Convert.ToBase64String(protectedData)}");

        // demonstrate that the payload round-trips properly
        var roundTripped = protector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped payload: {Encoding.UTF8.GetString(roundTripped)}");

        // get a reference to the key manager and revoke all keys in the key ring
        var keyManager = services.GetService<IKeyManager>();
        Console.WriteLine("Revoking all keys in the key ring...");
        keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");

        // try calling Protect - this should throw
        Console.WriteLine("Calling Unprotect...");
        try
        {
            var unprotectedPayload = protector.Unprotect(protectedData);
            Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }

        // try calling DangerousUnprotect
        Console.WriteLine("Calling DangerousUnprotect...");
        try
        {
            IPersistedDataProtector persistedProtector = protector as IPersistedDataProtector;
            if (persistedProtector == null)
            {
                throw new Exception("Can't call DangerousUnprotect.");
            }

            bool requiresMigration, wasRevoked;
```

```
            var unprotectedPayload = persistedProtector.DangerousUnprotect(
                protectedData: protectedData,
                ignoreRevocationErrors: true,
                requiresMigration: out requiresMigration,
                wasRevoked: out wasRevoked);
            Console.WriteLine($"Unprotected payload: {Encoding.UTF8.GetString(unprotectedPayload)}");
            Console.WriteLine($"Requires migration = {requiresMigration}, was revoked = {wasRevoked}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Input: Hello!
 * Protected payload: CfDJ8LHIzUCX1ZVBn2BZ...
 * Round-tripped payload: Hello!
 * Revoking all keys in the key ring...
 * Calling Unprotect...
 * CryptographicException: The key {...} has been revoked.
 * Calling DangerousUnprotect...
 * Unprotected payload: Hello!
 * Requires migration = True, was revoked = True
 */
```

Data Protection configuration in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Visit these topics to learn about Data Protection configuration in ASP.NET Core:

- Configure ASP.NET Core Data Protection
  An overview on configuring ASP.NET Core Data Protection.
- Data Protection key management and lifetime
  Information on Data Protection key management and lifetime.
- Data Protection machine-wide policy support
  Details on setting a default machine-wide policy for all apps that use Data Protection.
- Non-DI aware scenarios for Data Protection in ASP.NET Core
  How to use the DataProtectionProvider concrete type to use Data Protection without going through DI-specific code paths.

Configure ASP.NET Core Data Protection

9/22/2020 • 11 minutes to read • Edit Online

When the Data Protection system is initialized, it applies default settings based on the operational environment. These settings are generally appropriate for apps running on a single machine. There are cases where a developer may want to change the default settings:

- The app is spread across multiple machines.
- For compliance reasons.

For these scenarios, the Data Protection system offers a rich configuration API.

> **WARNING**
>
> Similar to configuration files, the data protection key ring should be protected using appropriate permissions. You can choose to encrypt keys at rest, but this doesn't prevent attackers from creating new keys. Consequently, your app's security is impacted. The storage location configured with Data Protection should have its access limited to the app itself, similar to the way you would protect configuration files. For example, if you choose to store your key ring on disk, use file system permissions. Ensure only the identity under which your web app runs has read, write, and create access to that directory. If you use Azure Blob Storage, only the web app should have the ability to read, write, or create new entries in the blob store, etc.
>
> The extension method AddDataProtection returns an IDataProtectionBuilder. `IDataProtectionBuilder` exposes extension methods that you can chain together to configure Data Protection options.

The following NuGet packages are required for the Data Protection extensions used in this article:

- Azure.Extensions.AspNetCore.DataProtection.Blobs
- Azure.Extensions.AspNetCore.DataProtection.Keys

ProtectKeysWithAzureKeyVault

To store keys in Azure Key Vault, configure the system with ProtectKeysWithAzureKeyVault in the `Startup` class. `blobUriWithSasToken` is the full URI where the key file should be stored. The URI must contain the SAS token as a query string parameter:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
        .ProtectKeysWithAzureKeyVault("<keyIdentifier>", "<clientId>", "<clientSecret>");
}
```

Set the key ring storage location (for example, PersistKeysToAzureBlobStorage). The location must be set because calling `ProtectKeysWithAzureKeyVault` implements an IXmlEncryptor that disables automatic data protection settings, including the key ring storage location. The preceding example uses Azure Blob Storage to persist the key ring. For more information, see Key storage providers: Azure Storage. You can also persist the key ring locally with PersistKeysToFileSystem.

The `keyIdentifier` is the key vault key identifier used for key encryption. For example, a key created in key vault named `dataprotection` in the `contosokeyvault` has the key identifier

`https://contosokeyvault.vault.azure.net/keys/dataprotection/`. Provide the app with **Unwrap Key** and **Wrap Key** permissions to the key vault.

`ProtectKeysWithAzureKeyVault` overloads:

- ProtectKeysWithAzureKeyVault(IDataProtectionBuilder, KeyVaultClient, String) permits the use of a KeyVaultClient to enable the data protection system to use the key vault.
- ProtectKeysWithAzureKeyVault(IDataProtectionBuilder, String, String, X509Certificate2) permits the use of a `ClientId` and X509Certificate to enable the data protection system to use the key vault.
- ProtectKeysWithAzureKeyVault(IDataProtectionBuilder, String, String, String) permits the use of a `ClientId` and `ClientSecret` to enable the data protection system to use the key vault.

If the app uses the prior Azure packages (`Microsoft.AspNetCore.DataProtection.AzureStorage` and `Microsoft.AspNetCore.DataProtection.AzureKeyVault`) and a combination of Azure Key Vault and Azure Storage to store and protect keys, System.UriFormatException is thrown if the blob for key storage doesn't exist. The blob can be manually created ahead of running the app in the Azure portal, or use the following procedure:

1. Remove the call to `ProtectKeysWithAzureKeyVault` for the first run to create the blob in place.
2. Add the call to `ProtectKeysWithAzureKeyVault` for subsequent runs.

Removing `ProtectKeysWithAzureKeyVault` for the first run is advised, as it ensures that the file is created with the proper schema and values in place.

We recommended upgrading to the Azure.Extensions.AspNetCore.DataProtection.Blobs and Azure.Extensions.AspNetCore.DataProtection.Keys packages because the API provided automatically creates the blob if it doesn't exist.

```
var storageAccount = CloudStorageAccount.Parse("<storage account connection string">);
var client = storageAccount.CreateCloudBlobClient();
var container = client.GetContainerReference("<key store container name>");

var azureServiceTokenProvider = new AzureServiceTokenProvider();
var keyVaultClient = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(
        azureServiceTokenProvider.KeyVaultTokenCallback));

services.AddDataProtection()
    //This blob must already exist before the application is run
    .PersistKeysToAzureBlobStorage(container, "<key store blob name>")
    //Removing this line below for an initial run will ensure the file is created correctly
    .ProtectKeysWithAzureKeyVault(keyVaultClient, "<keyIdentifier>");
```

PersistKeysToFileSystem

To store keys on a UNC share instead of at the *%LOCALAPPDATA%* default location, configure the system with PersistKeysToFileSystem:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"));
}
```

WARNING

If you change the key persistence location, the system no longer automatically encrypts keys at rest, since it doesn't know whether DPAPI is an appropriate encryption mechanism.

ProtectKeysWith*

You can configure the system to protect keys at rest by calling any of the ProtectKeysWith* configuration APIs. Consider the example below, which stores keys on a UNC share and encrypts those keys at rest with a specific X.509 certificate:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
        .ProtectKeysWithCertificate("thumbprint");
}
```

In ASP.NET Core 2.1 or later, you can provide an X509Certificate2 to ProtectKeysWithCertificate, such as a certificate loaded from a file:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
        .ProtectKeysWithCertificate(
            new X509Certificate2("certificate.pfx", "password"));
}
```

See Key Encryption At Rest for more examples and discussion on the built-in key encryption mechanisms.

UnprotectKeysWithAnyCertificate

In ASP.NET Core 2.1 or later, you can rotate certificates and decrypt keys at rest using an array of X509Certificate2 certificates with UnprotectKeysWithAnyCertificate:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\directory\"))
        .ProtectKeysWithCertificate(
            new X509Certificate2("certificate.pfx", "password"));
        .UnprotectKeysWithAnyCertificate(
            new X509Certificate2("certificate_old_1.pfx", "password_1"),
            new X509Certificate2("certificate_old_2.pfx", "password_2"));
}
```

SetDefaultKeyLifetime

To configure the system to use a key lifetime of 14 days instead of the default 90 days, use SetDefaultKeyLifetime:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
}
```

SetApplicationName

By default, the Data Protection system isolates apps from one another based on their content root paths, even

if they're sharing the same physical key repository. This prevents the apps from understanding each other's protected payloads.

To share protected payloads among apps:

- Configure SetApplicationName in each app with the same value.
- Use the same version of the Data Protection API stack across the apps. Perform **either** of the following in the apps' project files:
  - Reference the same shared framework version via the Microsoft.AspNetCore.App metapackage.
  - Reference the same Data Protection package version.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .SetApplicationName("shared app name");
}
```

DisableAutomaticKeyGeneration

You may have a scenario where you don't want an app to automatically roll keys (create new keys) as they approach expiration. One example of this might be apps set up in a primary/secondary relationship, where only the primary app is responsible for key management concerns and secondary apps simply have a read-only view of the key ring. The secondary apps can be configured to treat the key ring as read-only by configuring the system with DisableAutomaticKeyGeneration:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .DisableAutomaticKeyGeneration();
}
```

Per-application isolation

When the Data Protection system is provided by an ASP.NET Core host, it automatically isolates apps from one another, even if those apps are running under the same worker process account and are using the same master keying material. This is somewhat similar to the IsolateApps modifier from System.Web's `<machineKey>` element.

The isolation mechanism works by considering each app on the local machine as a unique tenant, thus the IDataProtector rooted for any given app automatically includes the app ID as a discriminator. The app's unique ID is the app's physical path:

- For apps hosted in IIS, the unique ID is the IIS physical path of the app. If an app is deployed in a web farm environment, this value is stable assuming that the IIS environments are configured similarly across all machines in the web farm.
- For self-hosted apps running on the Kestrel server, the unique ID is the physical path to the app on disk.

The unique identifier is designed to survive resets—both of the individual app and of the machine itself.

This isolation mechanism assumes that the apps are not malicious. A malicious app can always impact any other app running under the same worker process account. In a shared hosting environment where apps are mutually untrusted, the hosting provider should take steps to ensure OS-level isolation between apps, including separating the apps' underlying key repositories.

If the Data Protection system isn't provided by an ASP.NET Core host (for example, if you instantiate it via the

`DataProtectionProvider` concrete type) app isolation is disabled by default. When app isolation is disabled, all apps backed by the same keying material can share payloads as long as they provide the appropriate purposes. To provide app isolation in this environment, call the SetApplicationName method on the configuration object and provide a unique name for each app.

Changing algorithms with UseCryptographicAlgorithms

The Data Protection stack allows you to change the default algorithm used by newly-generated keys. The simplest way to do this is to call UseCryptographicAlgorithms from the configuration callback:

```
services.AddDataProtection()
    .UseCryptographicAlgorithms(
        new AuthenticatedEncryptorConfiguration()
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
```

```
services.AddDataProtection()
    .UseCryptographicAlgorithms(
        new AuthenticatedEncryptionSettings()
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
```

The default EncryptionAlgorithm is AES-256-CBC, and the default ValidationAlgorithm is HMACSHA256. The default policy can be set by a system administrator via a machine-wide policy, but an explicit call to `UseCryptographicAlgorithms` overrides the default policy.

Calling `UseCryptographicAlgorithms` allows you to specify the desired algorithm from a predefined built-in list. You don't need to worry about the implementation of the algorithm. In the scenario above, the Data Protection system attempts to use the CNG implementation of AES if running on Windows. Otherwise, it falls back to the managed System.Security.Cryptography.Aes class.

You can manually specify an implementation via a call to UseCustomCryptographicAlgorithms.

> **TIP**
> Changing algorithms doesn't affect existing keys in the key ring. It only affects newly-generated keys.

Specifying custom managed algorithms

To specify custom managed algorithms, create a ManagedAuthenticatedEncryptorConfiguration instance that points to the implementation types:

```
serviceCollection.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new ManagedAuthenticatedEncryptorConfiguration()
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
```

To specify custom managed algorithms, create a ManagedAuthenticatedEncryptionSettings instance that points to the implementation types:

```
serviceCollection.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new ManagedAuthenticatedEncryptionSettings()
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
```

Generally the *Type properties must point to concrete, instantiable (via a public parameterless ctor) implementations of SymmetricAlgorithm and KeyedHashAlgorithm, though the system special-cases some values like `typeof(Aes)` for convenience.

> **NOTE**
>
> The SymmetricAlgorithm must have a key length of ≥ 128 bits and a block size of ≥ 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The KeyedHashAlgorithm must have a digest size of >= 128 bits, and it must support keys of length equal to the hash algorithm's digest length. The KeyedHashAlgorithm isn't strictly required to be HMAC.

Specifying custom Windows CNG algorithms

To specify a custom Windows CNG algorithm using CBC-mode encryption with HMAC validation, create a CngCbcAuthenticatedEncryptorConfiguration instance that contains the algorithmic information:

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngCbcAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
```

To specify a custom Windows CNG algorithm using CBC-mode encryption with HMAC validation, create a CngCbcAuthenticatedEncryptionSettings instance that contains the algorithmic information:

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngCbcAuthenticatedEncryptionSettings()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
```

NOTE

The symmetric block cipher algorithm must have a key length of >= 128 bits, a block size of >= 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The hash algorithm must have a digest size of >= 128 bits and must support being opened with the BCRYPT_ALG_HANDLE_HMAC_FLAG flag. The *Provider properties can be set to null to use the default provider for the specified algorithm. See the BCryptOpenAlgorithmProvider documentation for more information.

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption with validation, create a CngGcmAuthenticatedEncryptorConfiguration instance that contains the algorithmic information:

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngGcmAuthenticatedEncryptorConfiguration()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256
    });
```

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption with validation, create a

CngGcmAuthenticatedEncryptionSettings instance that contains the algorithmic information:

```
services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(
        new CngGcmAuthenticatedEncryptionSettings()
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256
    });
```

> **NOTE**
> The symmetric block cipher algorithm must have a key length of >= 128 bits, a block size of exactly 128 bits, and it must support GCM encryption. You can set the EncryptionAlgorithmProvider property to null to use the default provider for the specified algorithm. See the BCryptOpenAlgorithmProvider documentation for more information.

Specifying other custom algorithms

Though not exposed as a first-class API, the Data Protection system is extensible enough to allow specifying almost any kind of algorithm. For example, it's possible to keep all keys contained within a Hardware Security Module (HSM) and to provide a custom implementation of the core encryption and decryption routines. See IAuthenticatedEncryptor in Core cryptography extensibility for more information.

Persisting keys when hosting in a Docker container

When hosting in a Docker container, keys should be maintained in either:

- A folder that's a Docker volume that persists beyond the container's lifetime, such as a shared volume or a host-mounted volume.
- An external provider, such as Azure Key Vault or Redis.

Persisting keys with Redis

Only Redis versions supporting Redis Data Persistence should be used to store keys. Azure Blob storage is persistent and can be used to store keys. For more information, see this GitHub issue.

Additional resources

- Non-DI aware scenarios for Data Protection in ASP.NET Core
- Data Protection machine-wide policy support in ASP.NET Core
- Host ASP.NET Core in a web farm
- Key storage providers in ASP.NET Core

Data Protection key management and lifetime in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Rick Anderson

Key management

The app attempts to detect its operational environment and handle key configuration on its own.

1. If the app is hosted in Azure Apps, keys are persisted to the *%HOME%\ASP.NET\DataProtection-Keys* folder. This folder is backed by network storage and is synchronized across all machines hosting the app.

   - Keys aren't protected at rest.
   - The *DataProtection-Keys* folder supplies the key ring to all instances of an app in a single deployment slot.
   - Separate deployment slots, such as Staging and Production, don't share a key ring. When you swap between deployment slots, for example swapping Staging to Production or using A/B testing, any app using Data Protection won't be able to decrypt stored data using the key ring inside the previous slot. This leads to users being logged out of an app that uses the standard ASP.NET Core cookie authentication, as it uses Data Protection to protect its cookies. If you desire slot-independent key rings, use an external key ring provider, such as Azure Blob Storage, Azure Key Vault, a SQL store, or Redis cache.

2. If the user profile is available, keys are persisted to the *%LOCALAPPDATA%\ASP.NET\DataProtection-Keys* folder. If the operating system is Windows, the keys are encrypted at rest using DPAPI.

   The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:

   a. Navigate to the *%windir%/system32/inetsrv/config* folder.
   b. Open the *applicationHost.config* file.
   c. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.
   d. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

3. If the app is hosted in IIS, keys are persisted to the HKLM registry in a special registry key that's ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.

4. If none of these conditions match, keys aren't persisted outside of the current process. When the process shuts down, all generated keys are lost.

The developer is always in full control and can override how and where keys are stored. The first three options above should provide good defaults for most apps similar to how the ASP.NET **<machineKey>** auto-generation routines worked in the past. The final, fallback option is the only scenario that requires the developer to specify configuration upfront if they want key persistence, but this fallback only occurs in rare situations.

When hosting in a Docker container, keys should be persisted in a folder that's a Docker volume (a shared volume or a host-mounted volume that persists beyond the container's lifetime) or in an external provider, such as Azure Key Vault or Redis. An external provider is also useful in web farm scenarios if apps can't access a shared network volume (see PersistKeysToFileSystem for more information).

> **WARNING**
>
> If the developer overrides the rules outlined above and points the Data Protection system at a specific key repository, automatic encryption of keys at rest is disabled. At-rest protection can be re-enabled via configuration.

Key lifetime

Keys have a 90-day lifetime by default. When a key expires, the app automatically generates a new key and sets the new key as the active key. As long as retired keys remain on the system, your app can decrypt any data protected with them. See key management for more information.

Default algorithms

The default payload protection algorithm used is AES-256-CBC for confidentiality and HMACSHA256 for authenticity. A 512-bit master key, changed every 90 days, is used to derive the two sub-keys used for these algorithms on a per-payload basis. See subkey derivation for more information.

Additional resources

- Key management extensibility in ASP.NET Core
- Host ASP.NET Core in a web farm

Data Protection machine-wide policy support in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Rick Anderson

When running on Windows, the Data Protection system has limited support for setting a default machine-wide policy for all apps that consume ASP.NET Core Data Protection. The general idea is that an administrator might wish to change a default setting, such as the algorithms used or key lifetime, without the need to manually update every app on the machine.

> **WARNING**
> The system administrator can set default policy, but they can't enforce it. The app developer can always override any value with one of their own choosing. The default policy only affects apps where the developer hasn't specified an explicit value for a setting.

Setting default policy

To set default policy, an administrator can set known values in the system registry under the following registry key:

HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection

If you're on a 64-bit operating system and want to affect the behavior of 32-bit apps, remember to configure the Wow6432Node equivalent of the above key.

The supported values are shown below.

VALUE	TYPE	DESCRIPTION
EncryptionType	string	Specifies which algorithms should be used for data protection. The value must be CNG-CBC, CNG-GCM, or Managed and is described in more detail below.
DefaultKeyLifetime	DWORD	Specifies the lifetime for newly-generated keys. The value is specified in days and must be >= 7.
KeyEscrowSinks	string	Specifies the types that are used for key escrow. The value is a semicolon-delimited list of key escrow sinks, where each element in the list is the assembly-qualified name of a type that implements IKeyEscrowSink.

Encryption types

If EncryptionType is CNG-CBC, the system is configured to use a CBC-mode symmetric block cipher for

confidentiality and HMAC for authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngCbcAuthenticatedEncryptionSettings type.

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in CBC mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm EncryptionAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.
HashAlgorithm	string	The name of a hash algorithm understood by CNG. This algorithm is opened in HMAC mode.
HashAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm HashAlgorithm.

If EncryptionType is CNG-GCM, the system is configured to use a Galois/Counter Mode symmetric block cipher for confidentiality and authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngGcmAuthenticatedEncryptionSettings type.

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in Galois/Counter Mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm EncryptionAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.

If EncryptionType is Managed, the system is configured to use a managed SymmetricAlgorithm for confidentiality and KeyedHashAlgorithm for authenticity (see Specifying custom managed algorithms for more details). The following additional values are supported, each of which corresponds to a property on the ManagedAuthenticatedEncryptionSettings type.

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithmType	string	The assembly-qualified name of a type that implements SymmetricAlgorithm.

VALUE	TYPE	DESCRIPTION
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric encryption algorithm.
ValidationAlgorithmType	string	The assembly-qualified name of a type that implements KeyedHashAlgorithm.

If EncryptionType has any other value other than null or empty, the Data Protection system throws an exception at startup.

> **WARNING**
> When configuring a default policy setting that involves type names (EncryptionAlgorithmType, ValidationAlgorithmType, KeyEscrowSinks), the types must be available to the app. This means that for apps running on Desktop CLR, the assemblies that contain these types should be present in the Global Assembly Cache (GAC). For ASP.NET Core apps running on .NET Core, the packages that contain these types should be installed.

Non-DI aware scenarios for Data Protection in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Rick Anderson

The ASP.NET Core Data Protection system is normally added to a service container and consumed by dependent components via dependency injection (DI). However, there are cases where this isn't feasible or desired, especially when importing the system into an existing app.

To support these scenarios, the Microsoft.AspNetCore.DataProtection.Extensions package provides a concrete type, DataProtectionProvider, which offers a simple way to use Data Protection without relying on DI. The `DataProtectionProvider` type implements IDataProtectionProvider. Constructing `DataProtectionProvider` only requires providing a DirectoryInfo instance to indicate where the provider's cryptographic keys should be stored, as seen in the following code sample:

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder));

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8FWbAn6...ch3hAPm1NJA
 * Unprotect returned: Hello world!
 *
 * Press any key...
*/
```

By default, the `DataProtectionProvider` concrete type doesn't encrypt raw key material before persisting it to the file system. This is to support scenarios where the developer points to a network share and the Data Protection system can't automatically deduce an appropriate at-rest key encryption mechanism.

Additionally, the `DataProtectionProvider` concrete type doesn't isolate apps by default. All apps using the same key directory can share payloads as long as their purpose parameters match.

The DataProtectionProvider constructor accepts an optional configuration callback that can be used to adjust the behaviors of the system. The sample below demonstrates restoring isolation with an explicit call to SetApplicationName. The sample also demonstrates configuring the system to automatically encrypt persisted keys using Windows DPAPI. If the directory points to a UNC share, you may wish to distribute a shared certificate across all relevant machines and to configure the system to use certificate-based encryption with a call to ProtectKeysWithCertificate.

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder),
            configuration =>
            {
                configuration.SetApplicationName("my app name");
                configuration.ProtectKeysWithDpapi();
            });

        var protector = dataProtectionProvider.CreateProtector("Program.No-DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}
```

TIP

Instances of the `DataProtectionProvider` concrete type are expensive to create. If an app maintains multiple instances of this type and if they're all using the same key storage directory, app performance might degrade. If you use the `DataProtectionProvider` type, we recommend that you create this type once and reuse it as much as possible. The `DataProtectionProvider` type and all IDataProtector instances created from it are thread-safe for multiple callers.

ASP.NET Core Data Protection extensibility APIs

9/22/2020 • 2 minutes to read • Edit Online

- Core cryptography extensibility
- Key management extensibility
- Miscellaneous APIs

Core cryptography extensibility in ASP.NET Core

9/22/2020 • 5 minutes to read • Edit Online

WARNING

Types that implement any of the following interfaces should be thread-safe for multiple callers.

IAuthenticatedEncryptor

The **IAuthenticatedEncryptor** interface is the basic building block of the cryptographic subsystem. There's generally one IAuthenticatedEncryptor per key, and the IAuthenticatedEncryptor instance wraps all cryptographic key material and algorithmic information necessary to perform cryptographic operations.

As its name suggests, the type is responsible for providing authenticated encryption and decryption services. It exposes the following two APIs.

- `Decrypt(ArraySegment<byte> ciphertext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`
- `Encrypt(ArraySegment<byte> plaintext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`

The Encrypt method returns a blob that includes the enciphered plaintext and an authentication tag. The authentication tag must encompass the additional authenticated data (AAD), though the AAD itself need not be recoverable from the final payload. The Decrypt method validates the authentication tag and returns the deciphered payload. All failures (except ArgumentNullException and similar) should be homogenized to CryptographicException.

NOTE

The IAuthenticatedEncryptor instance itself doesn't actually need to contain the key material. For example, the implementation could delegate to an HSM for all operations.

How to create an IAuthenticatedEncryptor

- ASP.NET Core 2.x
- ASP.NET Core 1.x

The **IAuthenticatedEncryptorFactory** interface represents a type that knows how to create an IAuthenticatedEncryptor instance. Its API is as follows.

- CreateEncryptorInstance(IKey key) : IAuthenticatedEncryptor

For any given IKey instance, any authenticated encryptors created by its CreateEncryptorInstance method should be considered equivalent, as in the below code sample.

```
// we have an IAuthenticatedEncryptorFactory instance and an IKey instance
IAuthenticatedEncryptorFactory factory = ...;
IKey key = ...;

// get an encryptor instance and perform an authenticated encryption operation
ArraySegment<byte> plaintext = new ArraySegment<byte>(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = factory.CreateEncryptorInstance(key);
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated decryption operation
var encryptor2 = factory.CreateEncryptorInstance(key);
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>(ciphertext), aad);


// the 'roundTripped' and 'plaintext' buffers should be equivalent
```

IAuthenticatedEncryptorDescriptor (ASP.NET Core 2.x only)

- ASP.NET Core 2.x
- ASP.NET Core 1.x

The **IAuthenticatedEncryptorDescriptor** interface represents a type that knows how to export itself to XML. Its API is as follows.

- ExportToXml() : XmlSerializedDescriptorInfo

XML Serialization

The primary difference between IAuthenticatedEncryptor and IAuthenticatedEncryptorDescriptor is that the descriptor knows how to create the encryptor and supply it with valid arguments. Consider an IAuthenticatedEncryptor whose implementation relies on SymmetricAlgorithm and KeyedHashAlgorithm. The encryptor's job is to consume these types, but it doesn't necessarily know where these types came from, so it can't really write out a proper description of how to recreate itself if the application restarts. The descriptor acts as a higher level on top of this. Since the descriptor knows how to create the encryptor instance (e.g., it knows how to create the required algorithms), it can serialize that knowledge in XML form so that the encryptor instance can be recreated after an application reset.

The descriptor can be serialized via its ExportToXml routine. This routine returns an XmlSerializedDescriptorInfo which contains two properties: the XElement representation of the descriptor and the Type which represents an IAuthenticatedEncryptorDescriptorDeserializer which can be used to resurrect this descriptor given the corresponding XElement.

The serialized descriptor may contain sensitive information such as cryptographic key material. The data protection system has built-in support for encrypting information before it's persisted to storage. To take advantage of this, the descriptor should mark the element which contains sensitive information with the attribute name "requiresEncryption" (xmlns "http://schemas.asp.net/2015/03/dataProtection"), value "true".

> **TIP**
> There's a helper API for setting this attribute. Call the extension method XElement.MarkAsRequiresEncryption() located in namespace Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel.

There can also be cases where the serialized descriptor doesn't contain sensitive information. Consider again the case of a cryptographic key stored in an HSM. The descriptor cannot write out the key material when serializing itself since the HSM won't expose the material in plaintext form. Instead, the descriptor might write out the key-

wrapped version of the key (if the HSM allows export in this fashion) or the HSM's own unique identifier for the key.

IAuthenticatedEncryptorDescriptorDeserializer

The **IAuthenticatedEncryptorDescriptorDeserializer** interface represents a type that knows how to deserialize an IAuthenticatedEncryptorDescriptor instance from an XElement. It exposes a single method:

- ImportFromXml(XElement element) : IAuthenticatedEncryptorDescriptor

The ImportFromXml method takes the XElement that was returned by IAuthenticatedEncryptorDescriptor.ExportToXml and creates an equivalent of the original IAuthenticatedEncryptorDescriptor.

Types which implement IAuthenticatedEncryptorDescriptorDeserializer should have one of the following two public constructors:

- .ctor(IServiceProvider)
- .ctor()

> **NOTE**
> The IServiceProvider passed to the constructor may be null.

The top-level factory

- ASP.NET Core 2.x
- ASP.NET Core 1.x

The **AlgorithmConfiguration** class represents a type which knows how to create IAuthenticatedEncryptorDescriptor instances. It exposes a single API.

- CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor

Think of AlgorithmConfiguration as the top-level factory. The configuration serves as a template. It wraps algorithmic information (e.g., this configuration produces descriptors with an AES-128-GCM master key), but it's not yet associated with a specific key.

When CreateNewDescriptor is called, fresh key material is created solely for this call, and a new IAuthenticatedEncryptorDescriptor is produced which wraps this key material and the algorithmic information required to consume the material. The key material could be created in software (and held in memory), it could be created and held within an HSM, and so on. The crucial point is that any two calls to CreateNewDescriptor should never create equivalent IAuthenticatedEncryptorDescriptor instances.

The AlgorithmConfiguration type serves as the entry point for key creation routines such as automatic key rolling. To change the implementation for all future keys, set the AuthenticatedEncryptorConfiguration property in KeyManagementOptions.

Key management extensibility in ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

Read the key management section before reading this section, as it explains some of the fundamental concepts behind these APIs.

Warning: Types that implement any of the following interfaces should be thread-safe for multiple callers.

Key

The `IKey` interface is the basic representation of a key in cryptosystem. The term key is used here in the abstract sense, not in the literal sense of "cryptographic key material". A key has the following properties:

- Activation, creation, and expiration dates
- Revocation status
- Key identifier (a GUID)

Additionally, `IKey` exposes a `CreateEncryptor` method which can be used to create an IAuthenticatedEncryptor instance tied to this key.

Additionally, `IKey` exposes a `CreateEncryptorInstance` method which can be used to create an IAuthenticatedEncryptor instance tied to this key.

> **NOTE**
> There's no API to retrieve the raw cryptographic material from an `IKey` instance.

IKeyManager

The `IKeyManager` interface represents an object responsible for general key storage, retrieval, and manipulation. It exposes three high-level operations:

- Create a new key and persist it to storage.
- Get all keys from storage.
- Revoke one or more keys and persist the revocation information to storage.

> **WARNING**
> Writing an `IKeyManager` is a very advanced task, and the majority of developers shouldn't attempt it. Instead, most developers should take advantage of the facilities offered by the XmlKeyManager class.

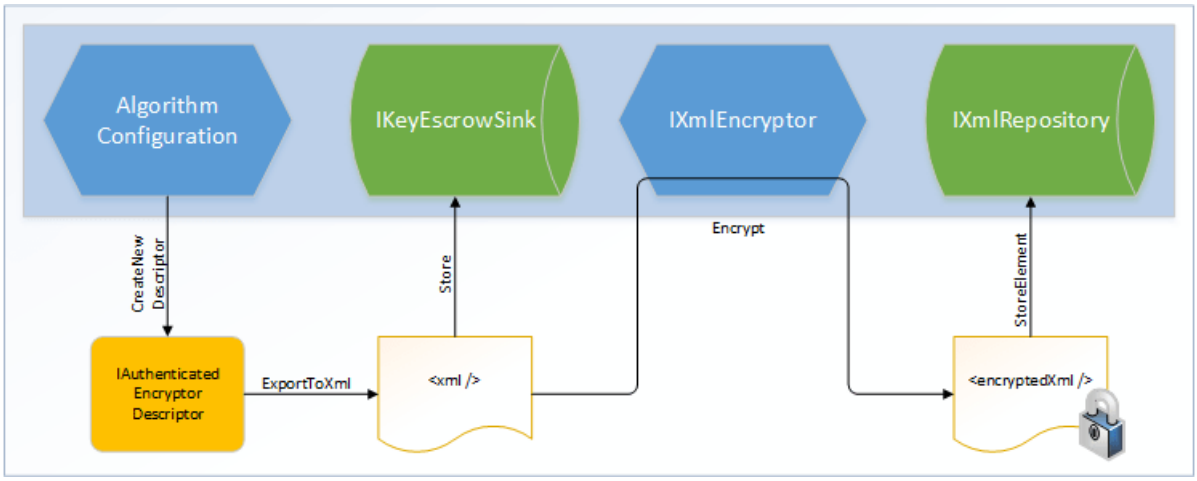
XmlKeyManager

The `XmlKeyManager` type is the in-box concrete implementation of `IKeyManager`. It provides several useful facilities, including key escrow and encryption of keys at rest. Keys in this system are represented as XML elements (specifically, XElement).

`XmlKeyManager` depends on several other components in the course of fulfilling its tasks:

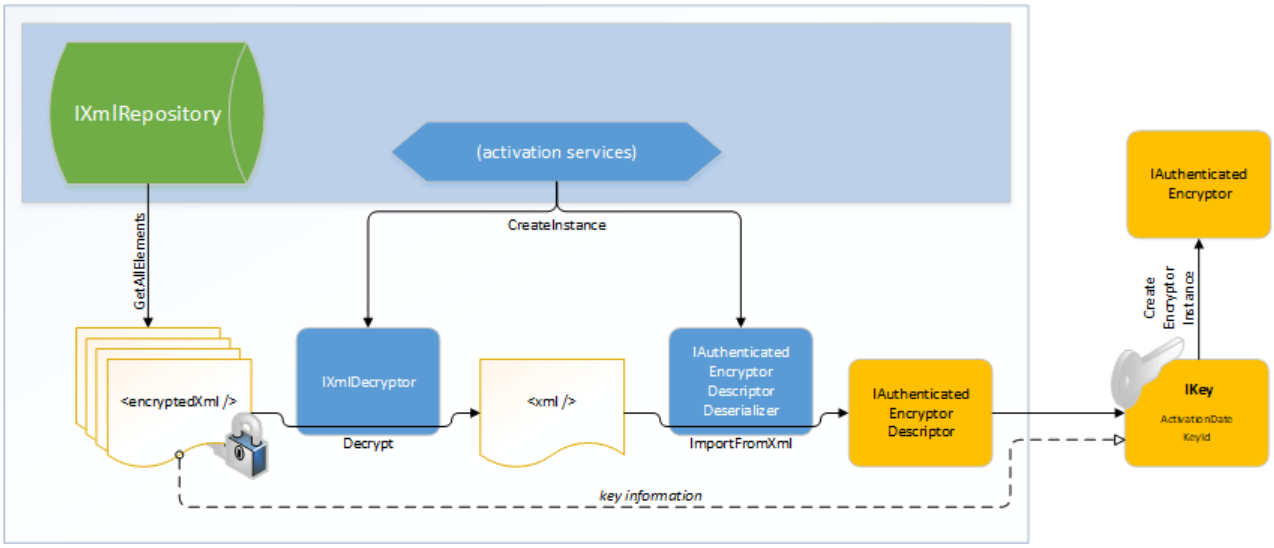
- `AlgorithmConfiguration`, which dictates the algorithms used by new keys.
- `IXmlRepository`, which controls where keys are persisted in storage.
- `IXmlEncryptor` [optional], which allows encrypting keys at rest.
- `IKeyEscrowSink` [optional], which provides key escrow services.
- `IXmlRepository`, which controls where keys are persisted in storage.
- `IXmlEncryptor` [optional], which allows encrypting keys at rest.
- `IKeyEscrowSink` [optional], which provides key escrow services.

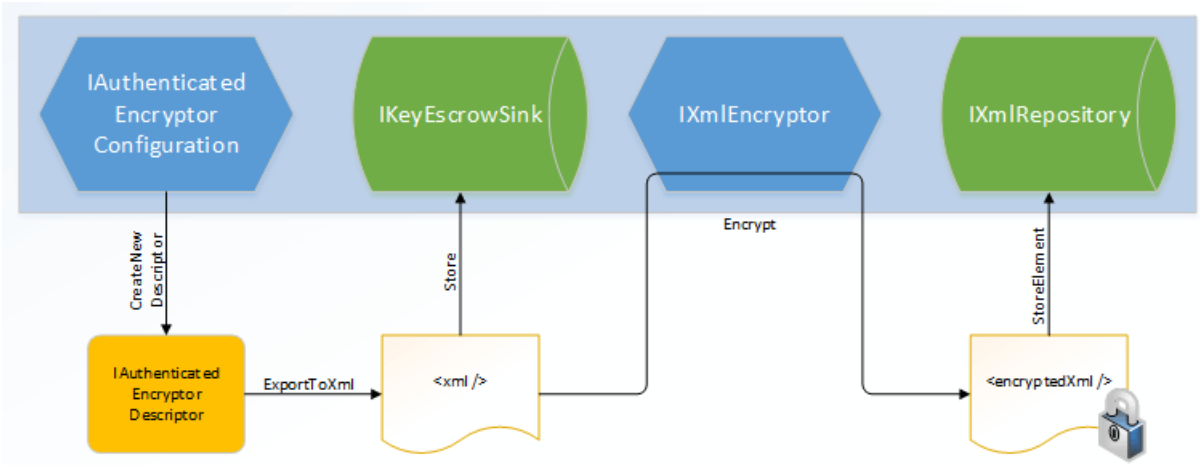
Below are high-level diagrams which indicate how these components are wired together within `XmlKeyManager`.



Key Creation / CreateNewKey

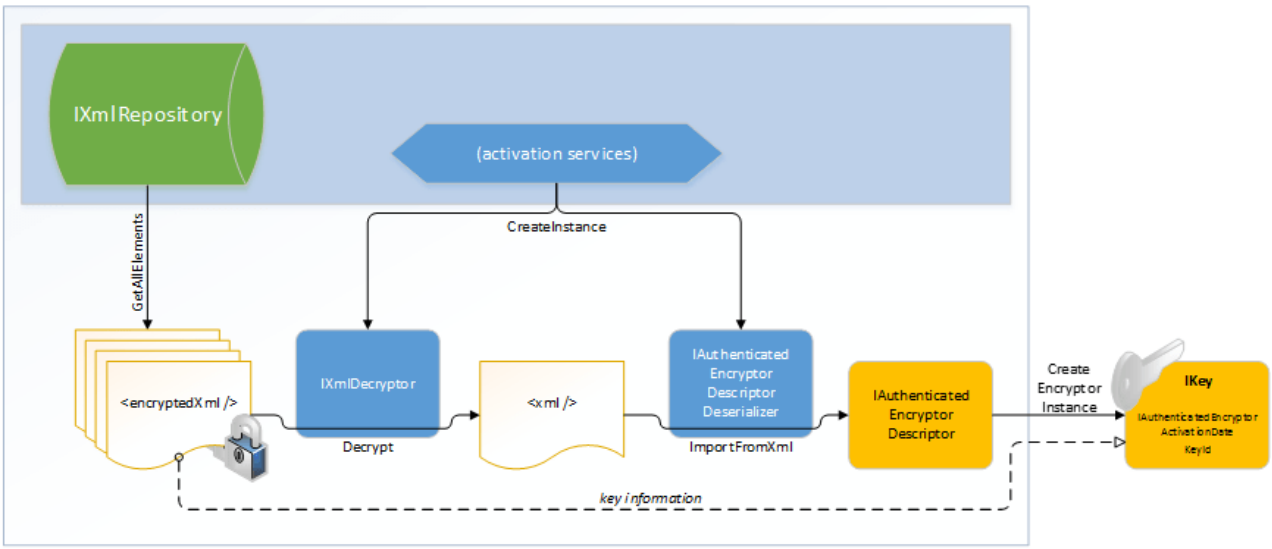
In the implementation of `CreateNewKey`, the `AlgorithmConfiguration` component is used to create a unique `IAuthenticatedEncryptorDescriptor`, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an `IXmlEncryptor` (if required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the `IXmlRepository`. (If no `IXmlEncryptor` is configured, the unencrypted document is persisted in the `IXmlRepository`.)





Key Creation / CreateNewKey

In the implementation of `CreateNewKey`, the `IAuthenticatedEncryptorConfiguration` component is used to create a unique `IAuthenticatedEncryptorDescriptor`, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an `IXmlEncryptor` (if required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the `IXmlRepository`. (If no `IXmlEncryptor` is configured, the unencrypted document is persisted in the `IXmlRepository`.)



Key Retrieval / GetAllKeys

In the implementation of `GetAllKeys`, the XML documents representing keys and revocations are read from the underlying `IXmlRepository`. If these documents are encrypted, the system will automatically decrypt them. `XmlKeyManager` creates the appropriate `IAuthenticatedEncryptorDescriptorDeserializer` instances to deserialize the documents back into `IAuthenticatedEncryptorDescriptor` instances, which are then wrapped in individual `IKey` instances. This collection of `IKey` instances is returned to the caller.

Further information on the particular XML elements can be found in the key storage format document.

IXmlRepository

The `IXmlRepository` interface represents a type that can persist XML to and retrieve XML from a backing store. It exposes two APIs:

- `GetAllElements` : `IReadOnlyCollection<XElement>`
- `StoreElement(XElement element, string friendlyName)`

Implementations of `IXmlRepository` don't need to parse the XML passing through them. They should treat the XML documents as opaque and let higher layers worry about generating and parsing the documents.

There are four built-in concrete types which implement `IXmlRepository`:

- FileSystemXmlRepository
- RegistryXmlRepository
- AzureStorage.AzureBlobXmlRepository
- RedisXmlRepository

- FileSystemXmlRepository
- RegistryXmlRepository
- AzureStorage.AzureBlobXmlRepository
- RedisXmlRepository

See the key storage providers document for more information.

Registering a custom `IXmlRepository` is appropriate when using a different backing store (for example, Azure Table Storage).

To change the default repository application-wide, register a custom `IXmlRepository` instance:

```
services.Configure<KeyManagementOptions>(options => options.XmlRepository = new MyCustomXmlRepository());
```

```
services.AddSingleton<IXmlRepository>(new MyCustomXmlRepository());
```

IXmlEncryptor

The `IXmlEncryptor` interface represents a type that can encrypt a plaintext XML element. It exposes a single API:

- Encrypt(XElement plaintextElement) : EncryptedXmlInfo

If a serialized `IAuthenticatedEncryptorDescriptor` contains any elements marked as "requires encryption", then `XmlKeyManager` will run those elements through the configured `IXmlEncryptor`'s `Encrypt` method, and it will persist the enciphered element rather than the plaintext element to the `IXmlRepository`. The output of the `Encrypt` method is an `EncryptedXmlInfo` object. This object is a wrapper which contains both the resultant enciphered `XElement` and the Type which represents an `IXmlDecryptor` which can be used to decipher the corresponding element.

There are four built-in concrete types which implement `IXmlEncryptor`:

- CertificateXmlEncryptor
- DpapiNGXmlEncryptor
- DpapiXmlEncryptor
- NullXmlEncryptor

See the key encryption at rest document for more information.

To change the default key-encryption-at-rest mechanism application-wide, register a custom `IXmlEncryptor` instance:

```
services.Configure<KeyManagementOptions>(options => options.XmlEncryptor = new MyCustomXmlEncryptor());
```

```
services.AddSingleton<IXmlEncryptor>(new MyCustomXmlEncryptor());
```

IXmlDecryptor

The `IXmlDecryptor` interface represents a type that knows how to decrypt an `XElement` that was enciphered via an `IXmlEncryptor`. It exposes a single API:

- Decrypt(XElement encryptedElement) : XElement

The `Decrypt` method undoes the encryption performed by `IXmlEncryptor.Encrypt`. Generally, each concrete `IXmlEncryptor` implementation will have a corresponding concrete `IXmlDecryptor` implementation.

Types which implement `IXmlDecryptor` should have one of the following two public constructors:

- .ctor(IServiceProvider)
- .ctor()

> **NOTE**
> The `IServiceProvider` passed to the constructor may be null.

IKeyEscrowSink

The `IKeyEscrowSink` interface represents a type that can perform escrow of sensitive information. Recall that serialized descriptors might contain sensitive information (such as cryptographic material), and this is what led to the introduction of the IXmlEncryptor type in the first place. However, accidents happen, and key rings can be deleted or become corrupted.

The escrow interface provides an emergency escape hatch, allowing access to the raw serialized XML before it's transformed by any configured IXmlEncryptor. The interface exposes a single API:

- Store(Guid keyId, XElement element)

It's up to the `IKeyEscrowSink` implementation to handle the provided element in a secure manner consistent with business policy. One possible implementation could be for the escrow sink to encrypt the XML element using a known corporate X.509 certificate where the certificate's private key has been escrowed; the `CertificateXmlEncryptor` type can assist with this. The `IKeyEscrowSink` implementation is also responsible for persisting the provided element appropriately.

By default no escrow mechanism is enabled, though server administrators can configure this globally. It can also be configured programmatically via the `IDataProtectionBuilder.AddKeyEscrowSink` method as shown in the sample below. The `AddKeyEscrowSink` method overloads mirror the `IServiceCollection.AddSingleton` and `IServiceCollection.AddInstance` overloads, as `IKeyEscrowSink` instances are intended to be singletons. If multiple `IKeyEscrowSink` instances are registered, each one will be called during key generation, so keys can be escrowed to multiple mechanisms simultaneously.

There's no API to read material from an `IKeyEscrowSink` instance. This is consistent with the design theory of the escrow mechanism: it's intended to make the key material accessible to a trusted authority, and since the application is itself not a trusted authority, it shouldn't have access to its own escrowed material.

The following sample code demonstrates creating and registering an `IKeyEscrowSink` where keys are escrowed such that only members of "CONTOSODomain Admins" can recover them.

NOTE

To run this sample, you must be on a domain-joined Windows 8 / Windows Server 2012 machine, and the domain controller must be Windows Server 2012 or later.

```
using System;
using System.IO;
using System.Xml.Linq;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.XmlEncryption;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi()
            .AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
        var services = serviceCollection.BuildServiceProvider();

        // get a reference to the key manager and force a new key to be generated
        Console.WriteLine("Generating new key...");
        var keyManager = services.GetService<IKeyManager>();
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddDays(7));
    }

    // A key escrow sink where keys are escrowed such that they
    // can be read by members of the CONTOSO\Domain Admins group.
    private class MyKeyEscrowSink : IKeyEscrowSink
    {
        private readonly IXmlEncryptor _escrowEncryptor;

        public MyKeyEscrowSink(IServiceProvider services)
        {
            // Assuming I'm on a machine that's a member of the CONTOSO
            // domain, I can use the Domain Admins SID to generate an
            // encrypted payload that only they can read. Sample SID from
            // https://technet.microsoft.com/library/cc778824(v=ws.10).aspx.
            _escrowEncryptor = new DpapiNGXmlEncryptor(
                "SID=S-1-5-21-1004336348-1177238915-682003330-512",
                DpapiNGProtectionDescriptorFlags.None,
                new LoggerFactory());
        }

        public void Store(Guid keyId, XElement element)
        {
            // Encrypt the key element to the escrow encryptor.
            var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);

            // A real implementation would save the escrowed key to a
            // write-only file share or some other stable storage, but
            // in this sample we'll just write it out to the console.
            Console.WriteLine($"Escrowing key {keyId}");
            Console.WriteLine(encryptedXmlInfo.EncryptedElement);

            // Note: We cannot read the escrowed key material ourselves.
            // We need to get a member of CONTOSO\Domain Admins to read
            // it for us in the event we need to recover it.
        }
```

```
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Generating new key...
 * Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
 * <encryptedKey>
 *   <!-- This key is encrypted with Windows DPAPI-NG. -->
 *   <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
 *   <value>MIIIfAYJKoZIhvcNAQcDoIIIbTCCCGkCAQ...T5rA4g==</value>
 * </encryptedKey>
 */
```

Miscellaneous ASP.NET Core Data Protection APIs

9/22/2020 • 2 minutes to read • Edit Online

WARNING

Types that implement any of the following interfaces should be thread-safe for multiple callers.

ISecret

The `ISecret` interface represents a secret value, such as cryptographic key material. It contains the following API surface:

- `Length` : `int`
- `Dispose()` : `void`
- `WriteSecretIntoBuffer(ArraySegment<byte> buffer)` : `void`

The `WriteSecretIntoBuffer` method populates the supplied buffer with the raw secret value. The reason this API takes the buffer as a parameter rather than returning a `byte[]` directly is that this gives the caller the opportunity to pin the buffer object, limiting secret exposure to the managed garbage collector.

The `Secret` type is a concrete implementation of `ISecret` where the secret value is stored in in-process memory. On Windows platforms, the secret value is encrypted via CryptProtectMemory.

ASP.NET Core Data Protection implementation

9/22/2020 • 2 minutes to read • Edit Online

Authenticated encryption details in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Calls to IDataProtector.Protect are authenticated encryption operations. The Protect method offers both confidentiality and authenticity, and it's tied to the purpose chain that was used to derive this particular IDataProtector instance from its root IDataProtectionProvider.

IDataProtector.Protect takes a byte[] plaintext parameter and produces a byte[] protected payload, whose format is described below. (There's also an extension method overload which takes a string plaintext parameter and returns a string protected payload. If this API is used the protected payload format will still have the below structure, but it will be base64url-encoded.)

Protected payload format

The protected payload format consists of three primary components:

- A 32-bit magic header that identifies the version of the data protection system.
- A 128-bit key id that identifies the key used to protect this particular payload.
- The remainder of the protected payload is specific to the encryptor encapsulated by this key. In the example below, the key represents an AES-256-CBC + HMACSHA256 encryptor, and the payload is further subdivided as follows:
  - A 128-bit key modifier.
  - A 128-bit initialization vector.
  - 48 bytes of AES-256-CBC output.
  - An HMACSHA256 authentication tag.

A sample protected payload is illustrated below.

```
09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
52 C9 74 A0
```

From the payload format above the first 32 bits, or 4 bytes are the magic header identifying the version (09 F0 C9 F0)

The next 128 bits, or 16 bytes is the key identifier (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

The remainder contains the payload and is specific to the format used.

WARNING

All payloads protected to a given key will begin with the same 20-byte (magic value, key id) header. Administrators can use this fact for diagnostic purposes to approximate when a payload was generated. For example, the payload above corresponds to key {0c819c80-6619-4019-9536-53f8aaffee57}. If after checking the key repository you find that this specific key's activation date was 2015-01-01 and its expiration date was 2015-03-01, then it's reasonable to assume that the payload (if not tampered with) was generated within that window, give or take a small fudge factor on either side.

Subkey derivation and authenticated encryption in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

Most keys in the key ring will contain some form of entropy and will have algorithmic information stating "CBC-mode encryption + HMAC validation" or "GCM encryption + validation". In these cases, we refer to the embedded entropy as the master keying material (or KM) for this key, and we perform a key derivation function to derive the keys that will be used for the actual cryptographic operations.

> **NOTE**
> Keys are abstract, and a custom implementation might not behave as below. If the key provides its own implementation of `IAuthenticatedEncryptor` rather than using one of our built-in factories, the mechanism described in this section no longer applies.

Additional authenticated data and subkey derivation

The `IAuthenticatedEncryptor` interface serves as the core interface for all authenticated encryption operations. Its `Encrypt` method takes two buffers: plaintext and additionalAuthenticatedData (AAD). The plaintext contents flow unchanged the call to `IDataProtector.Protect`, but the AAD is generated by the system and consists of three components:

1. The 32-bit magic header 09 F0 C9 F0 that identifies this version of the data protection system.
2. The 128-bit key id.
3. A variable-length string formed from the purpose chain that created the `IDataProtector` that's performing this operation.

Because the AAD is unique for the tuple of all three components, we can use it to derive new keys from KM instead of using KM itself in all of our cryptographic operations. For every call to `IAuthenticatedEncryptor.Encrypt`, the following key derivation process takes place:

```
( K_E, K_H ) = SP800_108_CTR_HMACSHA512(K_M, AAD, contextHeader || keyModifier)
```

Here, we're calling the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with the following parameters:

- Key derivation key (KDK) = `K_M`
- PRF = HMACSHA512
- label = additionalAuthenticatedData
- context = contextHeader || keyModifier

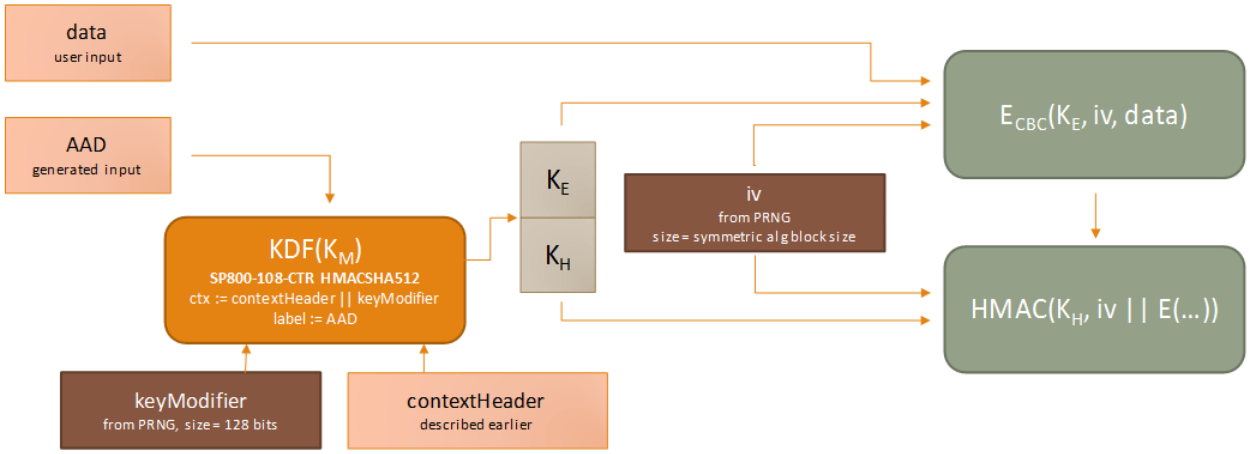
The context header is of variable length and essentially serves as a thumbprint of the algorithms for which we're deriving `K_E` and `K_H`. The key modifier is a 128-bit string randomly generated for each call to `Encrypt` and serves to ensure with overwhelming probability that KE and KH are unique for this specific authentication encryption operation, even if all other input to the KDF is constant.

For CBC-mode encryption + HMAC validation operations, `| K_E |` is the length of the symmetric block cipher key,

and `| K_H |` is the digest size of the HMAC routine. For GCM encryption + validation operations, `| K_H | = 0`.

CBC-mode encryption + HMAC validation

Once `K_E` is generated via the above mechanism, we generate a random initialization vector and run the symmetric block cipher algorithm to encipher the plaintext. The initialization vector and ciphertext are then run through the HMAC routine initialized with the key `K_H` to produce the MAC. This process and the return value is represented graphically below.

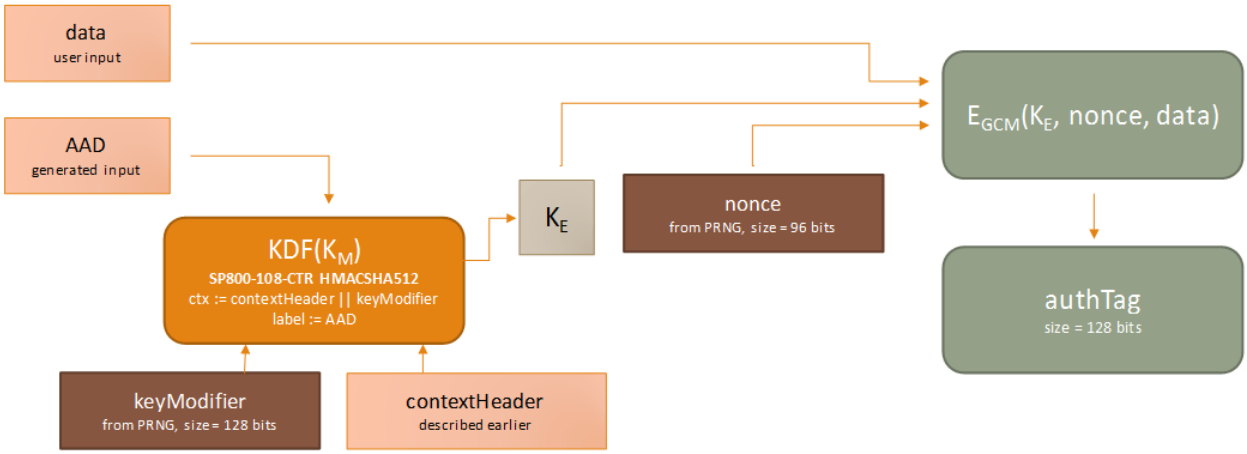



```
output:= keyModifier || iv || E_cbc (K_E,iv,data) || HMAC(K_H, iv || E_cbc (K_E,iv,data))
```

> **NOTE**
>
> The `IDataProtector.Protect` implementation will prepend the magic header and key id to output before returning it to the caller. Because the magic header and key id are implicitly part of AAD, and because the key modifier is fed as input to the KDF, this means that every single byte of the final returned payload is authenticated by the MAC.

Galois/Counter Mode encryption + validation

Once `K_E` is generated via the above mechanism, we generate a random 96-bit nonce and run the symmetric block cipher algorithm to encipher the plaintext and produce the 128-bit authentication tag.




```
output := keyModifier || nonce || E_gcm (K_E,nonce,data) || authTag
```

NOTE

Even though GCM natively supports the concept of AAD, we're still feeding AAD only to the original KDF, opting to pass an empty string into GCM for its AAD parameter. The reason for this is two-fold. First, to support agility we never want to use `K_M` directly as the encryption key. Additionally, GCM imposes very strict uniqueness requirements on its inputs. The probability that the GCM encryption routine is ever invoked on two or more distinct sets of input data with the same (key, nonce) pair must not exceed 2^32. If we fix `K_E` we cannot perform more than 2^32 encryption operations before we run afoul of the 2^-32 limit. This might seem like a very large number of operations, but a high-traffic web server can go through 4 billion requests in mere days, well within the normal lifetime for these keys. To stay compliant of the 2^-32 probability limit, we continue to use a 128-bit key modifier and 96-bit nonce, which radically extends the usable operation count for any given `K_M`. For simplicity of design we share the KDF code path between CBC and GCM operations, and since AAD is already considered in the KDF there's no need to forward it to the GCM routine.

Context headers in ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

Background and theory

In the data protection system, a "key" means an object that can provide authenticated encryption services. Each key is identified by a unique id (a GUID), and it carries with it algorithmic information and entropic material. It's intended that each key carry unique entropy, but the system cannot enforce that, and we also need to account for developers who might change the key ring manually by modifying the algorithmic information of an existing key in the key ring. To achieve our security requirements given these cases the data protection system has a concept of cryptographic agility, which allows securely using a single entropic value across multiple cryptographic algorithms.

Most systems which support cryptographic agility do so by including some identifying information about the algorithm inside the payload. The algorithm's OID is generally a good candidate for this. However, one problem that we ran into is that there are multiple ways to specify the same algorithm: "AES" (CNG) and the managed Aes, AesManaged, AesCryptoServiceProvider, AesCng, and RijndaelManaged (given specific parameters) classes are all actually the same thing, and we'd need to maintain a mapping of all of these to the correct OID. If a developer wanted to provide a custom algorithm (or even another implementation of AES!), they'd have to tell us its OID. This extra registration step makes system configuration particularly painful.

Stepping back, we decided that we were approaching the problem from the wrong direction. An OID tells you what the algorithm is, but we don't actually care about this. If we need to use a single entropic value securely in two different algorithms, it's not necessary for us to know what the algorithms actually are. What we actually care about is how they behave. Any decent symmetric block cipher algorithm is also a strong pseudorandom permutation (PRP): fix the inputs (key, chaining mode, IV, plaintext) and the ciphertext output will with overwhelming probability be distinct from any other symmetric block cipher algorithm given the same inputs. Similarly, any decent keyed hash function is also a strong pseudorandom function (PRF), and given a fixed input set its output will overwhelmingly be distinct from any other keyed hash function.

We use this concept of strong PRPs and PRFs to build up a context header. This context header essentially acts as a stable thumbprint over the algorithms in use for any given operation, and it provides the cryptographic agility needed by the data protection system. This header is reproducible and is used later as part of the subkey derivation process. There are two different ways to build the context header depending on the modes of operation of the underlying algorithms.

CBC-mode encryption + HMAC authentication

The context header consists of the following components:

- [16 bits] The value 00 00, which is a marker meaning "CBC encryption + HMAC authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The key length (in bytes, big-endian) of the HMAC algorithm. (Currently the key size always matches the digest size.)
- [32 bits] The digest size (in bytes, big-endian) of the HMAC algorithm.
- `EncCBC(K_E, IV, "")`, which is the output of the symmetric block cipher algorithm given an empty string input and where IV is an all-zero vector. The construction of `K_E` is described below.

- `MAC(K_H, "")`, which is the output of the HMAC algorithm given an empty string input. The construction of `K_H` is described below.

Ideally, we could pass all-zero vectors for `K_E` and `K_H`. However, we want to avoid the situation where the underlying algorithm checks for the existence of weak keys before performing any operations (notably DES and 3DES), which precludes using a simple or repeatable pattern like an all-zero vector.

Instead, we use the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with a zero-length key, label, and context and HMACSHA512 as the underlying PRF. We derive `| K_E | + | K_H |` bytes of output, then decompose the result into `K_E` and `K_H` themselves. Mathematically, this is represented as follows.

```
( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")
```

Example: AES-192-CBC + HMACSHA256

As an example, consider the case where the symmetric block cipher algorithm is AES-192-CBC and the validation algorithm is HMACSHA256. The system would generate the context header using the following steps.

First, let `( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 192 bits` and `| K_H | = 256 bits` per the specified algorithms. This leads to `K_E = 5BB6..21DD` and `K_H = A04A..00A9` in the example below:

```
5B B6 C9 83 13 78 22 1D 8E 10 73 CA CF 65 8E B0
61 62 42 71 CB 83 21 DD A0 4A 05 00 5B AB C0 A2
49 6F A5 61 E3 E2 49 87 AA 63 55 CD 74 0A DA C4
B7 92 3D BF 59 90 00 A9
```

Next, compute `Enc_CBC (K_E, IV, "")` for AES-192-CBC given `IV = 0*` and `K_E` as above.

```
result := F474B1872B3B53E4721DE19C0841DB6F
```

Next, compute `MAC(K_H, "")` for HMACSHA256 given `K_H` as above.

```
result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C
```

This produces the full context header below:

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36
22 0C
```

This context header is the thumbprint of the authenticated encryption algorithm pair (AES-192-CBC encryption + HMACSHA256 validation). The components, as described above are:

- the marker `(00 00)`
- the block cipher key length `(00 00 00 18)`
- the block cipher block size `(00 00 00 10)`
- the HMAC key length `(00 00 00 20)`
- the HMAC digest size `(00 00 00 20)`
- the block cipher PRP output `(F4 74 - DB 6F)` and
- the HMAC PRF output `(D4 79 - end)`.

> **NOTE**
> The CBC-mode encryption + HMAC authentication context header is built the same way regardless of whether the algorithms implementations are provided by Windows CNG or by managed SymmetricAlgorithm and KeyedHashAlgorithm types. This allows applications running on different operating systems to reliably produce the same context header even though the implementations of the algorithms differ between OSes. (In practice, the KeyedHashAlgorithm doesn't have to be a proper HMAC. It can be any keyed hash algorithm type.)

Example: 3DES-192-CBC + HMACSHA1

First, let `( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 192 bits` and `| K_H | = 160 bits` per the specified algorithms. This leads to `K_E = A219..E2BB` and `K_H = DC4A..B464` in the example below:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

Next, compute `Enc_CBC (K_E, IV, "")` for 3DES-192-CBC given `IV = 0*` and `K_E` as above.

`result := ABB100F81E53E10E`

Next, compute `MAC(K_H, "")` for HMACSHA1 given `K_H` as above.

`result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555`

This produces the full context header which is a thumbprint of the authenticated encryption algorithm pair (3DES-192-CBC encryption + HMACSHA1 validation), shown below:

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

The components break down as follows:

- the marker `(00 00)`
- the block cipher key length `(00 00 00 18)`
- the block cipher block size `(00 00 00 08)`
- the HMAC key length `(00 00 00 14)`
- the HMAC digest size `(00 00 00 14)`
- the block cipher PRP output `(AB B1 - E1 0E)` and
- the HMAC PRF output `(76 EB - end)`.

Galois/Counter Mode encryption + authentication

The context header consists of the following components:

- [16 bits] The value 00 01, which is a marker meaning "GCM encryption + authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The nonce size (in bytes, big-endian) used during authenticated encryption operations. (For our

system, this is fixed at nonce size = 96 bits.)

- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm. (For GCM, this is fixed at block size = 128 bits.)
- [32 bits] The authentication tag size (in bytes, big-endian) produced by the authenticated encryption function. (For our system, this is fixed at tag size = 128 bits.)
- [128 bits] The tag of `Enc_GCM (K_E, nonce, "")`, which is the output of the symmetric block cipher algorithm given an empty string input and where nonce is a 96-bit all-zero vector.

`K_E` is derived using the same mechanism as in the CBC encryption + HMAC authentication scenario. However, since there's no `K_H` in play here, we essentially have `| K_H | = 0`, and the algorithm collapses to the below form.

```
K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")
```

Example: AES-256-GCM

First, let `K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 256 bits`.

```
K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8
```

Next, compute the authentication tag of `Enc_GCM (K_E, nonce, "")` for AES-256-GCM given `nonce = 096` and `K_E` as above.

```
result := E7DCCE66DF855A323A6BB7BD7A59BE45
```

This produces the full context header below:

```
00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59
BE 45
```

The components break down as follows:

- the marker `(00 01)`
- the block cipher key length `(00 00 00 20)`
- the nonce size `(00 00 00 0C)`
- the block cipher block size `(00 00 00 10)`
- the authentication tag size `(00 00 00 10)` and
- the authentication tag from running the block cipher `(E7 DC - end)`.

Key management in ASP.NET Core

9/22/2020 • 6 minutes to read • Edit Online

The data protection system automatically manages the lifetime of master keys used to protect and unprotect payloads. Each key can exist in one of four stages:

- Created - the key exists in the key ring but has not yet been activated. The key shouldn't be used for new Protect operations until sufficient time has elapsed that the key has had a chance to propagate to all machines that are consuming this key ring.
- Active - the key exists in the key ring and should be used for all new Protect operations.
- Expired - the key has run its natural lifetime and should no longer be used for new Protect operations.
- Revoked - the key is compromised and must not be used for new Protect operations.

Created, active, and expired keys may all be used to unprotect incoming payloads. Revoked keys by default may not be used to unprotect payloads, but the application developer can override this behavior if necessary.

> **WARNING**
> The developer might be tempted to delete a key from the key ring (e.g., by deleting the corresponding file from the file system). At that point, all data protected by the key is permanently undecipherable, and there's no emergency override like there's with revoked keys. Deleting a key is truly destructive behavior, and consequently the data protection system exposes no first-class API for performing this operation.

Default key selection

When the data protection system reads the key ring from the backing repository, it will attempt to locate a "default" key from the key ring. The default key is used for new Protect operations.

The general heuristic is that the data protection system chooses the key with the most recent activation date as the default key. (There's a small fudge factor to allow for server-to-server clock skew.) If the key is expired or revoked, and if the application has not disabled automatic key generation, then a new key will be generated with immediate activation per the key expiration and rolling policy below.

The reason the data protection system generates a new key immediately rather than falling back to a different key is that new key generation should be treated as an implicit expiration of all keys that were activated prior to the new key. The general idea is that new keys may have been configured with different algorithms or encryption-at-rest mechanisms than old keys, and the system should prefer the current configuration over falling back.

There's an exception. If the application developer has disabled automatic key generation, then the data protection system must choose something as the default key. In this fallback scenario, the system will choose the non-revoked key with the most recent activation date, with preference given to keys that have had time to propagate to other machines in the cluster. The fallback system may end up choosing an expired default key as a result. The fallback system will never choose a revoked key as the default key, and if the key ring is empty or every key has been revoked then the system will produce an error upon initialization.

Key expiration and rolling

When a key is created, it's automatically given an activation date of { now + 2 days } and an expiration date of {

now + 90 days }. The 2-day delay before activation gives the key time to propagate through the system. That is, it allows other applications pointing at the backing store to observe the key at their next auto-refresh period, thus maximizing the chances that when the key ring does become active it has propagated to all applications that might need to use it.

If the default key will expire within 2 days and if the key ring doesn't already have a key that will be active upon expiration of the default key, then the data protection system will automatically persist a new key to the key ring. This new key has an activation date of { default key's expiration date } and an expiration date of { now + 90 days }. This allows the system to automatically roll keys on a regular basis with no interruption of service.

There might be circumstances where a key will be created with immediate activation. One example would be when the application hasn't run for a time and all keys in the key ring are expired. When this happens, the key is given an activation date of { now } without the normal 2-day activation delay.

The default key lifetime is 90 days, though this is configurable as in the following example.

```
services.AddDataProtection()
       // use 14-day lifetime instead of 90-day lifetime
       .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

An administrator can also change the default system-wide, though an explicit call to `SetDefaultKeyLifetime` will override any system-wide policy. The default key lifetime cannot be shorter than 7 days.

Automatic key ring refresh

When the data protection system initializes, it reads the key ring from the underlying repository and caches it in memory. This cache allows Protect and Unprotect operations to proceed without hitting the backing store. The system will automatically check the backing store for changes approximately every 24 hours or when the current default key expires, whichever comes first.

> **WARNING**
>
> Developers should very rarely (if ever) need to use the key management APIs directly. The data protection system will perform automatic key management as described above.

The data protection system exposes an interface `IKeyManager` that can be used to inspect and make changes to the key ring. The DI system that provided the instance of `IDataProtectionProvider` can also provide an instance of `IKeyManager` for your consumption. Alternatively, you can pull the `IKeyManager` straight from the `IServiceProvider` as in the example below.

Any operation which modifies the key ring (creating a new key explicitly or performing a revocation) will invalidate the in-memory cache. The next call to `Protect` or `Unprotect` will cause the data protection system to reread the key ring and recreate the cache.

The sample below demonstrates using the `IKeyManager` interface to inspect and manipulate the key ring, including revoking existing keys and generating a new key manually.

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
```

```
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // perform a protect operation to force the system to put at least
        // one key in the key ring
        services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
        Console.WriteLine("Performed a protect operation.");
        Thread.Sleep(2000);

        // get a reference to the key manager
        var keyManager = services.GetService<IKeyManager>();

        // list all keys in the key ring
        var allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
        }

        // revoke all keys in the key ring
        keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation reason here.");
        Console.WriteLine("Revoked all existing keys.");

        // add a new key to the key ring with immediate activation and a 1-month expiration
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddMonths(1));
        Console.WriteLine("Added a new key.");

        // list all keys in the key ring
        allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created = {key.CreationDate:u}, IsRevoked =
{key.IsRevoked}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Performed a protect operation.
 * The key ring contains 1 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = False
 * Revoked all existing keys.
 * Added a new key.
 * The key ring contains 2 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18 22:20:49Z, IsRevoked = True
 * Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18 22:20:51Z, IsRevoked = False
 */
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

Key storage

The data protection system has a heuristic whereby it attempts to deduce an appropriate key storage location and

encryption-at-rest mechanism automatically. The key persistence mechanism is also configurable by the app developer. The following documents discuss the in-box implementations of these mechanisms:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Key storage providers in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

The data protection system employs a discovery mechanism by default to determine where cryptographic keys should be persisted. The developer can override the default discovery mechanism and manually specify the location.

> **WARNING**
> If you specify an explicit key persistence location, the data protection system deregisters the default key encryption at rest mechanism, so keys are no longer encrypted at rest. It's recommended that you additionally specify an explicit key encryption mechanism for production deployments.

File system

To configure a file system-based key repository, call the PersistKeysToFileSystem configuration routine as shown below. Provide a DirectoryInfo pointing to the repository where keys should be stored:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
}
```

The `DirectoryInfo` can point to a directory on the local machine, or it can point to a folder on a network share. If pointing to a directory on the local machine (and the scenario is that only apps on the local machine require access to use this repository), consider using Windows DPAPI (on Windows) to encrypt the keys at rest. Otherwise, consider using an X.509 certificate to encrypt keys at rest.

Azure Storage

The Microsoft.AspNetCore.DataProtection.AzureStorage package allows storing data protection keys in Azure Blob Storage. Keys can be shared across several instances of a web app. Apps can share authentication cookies or CSRF protection across multiple servers.

To configure the Azure Blob Storage provider, call one of the PersistKeysToAzureBlobStorage overloads.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blob URI including SAS token>"));
}
```

If the web app is running as an Azure service, authentication tokens can be automatically created using Microsoft.Azure.Services.AppAuthentication.

```
var tokenProvider = new AzureServiceTokenProvider();
var token = await tokenProvider.GetAccessTokenAsync("https://storage.azure.com/");
var credentials = new StorageCredentials(new TokenCredential(token));
var storageAccount = new CloudStorageAccount(credentials, "mystorageaccount", "core.windows.net", useHttps:
true);
var client = storageAccount.CreateCloudBlobClient();
var container = client.GetContainerReference("my-key-container");

// optional - provision the container automatically
await container.CreateIfNotExistsAsync();

services.AddDataProtection()
    .PersistKeysToAzureBlobStorage(container, "keys.xml");
```

See more details about configuring service-to-service authentication.

Redis

The Microsoft.AspNetCore.DataProtection.StackExchangeRedis package allows storing data protection keys in a Redis cache. Keys can be shared across several instances of a web app. Apps can share authentication cookies or CSRF protection across multiple servers.

The Microsoft.AspNetCore.DataProtection.Redis package allows storing data protection keys in a Redis cache. Keys can be shared across several instances of a web app. Apps can share authentication cookies or CSRF protection across multiple servers.

To configure on Redis, call one of the PersistKeysToStackExchangeRedis overloads:

```
public void ConfigureServices(IServiceCollection services)
{
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToStackExchangeRedis(redis, "DataProtection-Keys");
}
```

To configure on Redis, call one of the PersistKeysToRedis overloads:

```
public void ConfigureServices(IServiceCollection services)
{
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToRedis(redis, "DataProtection-Keys");
}
```

For more information, see the following topics:

- StackExchange.Redis ConnectionMultiplexer
- Azure Redis Cache
- ASP.NET Core DataProtection samples

Registry

Only applies to Windows deployments.

Sometimes the app might not have write access to the file system. Consider a scenario where an app is running as a virtual service account (such as *w3wp.exe*'s app pool identity). In these cases, the administrator can provision a registry key that's accessible by the service account identity. Call the PersistKeysToRegistry extension

method as shown below. Provide a RegistryKey pointing to the location where cryptographic keys should be stored:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys"));
}
```

> **IMPORTANT**
>
> We recommend using Windows DPAPI to encrypt the keys at rest.

Entity Framework Core

The Microsoft.AspNetCore.DataProtection.EntityFrameworkCore package provides a mechanism for storing data protection keys to a database using Entity Framework Core. The `Microsoft.AspNetCore.DataProtection.EntityFrameworkCore` NuGet package must be added to the project file, it's not part of the Microsoft.AspNetCore.App metapackage.

With this package, keys can be shared across multiple instances of a web app.

To configure the EF Core provider, call the PersistKeysToDbContext<TContext> method:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    // Add a DbContext to store your Database Keys
    services.AddDbContext<MyKeysContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("MyKeysConnection")));

    // using Microsoft.AspNetCore.DataProtection;
    services.AddDataProtection()
        .PersistKeysToDbContext<MyKeysContext>();

    services.AddDefaultIdentity<IdentityUser>()
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The generic parameter, `TContext`, must inherit from DbContext and implement IDataProtectionKeyContext:

```
using Microsoft.AspNetCore.DataProtection.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using WebApp1.Data;

namespace WebApp1
{
    class MyKeysContext : DbContext, IDataProtectionKeyContext
    {
        // A recommended constructor overload when using EF Core
        // with dependency injection.
        public MyKeysContext(DbContextOptions<MyKeysContext> options)
            : base(options) { }

        // This maps to the table that stores keys.
        public DbSet<DataProtectionKey> DataProtectionKeys { get; set; }
    }
}
```

Create the `DataProtectionKeys` table.

- Visual Studio
- .NET Core CLI

Execute the following commands in the **Package Manager Console** (PMC) window:

```
Add-Migration AddDataProtectionKeys -Context MyKeysContext
Update-Database -Context MyKeysContext
```

`MyKeysContext` is the `DbContext` defined in the preceding code sample. If you're using a `DbContext` with a different name, substitute your `DbContext` name for `MyKeysContext`.

The `DataProtectionKeys` class/entity adopts the structure shown in the following table.

PROPERTY/FIELD	CLR TYPE	SQL TYPE
`Id`	`int`	`int`, PK, `IDENTITY(1,1)`, not null
`FriendlyName`	`string`	`nvarchar(MAX)`, null
`Xml`	`string`	`nvarchar(MAX)`, null

Custom key repository

If the in-box mechanisms aren't appropriate, the developer can specify their own key persistence mechanism by providing a custom IXmlRepository.

Key encryption at rest in Windows and Azure using ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

The data protection system employs a discovery mechanism by default to determine how cryptographic keys should be encrypted at rest. The developer can override the discovery mechanism and manually specify how keys should be encrypted at rest.

> **WARNING**
> If you specify an explicit key persistence location, the data protection system deregisters the default key encryption at rest mechanism. Consequently, keys are no longer encrypted at rest. We recommend that you specify an explicit key encryption mechanism for production deployments. The encryption-at-rest mechanism options are described in this topic.

Azure Key Vault

To store keys in Azure Key Vault, configure the system with ProtectKeysWithAzureKeyVault in the `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
        .ProtectKeysWithAzureKeyVault("<keyIdentifier>", "<clientId>", "<clientSecret>");
}
```

For more information, see Configure ASP.NET Core Data Protection: ProtectKeysWithAzureKeyVault.

Windows DPAPI

Only applies to Windows deployments.

When Windows DPAPI is used, key material is encrypted with CryptProtectData before being persisted to storage. DPAPI is an appropriate encryption mechanism for data that's never read outside of the current machine (though it's possible to back these keys up to Active Directory; see DPAPI and Roaming Profiles). To configure DPAPI key-at-rest encryption, call one of the ProtectKeysWithDpapi extension methods:

```
public void ConfigureServices(IServiceCollection services)
{
    // Only the local user account can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi();
}
```

If `ProtectKeysWithDpapi` is called with no parameters, only the current Windows user account can decipher the persisted key ring. You can optionally specify that any user account on the machine (not just the current user account) be able to decipher the key ring:

```
public void ConfigureServices(IServiceCollection services)
{
    // All user accounts on the machine can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi(protectToLocalMachine: true);
}
```

X.509 certificate

If the app is spread across multiple machines, it may be convenient to distribute a shared X.509 certificate across the machines and configure the hosted apps to use the certificate for encryption of keys at rest:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
}
```

Due to .NET Framework limitations, only certificates with CAPI private keys are supported. See the content below for possible workarounds to these limitations.

Windows DPAPI-NG

This mechanism is available only on Windows 8/Windows Server 2012 or later.

Beginning with Windows 8, Windows OS supports DPAPI-NG (also called CNG DPAPI). For more information, see About CNG DPAPI.

The principal is encoded as a protection descriptor rule. In the following example that calls ProtectKeysWithDpapiNG, only the domain-joined user with the specified SID can decrypt the key ring:

```
public void ConfigureServices(IServiceCollection services)
{
    // Uses the descriptor rule "SID=S-1-5-21-..."
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
        flags: DpapiNGProtectionDescriptorFlags.None);
}
```

There's also a parameterless overload of `ProtectKeysWithDpapiNG`. Use this convenience method to specify the rule "SID={CURRENT_ACCOUNT_SID}", where *CURRENT_ACCOUNT_SID* is the SID of the current Windows user account:

```
public void ConfigureServices(IServiceCollection services)
{
    // Use the descriptor rule "SID={current account SID}"
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG();
}
```

In this scenario, the AD domain controller is responsible for distributing the encryption keys used by the DPAPI-NG operations. The target user can decipher the encrypted payload from any domain-joined machine (provided that the process is running under their identity).

Certificate-based encryption with Windows DPAPI-NG

If the app is running on Windows 8.1/Windows Server 2012 R2 or later, you can use Windows DPAPI-NG to perform certificate-based encryption. Use the rule descriptor string "CERTIFICATE=HashId:THUMBPRINT", where *THUMBPRINT* is the hex-encoded SHA1 thumbprint of the certificate:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2...B5AEA2A9BD2575A0",
            flags: DpapiNGProtectionDescriptorFlags.None);
}
```

Any app pointed at this repository must be running on Windows 8.1/Windows Server 2012 R2 or later to decipher the keys.

Custom key encryption

If the in-box mechanisms aren't appropriate, the developer can specify their own key encryption mechanism by providing a custom IXmlEncryptor.

Key immutability and key settings in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Once an object is persisted to the backing store, its representation is forever fixed. New data can be added to the backing store, but existing data can never be mutated. The primary purpose of this behavior is to prevent data corruption.

One consequence of this behavior is that once a key is written to the backing store, it's immutable. Its creation, activation, and expiration dates can never be changed, though it can revoked by using `IKeyManager`. Additionally, its underlying algorithmic information, master keying material, and encryption at rest properties are also immutable.

If the developer changes any setting that affects key persistence, those changes won't go into effect until the next time a key is generated, either via an explicit call to `IKeyManager.CreateNewKey` or via the data protection system's own automatic key generation behavior. The settings that affect key persistence are as follows:

- The default key lifetime
- The key encryption at rest mechanism
- The algorithmic information contained within the key

If you need these settings to kick in earlier than the next automatic key rolling time, consider making an explicit call to `IKeyManager.CreateNewKey` to force the creation of a new key. Remember to provide an explicit activation date ({ now + 2 days } is a good rule of thumb to allow time for the change to propagate) and expiration date in the call.

> **TIP**
> All applications touching the repository should specify the same settings with the `IDataProtectionBuilder` extension methods. Otherwise, the properties of the persisted key will be dependent on the particular application that invoked the key generation routines.

Key storage format in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

Objects are stored at rest in XML representation. The default directory for key storage is:

- Windows: *%LOCALAPPDATA%\ASP.NET\DataProtection-Keys*
- macOS / Linux: *$HOME/.aspnet/DataProtection-Keys*

The <key> element

Keys exist as top-level objects in the key repository. By convention keys have the filename **key-{guid}.xml**, where {guid} is the id of the key. Each such file contains a single key. The format of the file is as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<key id="80732141-ec8f-4b80-af9c-c4d2d1ff8901" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor deserializerType="{deserializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

The <key> element contains the following attributes and child elements:

- The key id. This value is treated as authoritative; the filename is simply a nicety for human readability.
- The version of the <key> element, currently fixed at 1.
- The key's creation, activation, and expiration dates.
- A <descriptor> element, which contains information on the authenticated encryption implementation contained within this key.

In the above example, the key's id is {80732141-ec8f-4b80-af9c-c4d2d1ff8901}, it was created and activated on March 19, 2015, and it has a lifetime of 90 days. (Occasionally the activation date might be slightly before the creation date as in this example. This is due to a nit in how the APIs work and is harmless in practice.)

The <descriptor> element

The outer <descriptor> element contains an attribute deserializerType, which is the assembly-qualified name of a type which implements IAuthenticatedEncryptorDescriptorDeserializer. This type is responsible for reading the inner <descriptor> element and for parsing the information contained within.

The particular format of the <descriptor> element depends on the authenticated encryptor implementation encapsulated by the key, and each deserializer type expects a slightly different format for this. In general, though,

this element will contain algorithmic information (names, types, OIDs, or similar) and secret key material. In the above example, the descriptor specifies that this key wraps AES-256-CBC encryption + HMACSHA256 validation.

The <encryptedSecret> element

An **<encryptedSecret>** element which contains the encrypted form of the secret key material may be present if encryption of secrets at rest is enabled. The attribute `decryptorType` is the assembly-qualified name of a type which implements IXmlDecryptor. This type is responsible for reading the inner **<encryptedKey>** element and decrypting it to recover the original plaintext.

As with `<descriptor>`, the particular format of the `<encryptedSecret>` element depends on the at-rest encryption mechanism in use. In the above example, the master key is encrypted using Windows DPAPI per the comment.

The <revocation> element

Revocations exist as top-level objects in the key repository. By convention revocations have the filename **revocation-{timestamp}.xml** (for revoking all keys before a specific date) or **revocation-{guid}.xml** (for revoking a specific key). Each file contains a single <revocation> element.

For revocations of individual keys, the file contents will be as below.

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="eb4fc299-8808-409d-8a34-23fc83d026c9" />
  <reason>human-readable reason</reason>
</revocation>
```

In this case, only the specified key is revoked. If the key id is "*", however, as in the below example, all keys whose creation date is prior to the specified revocation date are revoked.

```
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

The <reason> element is never read by the system. It's simply a convenient place to store a human-readable reason for revocation.

Ephemeral data protection providers in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

There are scenarios where an application needs a throwaway `IDataProtectionProvider`. For example, the developer might just be experimenting in a one-off console application, or the application itself is transient (it's scripted or a unit test project). To support these scenarios the Microsoft.AspNetCore.DataProtection package includes a type `EphemeralDataProtectionProvider`. This type provides a basic implementation of `IDataProtectionProvider` whose key repository is held solely in-memory and isn't written out to any backing store.

Each instance of `EphemeralDataProtectionProvider` uses its own unique master key. Therefore, if an `IDataProtector` rooted at an `EphemeralDataProtectionProvider` generates a protected payload, that payload can only be unprotected by an equivalent `IDataProtector` (given the same purpose chain) rooted at the same `EphemeralDataProtectionProvider` instance.

The following sample demonstrates instantiating an `EphemeralDataProtectionProvider` and using it to protect and unprotect data.

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";

        // create an ephemeral provider and demonstrate that it can round-trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); // THROWS
    }
}

/*
* SAMPLE OUTPUT
*
* Enter input: Hello!
* Protect returned: CfDJ8AAAAAAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
* Unprotect returned: Hello!
* << throws CryptographicException >>
*/
```

Compatibility in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

- Replacing ASP.NET <machineKey> in ASP.NET Core
- `Microsoft.AspNetCore.DataProtection` 3.1 not compatible with Azure function apps. For more information, see this GitHub issue

Replace the ASP.NET machineKey in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

The implementation of the `<machineKey>` element in ASP.NET is replaceable. This allows most calls to ASP.NET cryptographic routines to be routed through a replacement data protection mechanism, including the new data protection system.

Package installation

> **NOTE**
> The new data protection system can only be installed into an existing ASP.NET application targeting .NET 4.5.1 or later. Installation will fail if the application targets .NET 4.5 or lower.

To install the new data protection system into an existing ASP.NET 4.5.1+ project, install the package Microsoft.AspNetCore.DataProtection.SystemWeb. This will instantiate the data protection system using the default configuration settings.

When you install the package, it inserts a line into *Web.config* that tells ASP.NET to use it for most cryptographic operations, including forms authentication, view state, and calls to MachineKey.Protect. The line that's inserted reads as follows.

```
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

> **TIP**
> You can tell if the new data protection system is active by inspecting fields like `__VIEWSTATE`, which should begin with "CfDJ8" as in the example below. "CfDJ8" is the base64 representation of the magic "09 F0 C9 F0" header that identifies a payload protected by the data protection system.

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="CfDJ8AWPr2EQPTBGs3L2GCZOpk...">
```

Package configuration

The data protection system is instantiated with a default zero-setup configuration. However, since by default keys are persisted to the local file system, this won't work for applications which are deployed in a farm. To resolve this, you can provide configuration by creating a type which subclasses DataProtectionStartup and overrides its ConfigureServices method.

Below is an example of a custom data protection startup type which configured both where keys are persisted and how they're encrypted at rest. It also overrides the default app isolation policy by providing its own application name.

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.SystemWeb;
using Microsoft.Extensions.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
        public override void ConfigureServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("my-app")
                .PersistKeysToFileSystem(new DirectoryInfo(@"\\server\share\myapp-keys\"))
                .ProtectKeysWithCertificate("thumbprint");
        }
    }
}
```

TIP

You can also use `<machineKey applicationName="my-app" ... />` in place of an explicit call to SetApplicationName. This is a convenience mechanism to avoid forcing the developer to create a DataProtectionStartup-derived type if all they wanted to configure was setting the application name.

To enable this custom configuration, go back to Web.config and look for the `<appSettings>` element that the package install added to the config file. It will look like the following markup:

```
<appSettings>
  <!--
  If you want to customize the behavior of the ASP.NET Core Data Protection stack, set the
  "aspnet:dataProtectionStartupType" switch below to be the fully-qualified name of a
  type which subclasses Microsoft.AspNetCore.DataProtection.SystemWeb.DataProtectionStartup.
  -->
  <add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>
```

Fill in the blank value with the assembly-qualified name of the DataProtectionStartup-derived type you just created. If the name of the application is DataProtectionDemo, this would look like the below.

```
<add key="aspnet:dataProtectionStartupType"
     value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo" />
```

The newly-configured data protection system is now ready for use inside the application.

Safe storage of app secrets in development in ASP.NET Core

9/22/2020 • 15 minutes to read • Edit Online

By Rick Anderson, Kirk Larkin, Daniel Roth, and Scott Addie

View or download sample code (how to download)

This document explains techniques for storing and retrieving sensitive data during development of an ASP.NET Core app on a development machine. Never store passwords or other sensitive data in source code. Production secrets shouldn't be used for development or test. Secrets shouldn't be deployed with the app. Instead, secrets should be made available in the production environment through a controlled means like environment variables, Azure Key Vault, etc. You can store and protect Azure test and production secrets with the Azure Key Vault configuration provider.

Environment variables

Environment variables are used to avoid storage of app secrets in code or in local configuration files. Environment variables override configuration values for all previously specified configuration sources.

Consider an ASP.NET Core web app in which **Individual User Accounts** security is enabled. A default database connection string is included in the project's *appsettings.json* file with the key `DefaultConnection`. The default connection string is for LocalDB, which runs in user mode and doesn't require a password. During app deployment, the `DefaultConnection` key value can be overridden with an environment variable's value. The environment variable may store the complete connection string with sensitive credentials.

> **WARNING**
> Environment variables are generally stored in plain, unencrypted text. If the machine or process is compromised, environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may be required.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

Secret Manager

The Secret Manager tool stores sensitive data during the development of an ASP.NET Core project. In this context, a piece of sensitive data is an app secret. App secrets are stored in a separate location from the project tree. The app secrets are associated with a specific project or shared across several projects. The app secrets aren't checked into source control.

> **WARNING**
> The Secret Manager tool doesn't encrypt the stored secrets and shouldn't be treated as a trusted store. It's for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

How the Secret Manager tool works

The Secret Manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. The values are stored in a JSON configuration file in a system-protected user profile folder on the local machine:

- Windows
- Linux / macOS

File system path:

```
%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json
```

In the preceding file paths, replace `<user_secrets_id>` with the `UserSecretsId` value specified in the *.csproj* file.

Don't write code that depends on the location or format of data saved with the Secret Manager tool. These implementation details may change. For example, the secret values aren't encrypted, but could be in the future.

Enable secret storage

The Secret Manager tool operates on project-specific configuration settings stored in your user profile.

The Secret Manager tool includes an `init` command in .NET Core SDK 3.0.100 or later. To use user secrets, run the following command in the project directory:

```
dotnet user-secrets init
```

The preceding command adds a `UserSecretsId` element within a `PropertyGroup` of the *.csproj* file. By default, the inner text of `UserSecretsId` is a GUID. The inner text is arbitrary, but is unique to the project.

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

In Visual Studio, right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a `UserSecretsId` element, populated with a GUID, to the *.csproj* file.

Set a secret

Define an app secret consisting of a key and its value. The secret is associated with the project's `UserSecretsId` value. For example, run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

In the preceding example, the colon denotes that `Movies` is an object literal with a `ServiceApiKey` property.

The Secret Manager tool can be used from other directories too. Use the `--project` option to supply the file system path at which the *.csproj* file exists. For example:

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345" --project "C:\apps\WebApp1\src\WebApp1"
```

JSON structure flattening in Visual Studio

Visual Studio's **Manage User Secrets** gesture opens a *secrets.json* file in the text editor. Replace the contents of *secrets.json* with the key-value pairs to be stored. For example:

```
{
  "Movies": {
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
    "ServiceApiKey": "12345"
  }
}
```

The JSON structure is flattened after modifications via `dotnet user-secrets remove` or `dotnet user-secrets set`. For example, running `dotnet user-secrets remove "Movies:ConnectionString"` collapses the `Movies` object literal. The modified file resembles the following:

```
{
  "Movies:ServiceApiKey": "12345"
}
```

Set multiple secrets

A batch of secrets can be set by piping JSON to the `set` command. In the following example, the *input.json* file's contents are piped to the `set` command.

- Windows
- Linux / macOS

Open a command shell, and execute the following command:

```
type .\input.json | dotnet user-secrets set
```

Access a secret

The ASP.NET Core Configuration API provides access to Secret Manager secrets.

The user secrets configuration source is automatically added in development mode when the project calls CreateDefaultBuilder to initialize a new instance of the host with preconfigured defaults. `CreateDefaultBuilder` calls AddUserSecrets when the EnvironmentName is Development:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

When `CreateDefaultBuilder` isn't called, add the user secrets configuration source explicitly by calling AddUserSecrets. Call `AddUserSecrets` only when the app runs in the Development environment, as shown in the following example:

```
var host = new HostBuilder()
    .ConfigureAppConfiguration((hostContext, builder) =>
    {
        // Add other providers for JSON, etc.

        if (hostContext.HostingEnvironment.IsDevelopment())
        {
            builder.AddUserSecrets<Program>();
        }
    })
    .Build();
```

User secrets can be retrieved via the `Configuration` API:

```
public class Startup
{
    private string _moviesApiKey = null;

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        _moviesApiKey = Configuration["Movies:ServiceApiKey"];
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            var result = string.IsNullOrEmpty(_moviesApiKey) ? "Null" : "Not Null";
            await context.Response.WriteAsync($"Secret is {result}");
        });
    }
}
```

Map secrets to a POCO

Mapping an entire object literal to a POCO (a simple .NET class with properties) is useful for aggregating related properties.

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

To map the preceding secrets to a POCO, use the `Configuration` API's object graph binding feature. The following code binds to a custom `MovieSettings` POCO and accesses the `ServiceApiKey` property value:

```
var moviesConfig = Configuration.GetSection("Movies")
                                .Get<MovieSettings>();
_moviesApiKey = moviesConfig.ServiceApiKey;
```

The `Movies:ConnectionString` and `Movies:ServiceApiKey` secrets are mapped to the respective properties in `MovieSettings`:

```
public class MovieSettings
{
    public string ConnectionString { get; set; }

    public string ServiceApiKey { get; set; }
}
```

String replacement with secrets

Storing passwords in plain text is insecure. For example, a database connection string stored in *appsettings.json* may include a password for the specified user:

```
{
  "ConnectionStrings": {
    "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User
Id=johndoe;Password=pass123;MultipleActiveResultSets=true"
  }
}
```

A more secure approach is to store the password as a secret. For example:

```
dotnet user-secrets set "DbPassword" "pass123"
```

Remove the `Password` key-value pair from the connection string in *appsettings.json*. For example:

```
{
  "ConnectionStrings": {
    "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User
Id=johndoe;MultipleActiveResultSets=true"
  }
}
```

The secret's value can be set on a SqlConnectionStringBuilder object's Password property to complete the connection string:

```
public class Startup
{
    private string _connection = null;

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        var builder = new SqlConnectionStringBuilder(
            Configuration.GetConnectionString("Movies"));
        builder.Password = Configuration["DbPassword"];
        _connection = builder.ConnectionString;
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"DB Connection: {_connection}");
        });
    }
}
```

List the secrets

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets list
```

The following output appears:

```
Movies:ConnectionString = Server=(localdb)\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true
Movies:ServiceApiKey = 12345
```

In the preceding example, a colon in the key names denotes the object hierarchy within *secrets.json*.

Remove a single secret

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets remove "Movies:ConnectionString"
```

The app's *secrets.json* file was modified to remove the key-value pair associated with the `MoviesConnectionString` key:

```
{
  "Movies": {
    "ServiceApiKey": "12345"
  }
}
```

`dotnet user-secrets list` displays the following message:

```
Movies:ServiceApiKey = 12345
```

Remove all secrets

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets clear
```

All user secrets for the app have been deleted from the *secrets.json* file:

```
{}
```

Running `dotnet user-secrets list` displays the following message:

```
No secrets configured for this application.
```

Additional resources

- See this issue for information on accessing Secret Manager from IIS.
- Configuration in ASP.NET Core
- Azure Key Vault Configuration Provider in ASP.NET Core

By Rick Anderson, Daniel Roth, and Scott Addie

View or download sample code (how to download)

This document explains techniques for storing and retrieving sensitive data during development of an ASP.NET Core app on a development machine. Never store passwords or other sensitive data in source code. Production secrets shouldn't be used for development or test. Secrets shouldn't be deployed with the app. Instead, secrets should be made available in the production environment through a controlled means like environment variables, Azure Key Vault, etc. You can store and protect Azure test and production secrets with the Azure Key Vault configuration provider.

Environment variables

Environment variables are used to avoid storage of app secrets in code or in local configuration files. Environment variables override configuration values for all previously specified configuration sources.

Consider an ASP.NET Core web app in which **Individual User Accounts** security is enabled. A default database connection string is included in the project's *appsettings.json* file with the key `DefaultConnection`. The default connection string is for LocalDB, which runs in user mode and doesn't require a password. During app deployment, the `DefaultConnection` key value can be overridden with an environment variable's value. The environment variable may store the complete connection string with sensitive credentials.

> **WARNING**
> Environment variables are generally stored in plain, unencrypted text. If the machine or process is compromised, environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may be required.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

Secret Manager

The Secret Manager tool stores sensitive data during the development of an ASP.NET Core project. In this context, a piece of sensitive data is an app secret. App secrets are stored in a separate location from the project tree. The app secrets are associated with a specific project or shared across several projects. The app secrets aren't checked into source control.

> **WARNING**
> The Secret Manager tool doesn't encrypt the stored secrets and shouldn't be treated as a trusted store. It's for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

How the Secret Manager tool works

The Secret Manager tool abstracts away the implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. The values are stored in a JSON configuration file in a system-protected user profile folder on the local machine:

- Windows
- Linux / macOS

File system path:

```
%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json
```

In the preceding file paths, replace `<user_secrets_id>` with the `UserSecretsId` value specified in the *.csproj* file.

Don't write code that depends on the location or format of data saved with the Secret Manager tool. These implementation details may change. For example, the secret values aren't encrypted, but could be in the future.

Enable secret storage

The Secret Manager tool operates on project-specific configuration settings stored in your user profile.

To use user secrets, define a `UserSecretsId` element within a `PropertyGroup` of the *.csproj* file. The inner text of `UserSecretsId` is arbitrary, but is unique to the project. Developers typically generate a GUID for the `UserSecretsId`.

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

> **TIP**
> In Visual Studio, right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a `UserSecretsId` element, populated with a GUID, to the *.csproj* file.

Set a secret

Define an app secret consisting of a key and its value. The secret is associated with the project's `UserSecretsId` value. For example, run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

In the preceding example, the colon denotes that `Movies` is an object literal with a `ServiceApiKey` property.

The Secret Manager tool can be used from other directories too. Use the `--project` option to supply the file system path at which the *.csproj* file exists. For example:

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345" --project "C:\apps\WebApp1\src\WebApp1"
```

JSON structure flattening in Visual Studio

Visual Studio's **Manage User Secrets** gesture opens a *secrets.json* file in the text editor. Replace the contents of *secrets.json* with the key-value pairs to be stored. For example:

```
{
  "Movies": {
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
    "ServiceApiKey": "12345"
  }
}
```

The JSON structure is flattened after modifications via `dotnet user-secrets remove` or `dotnet user-secrets set`. For example, running `dotnet user-secrets remove "Movies:ConnectionString"` collapses the `Movies` object literal. The modified file resembles the following:

```
{
  "Movies:ServiceApiKey": "12345"
}
```

Set multiple secrets

A batch of secrets can be set by piping JSON to the `set` command. In the following example, the *input.json* file's contents are piped to the `set` command.

- Windows
- Linux / macOS

Open a command shell, and execute the following command:

```
type .\input.json | dotnet user-secrets set
```

Access a secret

The ASP.NET Core Configuration API provides access to Secret Manager secrets.

If your project targets .NET Framework, install the Microsoft.Extensions.Configuration.UserSecrets NuGet package.

In ASP.NET Core 2.0 or later, the user secrets configuration source is automatically added in development mode when the project calls CreateDefaultBuilder to initialize a new instance of the host with preconfigured defaults. `CreateDefaultBuilder` calls AddUserSecrets when the EnvironmentName is Development:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

When `CreateDefaultBuilder` isn't called, add the user secrets configuration source explicitly by calling AddUserSecrets in the `Startup` constructor. Call `AddUserSecrets` only when the app runs in the Development environment, as shown in the following example:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json",
                     optional: false,
                     reloadOnChange: true)
        .AddEnvironmentVariables();

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    Configuration = builder.Build();
}
```

User secrets can be retrieved via the `Configuration` API:

```
public class Startup
{
    private string _moviesApiKey = null;

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        _moviesApiKey = Configuration["Movies:ServiceApiKey"];
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            var result = string.IsNullOrEmpty(_moviesApiKey) ? "Null" : "Not Null";
            await context.Response.WriteAsync($"Secret is {result}");
        });
    }
}
```

Map secrets to a POCO

Mapping an entire object literal to a POCO (a simple .NET class with properties) is useful for aggregating related properties.

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

To map the preceding secrets to a POCO, use the `Configuration` API's object graph binding feature. The following code binds to a custom `MovieSettings` POCO and accesses the `ServiceApiKey` property value:

```
var moviesConfig = Configuration.GetSection("Movies")
                                .Get<MovieSettings>();
_moviesApiKey = moviesConfig.ServiceApiKey;
```

The `Movies:ConnectionString` and `Movies:ServiceApiKey` secrets are mapped to the respective properties in `MovieSettings`:

```
public class MovieSettings
{
    public string ConnectionString { get; set; }

    public string ServiceApiKey { get; set; }
}
```

String replacement with secrets

Storing passwords in plain text is insecure. For example, a database connection string stored in *appsettings.json* may include a password for the specified user:

```
{
  "ConnectionStrings": {
    "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User
Id=johndoe;Password=pass123;MultipleActiveResultSets=true"
  }
}
```

A more secure approach is to store the password as a secret. For example:

```
dotnet user-secrets set "DbPassword" "pass123"
```

Remove the `Password` key-value pair from the connection string in *appsettings.json*. For example:

```
{
  "ConnectionStrings": {
    "Movies": "Server=(localdb)\\mssqllocaldb;Database=Movie-1;User
Id=johndoe;MultipleActiveResultSets=true"
  }
}
```

The secret's value can be set on a SqlConnectionStringBuilder object's Password property to complete the connection string:

```
public class Startup
{
    private string _connection = null;

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        var builder = new SqlConnectionStringBuilder(
            Configuration.GetConnectionString("Movies"));
        builder.Password = Configuration["DbPassword"];
        _connection = builder.ConnectionString;
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync($"DB Connection: {_connection}");
        });
    }
}
```

List the secrets

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets list
```

The following output appears:

```
Movies:ConnectionString = Server=(localdb)\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true
Movies:ServiceApiKey = 12345
```

In the preceding example, a colon in the key names denotes the object hierarchy within *secrets.json*.

Remove a single secret

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets remove "Movies:ConnectionString"
```

The app's *secrets.json* file was modified to remove the key-value pair associated with the `MoviesConnectionString` key:

```
{
  "Movies": {
    "ServiceApiKey": "12345"
  }
}
```

Running `dotnet user-secrets list` displays the following message:

```
Movies:ServiceApiKey = 12345
```

Remove all secrets

Assume the app's *secrets.json* file contains the following two secrets:

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the *.csproj* file exists:

```
dotnet user-secrets clear
```

All user secrets for the app have been deleted from the *secrets.json* file:

```
{}
```

Running `dotnet user-secrets list` displays the following message:

```
No secrets configured for this application.
```

Additional resources

- See this issue for information on accessing Secret Manager from IIS.
- Configuration in ASP.NET Core
- Azure Key Vault Configuration Provider in ASP.NET Core

Azure Key Vault Configuration Provider in ASP.NET Core

9/22/2020 • 29 minutes to read • Edit Online

By Andrew Stanton-Nurse

This document explains how to use the Microsoft Azure Key Vault Configuration Provider to load app configuration values from Azure Key Vault secrets. Azure Key Vault is a cloud-based service that assists in safeguarding cryptographic keys and secrets used by apps and services. Common scenarios for using Azure Key Vault with ASP.NET Core apps include:

- Controlling access to sensitive configuration data.
- Meeting the requirement for FIPS 140-2 Level 2 validated Hardware Security Modules (HSM's) when storing configuration data.

View or download sample code (how to download)

Packages

Add a package reference to the Microsoft.Extensions.Configuration.AzureKeyVault package.

Sample app

The sample app runs in either of two modes determined by the `#define` statement at the top of the *Program.cs* file:

- `Certificate`: Demonstrates the use of an Azure Key Vault Client ID and X.509 certificate to access secrets stored in Azure Key Vault. This version of the sample can be run from any location, deployed to Azure App Service or any host capable of serving an ASP.NET Core app.
- `Managed`: Demonstrates how to use Managed identities for Azure resources to authenticate the app to Azure Key Vault with Azure AD authentication without credentials stored in the app's code or configuration. When using managed identities to authenticate, an Azure AD Application ID and Password (Client Secret) aren't required. The `Managed` version of the sample must be deployed to Azure. Follow the guidance in the Use the Managed identities for Azure resources section.

For more information on how to configure a sample app using preprocessor directives (`#define`), see Introduction to ASP.NET Core.

Secret storage in the Development environment

Set secrets locally using the Secret Manager tool. When the sample app runs on the local machine in the Development environment, secrets are loaded from the local Secret Manager store.

The Secret Manager tool requires a `<UserSecretsId>` property in the app's project file. Set the property value (`{GUID}`) to any unique GUID:

```
<PropertyGroup>
  <UserSecretsId>{GUID}</UserSecretsId>
</PropertyGroup>
```

Secrets are created as name-value pairs. Hierarchical values (configuration sections) use a `:` (colon) as a separator in ASP.NET Core configuration key names.

The Secret Manager is used from a command shell opened to the project's content root, where `{SECRET NAME}` is the name and `{SECRET VALUE}` is the value:

```
dotnet user-secrets set "{SECRET NAME}" "{SECRET VALUE}"
```

Execute the following commands in a command shell from the project's content root to set the secrets for the sample app:

```
dotnet user-secrets set "SecretName" "secret_value_1_dev"
dotnet user-secrets set "Section:SecretName" "secret_value_2_dev"
```

When these secrets are stored in Azure Key Vault in the Secret storage in the Production environment with Azure Key Vault section, the `_dev` suffix is changed to `_prod`. The suffix provides a visual cue in the app's output indicating the source of the configuration values.

Secret storage in the Production environment with Azure Key Vault

The instructions provided by the Quickstart: Set and retrieve a secret from Azure Key Vault using Azure CLI topic are summarized here for creating an Azure Key Vault and storing secrets used by the sample app. Refer to the topic for further details.

1. Open Azure Cloud shell using any one of the following methods in the Azure portal:

   - Select **Try It** in the upper-right corner of a code block. Use the search string "Azure CLI" in the text box.
   - Open Cloud Shell in your browser with the **Launch Cloud Shell** button.
   - Select the **Cloud Shell** button on the menu in the upper-right corner of the Azure portal.

   For more information, see Azure CLI and Overview of Azure Cloud Shell.

2. If you aren't already authenticated, sign in with the `az login` command.

3. Create a resource group with the following command, where `{RESOURCE GROUP NAME}` is the resource group name for the new resource group and `{LOCATION}` is the Azure region (datacenter):

   ```
   az group create --name "{RESOURCE GROUP NAME}" --location {LOCATION}
   ```

4. Create a key vault in the resource group with the following command, where `{KEY VAULT NAME}` is the name for the new key vault and `{LOCATION}` is the Azure region (datacenter):

   ```
   az keyvault create --name {KEY VAULT NAME} --resource-group "{RESOURCE GROUP NAME}" --location
   {LOCATION}
   ```

5. Create secrets in the key vault as name-value pairs.

   Azure Key Vault secret names are limited to alphanumeric characters and dashes. Hierarchical values (configuration sections) use `--` (two dashes) as a separator. Colons, which are normally used to delimit a section from a subkey in ASP.NET Core configuration, aren't allowed in key vault secret names. Therefore, two dashes are used and swapped for a colon when the secrets are loaded into the app's configuration.

The following secrets are for use with the sample app. The values include a `_prod` suffix to distinguish them from the `_dev` suffix values loaded in the Development environment from User Secrets. Replace `{KEY VAULT NAME}` with the name of the key vault that you created in the prior step:

```
az keyvault secret set --vault-name {KEY VAULT NAME} --name "SecretName" --value
"secret_value_1_prod"
az keyvault secret set --vault-name {KEY VAULT NAME} --name "Section--SecretName" --value
"secret_value_2_prod"
```

Use Application ID and X.509 certificate for non-Azure-hosted apps

Configure Azure AD, Azure Key Vault, and the app to use an Azure Active Directory Application ID and X.509 certificate to authenticate to a key vault **when the app is hosted outside of Azure**. For more information, see About keys, secrets, and certificates.

> **NOTE**
>
> Although using an Application ID and X.509 certificate is supported for apps hosted in Azure, we recommend using Managed identities for Azure resources when hosting an app in Azure. Managed identities don't require storing a certificate in the app or in the development environment.

The sample app uses an Application ID and X.509 certificate when the `#define` statement at the top of the *Program.cs* file is set to `Certificate`.

1. Create a PKCS#12 archive (*.pfx*) certificate. Options for creating certificates include MakeCert on Windows and OpenSSL.
2. Install the certificate into the current user's personal certificate store. Marking the key as exportable is optional. Note the certificate's thumbprint, which is used later in this process.
3. Export the PKCS#12 archive (*.pfx*) certificate as a DER-encoded certificate (*.cer*).
4. Register the app with Azure AD (**App registrations**).
5. Upload the DER-encoded certificate (*.cer*) to Azure AD:
   a. Select the app in Azure AD.
   b. Navigate to **Certificates & secrets**.
   c. Select **Upload certificate** to upload the certificate, which contains the public key. A *.cer*, *.pem*, or *.crt* certificate is acceptable.
6. Store the key vault name, Application ID, and certificate thumbprint in the app's *appsettings.json* file.
7. Navigate to **Key vaults** in the Azure portal.
8. Select the key vault that you created in the Secret storage in the Production environment with Azure Key Vault section.
9. Select **Access policies**.
10. Select **Add Access Policy**.
11. Open **Secret permissions** and provide the app with **Get** and **List** permissions.
12. Select **Select principal** and select the registered app by name. Select the **Select** button.
13. Select **OK**.
14. Select **Save**.
15. Deploy the app.

The `Certificate` sample app obtains its configuration values from `IConfigurationRoot` with the same name as the secret name:

- Non-hierarchical values: The value for `SecretName` is obtained with `config["SecretName"]`.

- Hierarchical values (sections): Use `:` (colon) notation or the `GetSection` extension method. Use either of these approaches to obtain the configuration value:
  - `config["Section:SecretName"]`
  - `config.GetSection("Section")["SecretName"]`

The X.509 certificate is managed by the OS. The app calls AddAzureKeyVault with values supplied by the *appsettings.json* file:

```
// using System.Linq;
// using System.Security.Cryptography.X509Certificates;
// using Microsoft.Extensions.Configuration;

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            if (context.HostingEnvironment.IsProduction())
            {
                var builtConfig = config.Build();

                using (var store = new X509Store(StoreLocation.CurrentUser))
                {
                    store.Open(OpenFlags.ReadOnly);
                    var certs = store.Certificates
                        .Find(X509FindType.FindByThumbprint,
                            builtConfig["AzureADCertThumbprint"], false);

                    config.AddAzureKeyVault(
                        $"https://{builtConfig["KeyVaultName"]}.vault.azure.net/",
                        builtConfig["AzureADApplicationId"],
                        certs.OfType<X509Certificate2>().Single());

                    store.Close();
                }
            }
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

Example values:

- Key vault name: `contosovault`
- Application ID: `627e911e-43cc-61d4-992e-12db9c81b413`
- Certificate thumbprint: `fe14593dd66b2406c5269d742d04b6e1ab03adb1`

appsettings.json:

```
{
  "KeyVaultName": "Key Vault Name",
  "AzureADApplicationId": "Azure AD Application ID",
  "AzureADCertThumbprint": "Azure AD Certificate Thumbprint"
}
```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values load with the `_dev` suffix. In the Production environment, the values load with the `_prod` suffix.

Use Managed identities for Azure resources

An app deployed to Azure can take advantage of Managed identities for Azure resources, which allows the app to authenticate with Azure Key Vault using Azure AD authentication without credentials (Application ID and Password/Client Secret) stored in the app.

The sample app uses Managed identities for Azure resources when the `#define` statement at the top of the *Program.cs* file is set to `Managed`.

Enter the vault name into the app's *appsettings.json* file. The sample app doesn't require an Application ID and Password (Client Secret) when set to the `Managed` version, so you can ignore those configuration entries. The app is deployed to Azure, and Azure authenticates the app to access Azure Key Vault only using the vault name stored in the *appsettings.json* file.

Deploy the sample app to Azure App Service.

An app deployed to Azure App Service is automatically registered with Azure AD when the service is created. Obtain the Object ID from the deployment for use in the following command. The Object ID is shown in the Azure portal on the **Identity** panel of the App Service.

Using Azure CLI and the app's Object ID, provide the app with `list` and `get` permissions to access the key vault:

```
az keyvault set-policy --name {KEY VAULT NAME} --object-id {OBJECT ID} --secret-permissions get list
```

Restart the app using Azure CLI, PowerShell, or the Azure portal.

The sample app:

- Creates an instance of the `AzureServiceTokenProvider` class without a connection string. When a connection string isn't provided, the provider attempts to obtain an access token from Managed identities for Azure resources.
- A new KeyVaultClient is created with the `AzureServiceTokenProvider` instance token callback.
- The KeyVaultClient instance is used with a default implementation of IKeyVaultSecretManager that loads all secret values and replaces double-dashes (`--`) with colons (`:`) in key names.

```
// using Microsoft.Azure.KeyVault;
// using Microsoft.Azure.Services.AppAuthentication;
// using Microsoft.Extensions.Configuration.AzureKeyVault;

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            if (context.HostingEnvironment.IsProduction())
            {
                var builtConfig = config.Build();

                var azureServiceTokenProvider = new AzureServiceTokenProvider();
                var keyVaultClient = new KeyVaultClient(
                    new KeyVaultClient.AuthenticationCallback(
                        azureServiceTokenProvider.KeyVaultTokenCallback));

                config.AddAzureKeyVault(
                    $"https://{builtConfig["KeyVaultName"]}.vault.azure.net/",
                    keyVaultClient,
                    new DefaultKeyVaultSecretManager());
            }
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

Key vault name example value: `contosovault`

appsettings.json:

```
{
  "KeyVaultName": "Key Vault Name"
}
```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values have the `_dev` suffix because they're provided by User Secrets. In the Production environment, the values load with the `_prod` suffix because they're provided by Azure Key Vault.

If you receive an `Access denied` error, confirm that the app is registered with Azure AD and provided access to the key vault. Confirm that you've restarted the service in Azure.

For information on using the provider with a managed identity and an Azure DevOps pipeline, see Create an Azure Resource Manager service connection to a VM with a managed service identity.

Configuration options

AddAzureKeyVault can accept an AzureKeyVaultConfigurationOptions:

```
config.AddAzureKeyVault(
    new AzureKeyVaultConfigurationOptions()
    {
        ...
    });
```

PROPERTY	DESCRIPTION
`Client`	KeyVaultClient to use for retrieving values.
`Manager`	IKeyVaultSecretManager instance used to control secret loading.
`ReloadInterval`	`Timespan` to wait between attempts at polling the key vault for changes. The default value is `null` (configuration isn't reloaded).
`Vault`	Key vault URI.

Use a key name prefix

AddAzureKeyVault provides an overload that accepts an implementation of IKeyVaultSecretManager, which allows you to control how key vault secrets are converted into configuration keys. For example, you can implement the interface to load secret values based on a prefix value you provide at app startup. This allows you, for example, to load secrets based on the version of the app.

> **WARNING**
> Don't use prefixes on key vault secrets to place secrets for multiple apps into the same key vault or to place environmental secrets (for example, *development* versus *production* secrets) into the same vault. We recommend that different apps and development/production environments use separate key vaults to isolate app environments for the highest level of security.

In the following example, a secret is established in the key vault (and using the Secret Manager tool for the Development environment) for `5000-AppSecret` (periods aren't allowed in key vault secret names). This secret represents an app secret for version 5.0.0.0 of the app. For another version of the app, 5.1.0.0, a secret is added to the key vault (and using the Secret Manager tool) for `5100-AppSecret`. Each app version loads its versioned secret value into its configuration as `AppSecret`, stripping off the version as it loads the secret.

AddAzureKeyVault is called with a custom IKeyVaultSecretManager:

```
config.AddAzureKeyVault(
    $"https://{builtConfig["KeyVaultName"]}.vault.azure.net/",
    builtConfig["AzureADApplicationId"],
    certs.OfType<X509Certificate2>().Single(),
    new PrefixKeyVaultSecretManager(versionPrefix));
```

The IKeyVaultSecretManager implementation reacts to the version prefixes of secrets to load the proper secret into configuration:

- `Load` loads a secret when its name starts with the prefix. Other secrets aren't loaded.
- `GetKey`:
  - Removes the prefix from the secret name.
  - Replaces two dashes in any name with the `KeyDelimiter`, which is the delimiter used in configuration (usually a colon). Azure Key Vault doesn't allow a colon in secret names.

```
public class PrefixKeyVaultSecretManager : IKeyVaultSecretManager
{
    private readonly string _prefix;

    public PrefixKeyVaultSecretManager(string prefix)
    {
        _prefix = $"{prefix}-";
    }

    public bool Load(SecretItem secret)
    {
        return secret.Identifier.Name.StartsWith(_prefix);
    }

    public string GetKey(SecretBundle secret)
    {
        return secret.SecretIdentifier.Name
            .Substring(_prefix.Length)
            .Replace("--", ConfigurationPath.KeyDelimiter);
    }
}
```

The `Load` method is called by a provider algorithm that iterates through the vault secrets to find the ones that have the version prefix. When a version prefix is found with `Load`, the algorithm uses the `GetKey` method to return the configuration name of the secret name. It strips off the version prefix from the secret's name and returns the rest of the secret name for loading into the app's configuration name-value pairs.

When this approach is implemented:

1. The app's version specified in the app's project file. In the following example, the app's version is set to `5.0.0.0`:

   ```
   <PropertyGroup>
     <Version>5.0.0.0</Version>
   </PropertyGroup>
   ```

2. Confirm that a `<UserSecretsId>` property is present in the app's project file, where `{GUID}` is a user-supplied GUID:

   ```
   <PropertyGroup>
     <UserSecretsId>{GUID}</UserSecretsId>
   </PropertyGroup>
   ```

   Save the following secrets locally with the Secret Manager tool:

   ```
   dotnet user-secrets set "5000-AppSecret" "5.0.0.0_secret_value_dev"
   dotnet user-secrets set "5100-AppSecret" "5.1.0.0_secret_value_dev"
   ```

3. Secrets are saved in Azure Key Vault using the following Azure CLI commands:

   ```
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5000-AppSecret" --value
   "5.0.0.0_secret_value_prod"
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5100-AppSecret" --value
   "5.1.0.0_secret_value_prod"
   ```

4. When the app is run, the key vault secrets are loaded. The string secret for `5000-AppSecret` is matched to the app's version specified in the app's project file (`5.0.0.0`).

5. The version, `5000` (with the dash), is stripped from the key name. Throughout the app, reading configuration with the key `AppSecret` loads the secret value.

6. If the app's version is changed in the project file to `5.1.0.0` and the app is run again, the secret value returned is `5.1.0.0_secret_value_dev` in the Development environment and `5.1.0.0_secret_value_prod` in Production.

> **NOTE**
>
> You can also provide your own KeyVaultClient implementation to AddAzureKeyVault. A custom client permits sharing a single instance of the client across the app.

Bind an array to a class

The provider is capable of reading configuration values into an array for binding to a POCO array.

When reading from a configuration source that allows keys to contain colon (`:`) separators, a numeric key segment is used to distinguish the keys that make up an array (`:0:` , `:1:` , ... `:{n}:`). For more information, see Configuration: Bind an array to a class.

Azure Key Vault keys can't use a colon as a separator. The approach described in this topic uses double dashes (`--`) as a separator for hierarchical values (sections). Array keys are stored in Azure Key Vault with double dashes and numeric key segments (`--0--` , `--1--` , ... `--{n}--`).

Examine the following Serilog logging provider configuration provided by a JSON file. There are two object literals defined in the `WriteTo` array that reflect two Serilog *sinks*, which describe destinations for logging output:

```
"Serilog": {
  "WriteTo": [
    {
      "Name": "AzureTableStorage",
      "Args": {
        "storageTableName": "logs",
        "connectionString": "DefaultEnd...ountKey=Eby8...GMGw=="
      }
    },
    {
      "Name": "AzureDocumentDB",
      "Args": {
        "endpointUrl": "https://contoso.documents.azure.com:443",
        "authorizationKey": "Eby8...GMGw=="
      }
    }
  ]
}
```

The configuration shown in the preceding JSON file is stored in Azure Key Vault using double dash (`--`) notation and numeric segments:

KEY	VALUE
`Serilog--WriteTo--0--Name`	`AzureTableStorage`
`Serilog--WriteTo--0--Args--storageTableName`	`logs`

KEY	VALUE
`Serilog--WriteTo--0--Args--connectionString`	`DefaultEnd...ountKey=Eby8...GMGw==`
`Serilog--WriteTo--1--Name`	`AzureDocumentDB`
`Serilog--WriteTo--1--Args--endpointUrl`	`https://contoso.documents.azure.com:443`
`Serilog--WriteTo--1--Args--authorizationKey`	`Eby8...GMGw==`

Reload secrets

Secrets are cached until `IConfigurationRoot.Reload()` is called. Expired, disabled, and updated secrets in the key vault are not respected by the app until `Reload` is executed.

```
Configuration.Reload();
```

Disabled and expired secrets

Disabled and expired secrets throw a KeyVaultErrorException. To prevent the app from throwing, provide the configuration using a different configuration provider or update the disabled or expired secret.

Troubleshoot

When the app fails to load configuration using the provider, an error message is written to the ASP.NET Core Logging infrastructure. The following conditions will prevent configuration from loading:

- The app or certificate isn't configured correctly in Azure Active Directory.
- The key vault doesn't exist in Azure Key Vault.
- The app isn't authorized to access the key vault.
- The access policy doesn't include `Get` and `List` permissions.
- In the key vault, the configuration data (name-value pair) is incorrectly named, missing, disabled, or expired.
- The app has the wrong key vault name (`KeyVaultName`), Azure AD Application Id (`AzureADApplicationId`), or Azure AD certificate thumbprint (`AzureADCertThumbprint`).
- The configuration key (name) is incorrect in the app for the value you're trying to load.
- When adding the access policy for the app to the key vault, the policy was created, but the **Save** button wasn't selected in the **Access policies** UI.

Additional resources

- Configuration in ASP.NET Core
- Microsoft Azure: Key Vault
- Microsoft Azure: Key Vault Documentation
- How to generate and transfer HSM-protected keys for Azure Key Vault
- KeyVaultClient Class
- Quickstart: Set and retrieve a secret from Azure Key Vault by using a .NET web app
- Tutorial: How to use Azure Key Vault with Azure Windows Virtual Machine in .NET

This document explains how to use the Microsoft Azure Key Vault Configuration Provider to load app configuration values from Azure Key Vault secrets. Azure Key Vault is a cloud-based service that assists in

safeguarding cryptographic keys and secrets used by apps and services. Common scenarios for using Azure Key Vault with ASP.NET Core apps include:

- Controlling access to sensitive configuration data.
- Meeting the requirement for FIPS 140-2 Level 2 validated Hardware Security Modules (HSM's) when storing configuration data.

View or download sample code (how to download)

Packages

Add a package reference to the Microsoft.Extensions.Configuration.AzureKeyVault package.

Sample app

The sample app runs in either of two modes determined by the `#define` statement at the top of the *Program.cs* file:

- `Certificate`: Demonstrates the use of an Azure Key Vault Client ID and X.509 certificate to access secrets stored in Azure Key Vault. This version of the sample can be run from any location, deployed to Azure App Service or any host capable of serving an ASP.NET Core app.
- `Managed`: Demonstrates how to use Managed identities for Azure resources to authenticate the app to Azure Key Vault with Azure AD authentication without credentials stored in the app's code or configuration. When using managed identities to authenticate, an Azure AD Application ID and Password (Client Secret) aren't required. The `Managed` version of the sample must be deployed to Azure. Follow the guidance in the Use the Managed identities for Azure resources section.

For more information on how to configure a sample app using preprocessor directives (`#define`), see Introduction to ASP.NET Core.

Secret storage in the Development environment

Set secrets locally using the Secret Manager tool. When the sample app runs on the local machine in the Development environment, secrets are loaded from the local Secret Manager store.

The Secret Manager tool requires a `<UserSecretsId>` property in the app's project file. Set the property value (`{GUID}`) to any unique GUID:

```
<PropertyGroup>
  <UserSecretsId>{GUID}</UserSecretsId>
</PropertyGroup>
```

Secrets are created as name-value pairs. Hierarchical values (configuration sections) use a `:` (colon) as a separator in ASP.NET Core configuration key names.

The Secret Manager is used from a command shell opened to the project's content root, where `{SECRET NAME}` is the name and `{SECRET VALUE}` is the value:

```
dotnet user-secrets set "{SECRET NAME}" "{SECRET VALUE}"
```

Execute the following commands in a command shell from the project's content root to set the secrets for the sample app:

```
dotnet user-secrets set "SecretName" "secret_value_1_dev"
dotnet user-secrets set "Section:SecretName" "secret_value_2_dev"
```

When these secrets are stored in Azure Key Vault in the Secret storage in the Production environment with Azure Key Vault section, the `_dev` suffix is changed to `_prod`. The suffix provides a visual cue in the app's output indicating the source of the configuration values.

Secret storage in the Production environment with Azure Key Vault

The instructions provided by the Quickstart: Set and retrieve a secret from Azure Key Vault using Azure CLI topic are summarized here for creating an Azure Key Vault and storing secrets used by the sample app. Refer to the topic for further details.

1. Open Azure Cloud shell using any one of the following methods in the Azure portal:

   - Select **Try It** in the upper-right corner of a code block. Use the search string "Azure CLI" in the text box.
   - Open Cloud Shell in your browser with the **Launch Cloud Shell** button.
   - Select the **Cloud Shell** button on the menu in the upper-right corner of the Azure portal.

   For more information, see Azure CLI and Overview of Azure Cloud Shell.

2. If you aren't already authenticated, sign in with the `az login` command.

3. Create a resource group with the following command, where `{RESOURCE GROUP NAME}` is the resource group name for the new resource group and `{LOCATION}` is the Azure region (datacenter):

   ```
   az group create --name "{RESOURCE GROUP NAME}" --location {LOCATION}
   ```

4. Create a key vault in the resource group with the following command, where `{KEY VAULT NAME}` is the name for the new key vault and `{LOCATION}` is the Azure region (datacenter):

   ```
   az keyvault create --name {KEY VAULT NAME} --resource-group "{RESOURCE GROUP NAME}" --location
   {LOCATION}
   ```

5. Create secrets in the key vault as name-value pairs.

   Azure Key Vault secret names are limited to alphanumeric characters and dashes. Hierarchical values (configuration sections) use `--` (two dashes) as a separator. Colons, which are normally used to delimit a section from a subkey in ASP.NET Core configuration, aren't allowed in key vault secret names. Therefore, two dashes are used and swapped for a colon when the secrets are loaded into the app's configuration.

   The following secrets are for use with the sample app. The values include a `_prod` suffix to distinguish them from the `_dev` suffix values loaded in the Development environment from User Secrets. Replace `{KEY VAULT NAME}` with the name of the key vault that you created in the prior step:

   ```
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "SecretName" --value
   "secret_value_1_prod"
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "Section--SecretName" --value
   "secret_value_2_prod"
   ```

Use Application ID and X.509 certificate for non-Azure-hosted apps

Configure Azure AD, Azure Key Vault, and the app to use an Azure Active Directory Application ID and X.509 certificate to authenticate to a key vault **when the app is hosted outside of Azure**. For more information, see About keys, secrets, and certificates.

> **NOTE**
>
> Although using an Application ID and X.509 certificate is supported for apps hosted in Azure, we recommend using Managed identities for Azure resources when hosting an app in Azure. Managed identities don't require storing a certificate in the app or in the development environment.

The sample app uses an Application ID and X.509 certificate when the `#define` statement at the top of the *Program.cs* file is set to `Certificate`.

1. Create a PKCS#12 archive (*.pfx*) certificate. Options for creating certificates include MakeCert on Windows and OpenSSL.
2. Install the certificate into the current user's personal certificate store. Marking the key as exportable is optional. Note the certificate's thumbprint, which is used later in this process.
3. Export the PKCS#12 archive (*.pfx*) certificate as a DER-encoded certificate (*.cer*).
4. Register the app with Azure AD (**App registrations**).
5. Upload the DER-encoded certificate (*.cer*) to Azure AD:
   a. Select the app in Azure AD.
   b. Navigate to **Certificates & secrets**.
   c. Select **Upload certificate** to upload the certificate, which contains the public key. A *.cer*, *.pem*, or *.crt* certificate is acceptable.
6. Store the key vault name, Application ID, and certificate thumbprint in the app's *appsettings.json* file.
7. Navigate to **Key vaults** in the Azure portal.
8. Select the key vault that you created in the Secret storage in the Production environment with Azure Key Vault section.
9. Select **Access policies**.
10. Select **Add Access Policy**.
11. Open **Secret permissions** and provide the app with **Get** and **List** permissions.
12. Select **Select principal** and select the registered app by name. Select the **Select** button.
13. Select **OK**.
14. Select **Save**.
15. Deploy the app.

The `Certificate` sample app obtains its configuration values from `IConfigurationRoot` with the same name as the secret name:

- Non-hierarchical values: The value for `SecretName` is obtained with `config["SecretName"]`.
- Hierarchical values (sections): Use `:` (colon) notation or the `GetSection` extension method. Use either of these approaches to obtain the configuration value:
  - `config["Section:SecretName"]`
  - `config.GetSection("Section")["SecretName"]`

The X.509 certificate is managed by the OS. The app calls AddAzureKeyVault with values supplied by the *appsettings.json* file:

```
// using System.Linq;
// using System.Security.Cryptography.X509Certificates;
// using Microsoft.Extensions.Configuration;

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            if (context.HostingEnvironment.IsProduction())
            {
                var builtConfig = config.Build();

                using (var store = new X509Store(StoreLocation.CurrentUser))
                {
                    store.Open(OpenFlags.ReadOnly);
                    var certs = store.Certificates
                        .Find(X509FindType.FindByThumbprint,
                            builtConfig["AzureADCertThumbprint"], false);

                    config.AddAzureKeyVault(
                        $"https://{builtConfig["KeyVaultName"]}.vault.azure.net/",
                        builtConfig["AzureADApplicationId"],
                        certs.OfType<X509Certificate2>().Single());

                    store.Close();
                }
            }
        })
        .UseStartup<Startup>();
```

Example values:

- Key vault name: `contosovault`
- Application ID: `627e911e-43cc-61d4-992e-12db9c81b413`
- Certificate thumbprint: `fe14593dd66b2406c5269d742d04b6e1ab03adb1`

appsettings.json:

```
{
  "KeyVaultName": "Key Vault Name",
  "AzureADApplicationId": "Azure AD Application ID",
  "AzureADCertThumbprint": "Azure AD Certificate Thumbprint"
}
```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values load with the `_dev` suffix. In the Production environment, the values load with the `_prod` suffix.

Use Managed identities for Azure resources

An app deployed to Azure can take advantage of Managed identities for Azure resources, which allows the app to authenticate with Azure Key Vault using Azure AD authentication without credentials (Application ID and Password/Client Secret) stored in the app.

The sample app uses Managed identities for Azure resources when the `#define` statement at the top of the *Program.cs* file is set to `Managed`.

Enter the vault name into the app's *appsettings.json* file. The sample app doesn't require an Application ID and Password (Client Secret) when set to the `Managed` version, so you can ignore those configuration entries. The app is deployed to Azure, and Azure authenticates the app to access Azure Key Vault only using the vault name stored in the *appsettings.json* file.

Deploy the sample app to Azure App Service.

An app deployed to Azure App Service is automatically registered with Azure AD when the service is created. Obtain the Object ID from the deployment for use in the following command. The Object ID is shown in the Azure portal on the **Identity** panel of the App Service.

Using Azure CLI and the app's Object ID, provide the app with `list` and `get` permissions to access the key vault:

```
az keyvault set-policy --name {KEY VAULT NAME} --object-id {OBJECT ID} --secret-permissions get list
```

Restart the app using Azure CLI, PowerShell, or the Azure portal.

The sample app:

- Creates an instance of the `AzureServiceTokenProvider` class without a connection string. When a connection string isn't provided, the provider attempts to obtain an access token from Managed identities for Azure resources.
- A new KeyVaultClient is created with the `AzureServiceTokenProvider` instance token callback.
- The KeyVaultClient instance is used with a default implementation of IKeyVaultSecretManager that loads all secret values and replaces double-dashes (`--`) with colons (`:`) in key names.

```
// using Microsoft.Azure.KeyVault;
// using Microsoft.Azure.Services.AppAuthentication;
// using Microsoft.Extensions.Configuration.AzureKeyVault;

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
        {
            if (context.HostingEnvironment.IsProduction())
            {
                var builtConfig = config.Build();

                var azureServiceTokenProvider = new AzureServiceTokenProvider();
                var keyVaultClient = new KeyVaultClient(
                    new KeyVaultClient.AuthenticationCallback(
                        azureServiceTokenProvider.KeyVaultTokenCallback));

                config.AddAzureKeyVault(
                    $"https://{builtConfig["KeyVaultName"]}.vault.azure.net/",
                    keyVaultClient,
                    new DefaultKeyVaultSecretManager());
            }
        })
        .UseStartup<Startup>();
```

Key vault name example value: `contosovault`

appsettings.json:

```
{
  "KeyVaultName": "Key Vault Name"
}
```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values have the `_dev` suffix because they're provided by User Secrets. In the Production environment, the values load with the `_prod` suffix because they're provided by Azure Key Vault.

If you receive an `Access denied` error, confirm that the app is registered with Azure AD and provided access to the key vault. Confirm that you've restarted the service in Azure.

For information on using the provider with a managed identity and an Azure DevOps pipeline, see Create an Azure Resource Manager service connection to a VM with a managed service identity.

Use a key name prefix

AddAzureKeyVault provides an overload that accepts an implementation of IKeyVaultSecretManager, which allows you to control how key vault secrets are converted into configuration keys. For example, you can implement the interface to load secret values based on a prefix value you provide at app startup. This allows you, for example, to load secrets based on the version of the app.

> **WARNING**
> Don't use prefixes on key vault secrets to place secrets for multiple apps into the same key vault or to place environmental secrets (for example, *development* versus *production* secrets) into the same vault. We recommend that different apps and development/production environments use separate key vaults to isolate app environments for the highest level of security.

In the following example, a secret is established in the key vault (and using the Secret Manager tool for the Development environment) for `5000-AppSecret` (periods aren't allowed in key vault secret names). This secret represents an app secret for version 5.0.0.0 of the app. For another version of the app, 5.1.0.0, a secret is added to the key vault (and using the Secret Manager tool) for `5100-AppSecret`. Each app version loads its versioned secret value into its configuration as `AppSecret`, stripping off the version as it loads the secret.

AddAzureKeyVault is called with a custom IKeyVaultSecretManager:

```
config.AddAzureKeyVault(
    $"https://{builtConfig["KeyVaultName"]}.vault.azure.net/",
    builtConfig["AzureADApplicationId"],
    certs.OfType<X509Certificate2>().Single(),
    new PrefixKeyVaultSecretManager(versionPrefix));
```

The IKeyVaultSecretManager implementation reacts to the version prefixes of secrets to load the proper secret into configuration:

- `Load` loads a secret when its name starts with the prefix. Other secrets aren't loaded.
- `GetKey`:
  - Removes the prefix from the secret name.
  - Replaces two dashes in any name with the `KeyDelimiter`, which is the delimiter used in configuration (usually a colon). Azure Key Vault doesn't allow a colon in secret names.

```
public class PrefixKeyVaultSecretManager : IKeyVaultSecretManager
{
    private readonly string _prefix;

    public PrefixKeyVaultSecretManager(string prefix)
    {
        _prefix = $"{prefix}-";
    }

    public bool Load(SecretItem secret)
    {
        return secret.Identifier.Name.StartsWith(_prefix);
    }

    public string GetKey(SecretBundle secret)
    {
        return secret.SecretIdentifier.Name
            .Substring(_prefix.Length)
            .Replace("--", ConfigurationPath.KeyDelimiter);
    }
}
```

The `Load` method is called by a provider algorithm that iterates through the vault secrets to find the ones that have the version prefix. When a version prefix is found with `Load`, the algorithm uses the `GetKey` method to return the configuration name of the secret name. It strips off the version prefix from the secret's name and returns the rest of the secret name for loading into the app's configuration name-value pairs.

When this approach is implemented:

1. The app's version specified in the app's project file. In the following example, the app's version is set to `5.0.0.0`:

   ```
   <PropertyGroup>
     <Version>5.0.0.0</Version>
   </PropertyGroup>
   ```

2. Confirm that a `<UserSecretsId>` property is present in the app's project file, where `{GUID}` is a user-supplied GUID:

   ```
   <PropertyGroup>
     <UserSecretsId>{GUID}</UserSecretsId>
   </PropertyGroup>
   ```

   Save the following secrets locally with the Secret Manager tool:

   ```
   dotnet user-secrets set "5000-AppSecret" "5.0.0.0_secret_value_dev"
   dotnet user-secrets set "5100-AppSecret" "5.1.0.0_secret_value_dev"
   ```

3. Secrets are saved in Azure Key Vault using the following Azure CLI commands:

   ```
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5000-AppSecret" --value
   "5.0.0.0_secret_value_prod"
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5100-AppSecret" --value
   "5.1.0.0_secret_value_prod"
   ```

4. When the app is run, the key vault secrets are loaded. The string secret for `5000-AppSecret` is matched to the app's version specified in the app's project file (`5.0.0.0`).

5. The version, `5000` (with the dash), is stripped from the key name. Throughout the app, reading configuration with the key `AppSecret` loads the secret value.

6. If the app's version is changed in the project file to `5.1.0.0` and the app is run again, the secret value returned is `5.1.0.0_secret_value_dev` in the Development environment and `5.1.0.0_secret_value_prod` in Production.

> **NOTE**
>
> You can also provide your own KeyVaultClient implementation to AddAzureKeyVault. A custom client permits sharing a single instance of the client across the app.

Bind an array to a class

The provider is capable of reading configuration values into an array for binding to a POCO array.

When reading from a configuration source that allows keys to contain colon (`:`) separators, a numeric key segment is used to distinguish the keys that make up an array (`:0:` , `:1:` , ... `:{n}:`). For more information, see Configuration: Bind an array to a class.

Azure Key Vault keys can't use a colon as a separator. The approach described in this topic uses double dashes (`--`) as a separator for hierarchical values (sections). Array keys are stored in Azure Key Vault with double dashes and numeric key segments (`--0--` , `--1--` , ... `--{n}--`).

Examine the following Serilog logging provider configuration provided by a JSON file. There are two object literals defined in the `WriteTo` array that reflect two Serilog *sinks*, which describe destinations for logging output:

```
"Serilog": {
  "WriteTo": [
    {
      "Name": "AzureTableStorage",
      "Args": {
        "storageTableName": "logs",
        "connectionString": "DefaultEnd...ountKey=Eby8...GMGw=="
      }
    },
    {
      "Name": "AzureDocumentDB",
      "Args": {
        "endpointUrl": "https://contoso.documents.azure.com:443",
        "authorizationKey": "Eby8...GMGw=="
      }
    }
  ]
}
```

The configuration shown in the preceding JSON file is stored in Azure Key Vault using double dash (`--`) notation and numeric segments:

KEY	VALUE
`Serilog--WriteTo--0--Name`	`AzureTableStorage`
`Serilog--WriteTo--0--Args--storageTableName`	`logs`

KEY	VALUE
`Serilog--WriteTo--0--Args--connectionString`	`DefaultEnd...ountKey=Eby8...GMGw==`
`Serilog--WriteTo--1--Name`	`AzureDocumentDB`
`Serilog--WriteTo--1--Args--endpointUrl`	`https://contoso.documents.azure.com:443`
`Serilog--WriteTo--1--Args--authorizationKey`	`Eby8...GMGw==`

Reload secrets

Secrets are cached until `IConfigurationRoot.Reload()` is called. Expired, disabled, and updated secrets in the key vault are not respected by the app until `Reload` is executed.

```
Configuration.Reload();
```

Disabled and expired secrets

Disabled and expired secrets throw a KeyVaultErrorException. To prevent the app from throwing, provide the configuration using a different configuration provider or update the disabled or expired secret.

Troubleshoot

When the app fails to load configuration using the provider, an error message is written to the ASP.NET Core Logging infrastructure. The following conditions will prevent configuration from loading:

- The app or certificate isn't configured correctly in Azure Active Directory.
- The key vault doesn't exist in Azure Key Vault.
- The app isn't authorized to access the key vault.
- The access policy doesn't include `Get` and `List` permissions.
- In the key vault, the configuration data (name-value pair) is incorrectly named, missing, disabled, or expired.
- The app has the wrong key vault name (`KeyVaultName`), Azure AD Application Id (`AzureADApplicationId`), or Azure AD certificate thumbprint (`AzureADCertThumbprint`).
- The configuration key (name) is incorrect in the app for the value you're trying to load.
- When adding the access policy for the app to the key vault, the policy was created, but the **Save** button wasn't selected in the **Access policies** UI.

Additional resources

- Configuration in ASP.NET Core
- Microsoft Azure: Key Vault
- Microsoft Azure: Key Vault Documentation
- How to generate and transfer HSM-protected keys for Azure Key Vault
- KeyVaultClient Class
- Quickstart: Set and retrieve a secret from Azure Key Vault by using a .NET web app
- Tutorial: How to use Azure Key Vault with Azure Windows Virtual Machine in .NET

Enforce HTTPS in ASP.NET Core

9/22/2020 • 12 minutes to read • Edit Online

By Rick Anderson

This document shows how to:

- Require HTTPS for all requests.
- Redirect all HTTP requests to HTTPS.

No API can prevent a client from sending sensitive data on the first request.

WARNING

API projects

Do **not** use RequireHttpsAttribute on Web APIs that receive sensitive information. `RequireHttpsAttribute` uses HTTP status codes to redirect browsers from HTTP to HTTPS. API clients may not understand or obey redirects from HTTP to HTTPS. Such clients may send information over HTTP. Web APIs should either:

- Not listen on HTTP.
- Close the connection with status code 400 (Bad Request) and not serve the request.

HSTS and API projects

The default API projects don't include HSTS because HSTS is generally a browser only instruction. Other callers, such as phone or desktop apps, do **not** obey the instruction. Even within browsers, a single authenticated call to an API over HTTP has risks on insecure networks. The secure approach is to configure API projects to only listen to and respond over HTTPS.

WARNING

API projects

Do **not** use RequireHttpsAttribute on Web APIs that receive sensitive information. `RequireHttpsAttribute` uses HTTP status codes to redirect browsers from HTTP to HTTPS. API clients may not understand or obey redirects from HTTP to HTTPS. Such clients may send information over HTTP. Web APIs should either:

- Not listen on HTTP.
- Close the connection with status code 400 (Bad Request) and not serve the request.

Require HTTPS

We recommend that production ASP.NET Core web apps use:

- HTTPS Redirection Middleware (UseHttpsRedirection) to redirect HTTP requests to HTTPS.
- HSTS Middleware (UseHsts) to send HTTP Strict Transport Security Protocol (HSTS) headers to clients.

> **NOTE**
>
> Apps deployed in a reverse proxy configuration allow the proxy to handle connection security (HTTPS). If the proxy also handles HTTPS redirection, there's no need to use HTTPS Redirection Middleware. If the proxy server also handles writing HSTS headers (for example, native HSTS support in IIS 10.0 (1709) or later), HSTS Middleware isn't required by the app. For more information, see Opt-out of HTTPS/HSTS on project creation.

UseHttpsRedirection

The following code calls `UseHttpsRedirection` in the `Startup` class:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseMvc();
}
```

The preceding highlighted code:

- Uses the default HttpsRedirectionOptions.RedirectStatusCode (Status307TemporaryRedirect).
- Uses the default HttpsRedirectionOptions.HttpsPort (null) unless overridden by the `ASPNETCORE_HTTPS_PORT`

environment variable or IServerAddressesFeature.

We recommend using temporary redirects rather than permanent redirects. Link caching can cause unstable behavior in development environments. If you prefer to send a permanent redirect status code when the app is in a non-Development environment, see the Configure permanent redirects in production section. We recommend using HSTS to signal to clients that only secure resource requests should be sent to the app (only in production).

Port configuration

A port must be available for the middleware to redirect an insecure request to HTTPS. If no port is available:

- Redirection to HTTPS doesn't occur.
- The middleware logs the warning "Failed to determine the https port for redirect."

Specify the HTTPS port using any of the following approaches:

- Set HttpsRedirectionOptions.HttpsPort.
- Set the `https_port` host setting:
  - In host configuration.
  - By setting the `ASPNETCORE_HTTPS_PORT` environment variable.
  - By adding a top-level entry in *appsettings.json*:

```
{
    "https_port": 443,
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "AllowedHosts": "*"
}
```

- Indicate a port with the secure scheme using the ASPNETCORE_URLS environment variable. The environment variable configures the server. The middleware indirectly discovers the HTTPS port via IServerAddressesFeature. This approach doesn't work in reverse proxy deployments.
- Set the `https_port` host setting:
  - In host configuration.
  - By setting the `ASPNETCORE_HTTPS_PORT` environment variable.
  - By adding a top-level entry in *appsettings.json*:

```
{
    "https_port": 443,
    "Logging": {
        "LogLevel": {
            "Default": "Warning"
        }
    },
    "AllowedHosts": "*"
}
```

- Indicate a port with the secure scheme using the ASPNETCORE_URLS environment variable. The environment variable configures the server. The middleware indirectly discovers the HTTPS port via IServerAddressesFeature. This approach doesn't work in reverse proxy deployments.
- In development, set an HTTPS URL in *launchsettings.json*. Enable HTTPS when IIS Express is used.
- Configure an HTTPS URL endpoint for a public-facing edge deployment of Kestrel server or HTTP.sys server. Only **one HTTPS port** is used by the app. The middleware discovers the port via IServerAddressesFeature.

> **NOTE**
> When an app is run in a reverse proxy configuration, IServerAddressesFeature isn't available. Set the port using one of the other approaches described in this section.

Edge deployments

When Kestrel or HTTP.sys is used as a public-facing edge server, Kestrel or HTTP.sys must be configured to listen on both:

- The secure port where the client is redirected (typically, 443 in production and 5001 in development).
- The insecure port (typically, 80 in production and 5000 in development).

The insecure port must be accessible by the client in order for the app to receive an insecure request and redirect the client to the secure port.

For more information, see Kestrel endpoint configuration or HTTP.sys web server implementation in ASP.NET Core.

Deployment scenarios

Any firewall between the client and server must also have communication ports open for traffic.

If requests are forwarded in a reverse proxy configuration, use Forwarded Headers Middleware before calling HTTPS Redirection Middleware. Forwarded Headers Middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header. The middleware permits redirect URIs and other security policies to work correctly. When Forwarded Headers Middleware isn't used, the backend app might not receive the correct scheme and end up in a redirect loop. A common end user error message is that too many redirects have occurred.

When deploying to Azure App Service, follow the guidance in Tutorial: Bind an existing custom SSL certificate to Azure Web Apps.

Options

The following highlighted code calls AddHttpsRedirection to configure middleware options:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddHsts(options =>
    {
        options.Preload = true;
        options.IncludeSubDomains = true;
        options.MaxAge = TimeSpan.FromDays(60);
        options.ExcludedHosts.Add("example.com");
        options.ExcludedHosts.Add("www.example.com");
    });

    services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
        options.HttpsPort = 5001;
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddHsts(options =>
    {
        options.Preload = true;
        options.IncludeSubDomains = true;
        options.MaxAge = TimeSpan.FromDays(60);
        options.ExcludedHosts.Add("example.com");
        options.ExcludedHosts.Add("www.example.com");
    });

    services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
        options.HttpsPort = 5001;
    });
}
```

Calling `AddHttpsRedirection` is only necessary to change the values of `HttpsPort` or `RedirectStatusCode`.

The preceding highlighted code:

- Sets HttpsRedirectionOptions.RedirectStatusCode to Status307TemporaryRedirect, which is the default value. Use the fields of the StatusCodes class for assignments to `RedirectStatusCode`.
- Sets the HTTPS port to 5001.

Configure permanent redirects in production

The middleware defaults to sending a Status307TemporaryRedirect with all redirects. If you prefer to send a permanent redirect status code when the app is in a non-Development environment, wrap the middleware options configuration in a conditional check for a non-Development environment.

When configuring services in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // IWebHostEnvironment (stored in _env) is injected into the Startup class.
    if (!_env.IsDevelopment())
    {
        services.AddHttpsRedirection(options =>
        {
            options.RedirectStatusCode = StatusCodes.Status308PermanentRedirect;
            options.HttpsPort = 443;
        });
    }
}
```

When configuring services in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // IHostingEnvironment (stored in _env) is injected into the Startup class.
    if (!_env.IsDevelopment())
    {
        services.AddHttpsRedirection(options =>
        {
            options.RedirectStatusCode = StatusCodes.Status308PermanentRedirect;
            options.HttpsPort = 443;
        });
    }
}
```

HTTPS Redirection Middleware alternative approach

An alternative to using HTTPS Redirection Middleware (`UseHttpsRedirection`) is to use URL Rewriting Middleware (`AddRedirectToHttps`). `AddRedirectToHttps` can also set the status code and port when the redirect is executed. For more information, see URL Rewriting Middleware.

When redirecting to HTTPS without the requirement for additional redirect rules, we recommend using HTTPS Redirection Middleware (`UseHttpsRedirection`) described in this topic.

HTTP Strict Transport Security Protocol (HSTS)

Per OWASP, HTTP Strict Transport Security (HSTS) is an opt-in security enhancement that's specified by a web app through the use of a response header. When a browser that supports HSTS receives this header:

- The browser stores configuration for the domain that prevents sending any communication over HTTP. The browser forces all communication over HTTPS.
- The browser prevents the user from using untrusted or invalid certificates. The browser disables prompts that allow a user to temporarily trust such a certificate.

Because HSTS is enforced by the client, it has some limitations:

- The client must support HSTS.
- HSTS requires at least one successful HTTPS request to establish the HSTS policy.
- The application must check every HTTP request and redirect or reject the HTTP request.

ASP.NET Core 2.1 and later implements HSTS with the `UseHsts` extension method. The following code calls `UseHsts` when the app isn't in development mode:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();

    app.UseMvc();
}
```

`UseHsts` isn't recommended in development because the HSTS settings are highly cacheable by browsers. By default, `UseHsts` excludes the local loopback address.

For production environments that are implementing HTTPS for the first time, set the initial HstsOptions.MaxAge to a small value using one of the TimeSpan methods. Set the value from hours to no more than a single day in case you need to revert the HTTPS infrastructure to HTTP. After you're confident in the sustainability of the HTTPS configuration, increase the HSTS `max-age` value; a commonly used value is one year.

The following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddHsts(options =>
    {
        options.Preload = true;
        options.IncludeSubDomains = true;
        options.MaxAge = TimeSpan.FromDays(60);
        options.ExcludedHosts.Add("example.com");
        options.ExcludedHosts.Add("www.example.com");
    });

    services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
        options.HttpsPort = 5001;
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddHsts(options =>
    {
        options.Preload = true;
        options.IncludeSubDomains = true;
        options.MaxAge = TimeSpan.FromDays(60);
        options.ExcludedHosts.Add("example.com");
        options.ExcludedHosts.Add("www.example.com");
    });

    services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
        options.HttpsPort = 5001;
    });
}
```

- Sets the preload parameter of the `Strict-Transport-Security` header. Preload isn't part of the RFC HSTS specification, but is supported by web browsers to preload HSTS sites on fresh install. For more information, see https://hstspreload.org/.
- Enables includeSubDomain, which applies the HSTS policy to Host subdomains.
- Explicitly sets the `max-age` parameter of the `Strict-Transport-Security` header to 60 days. If not set, defaults to 30 days. For more information, see the max-age directive.
- Adds `example.com` to the list of hosts to exclude.

`UseHsts` excludes the following loopback hosts:

- `localhost` : The IPv4 loopback address.
- `127.0.0.1` : The IPv4 loopback address.
- `[::1]` : The IPv6 loopback address.

Opt-out of HTTPS/HSTS on project creation

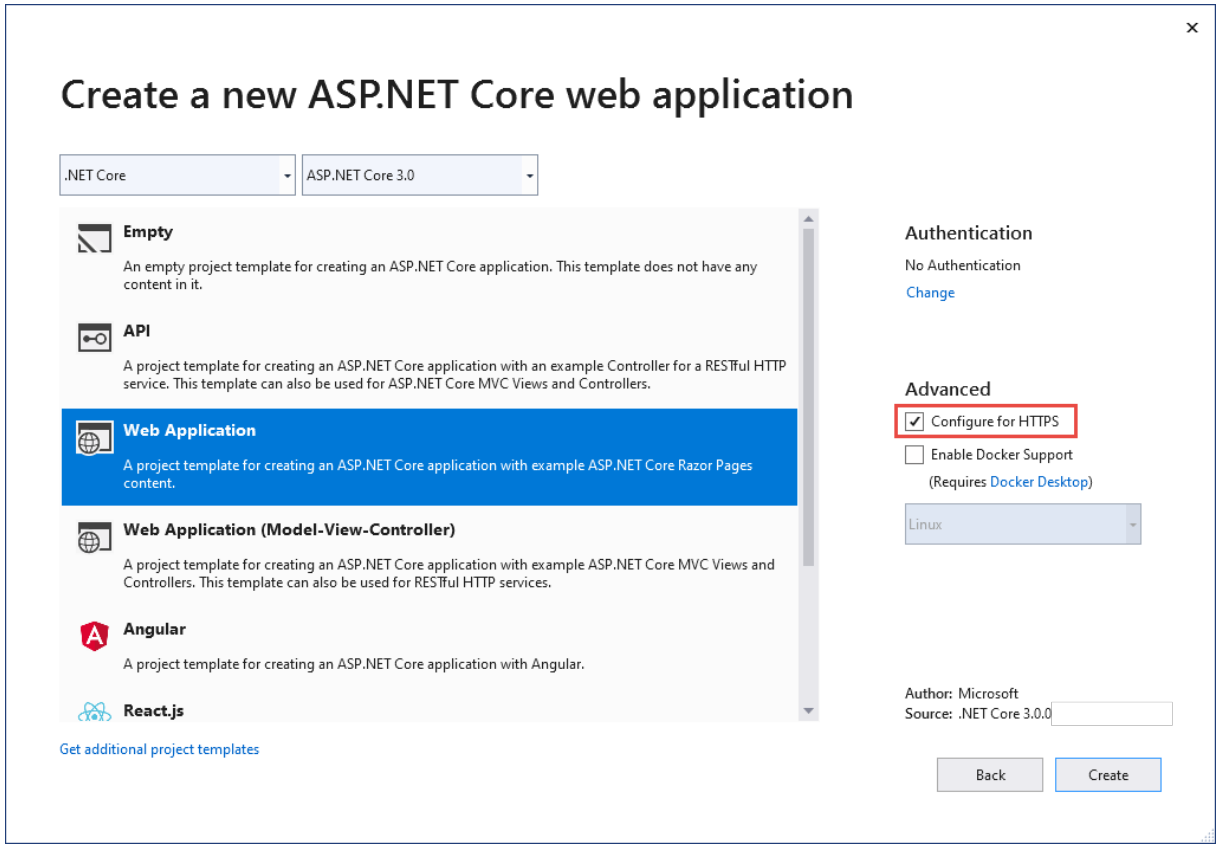
In some backend service scenarios where connection security is handled at the public-facing edge of the network, configuring connection security at each node isn't required. Web apps that are generated from the templates in Visual Studio or from the dotnet new command enable HTTPS redirection and HSTS. For

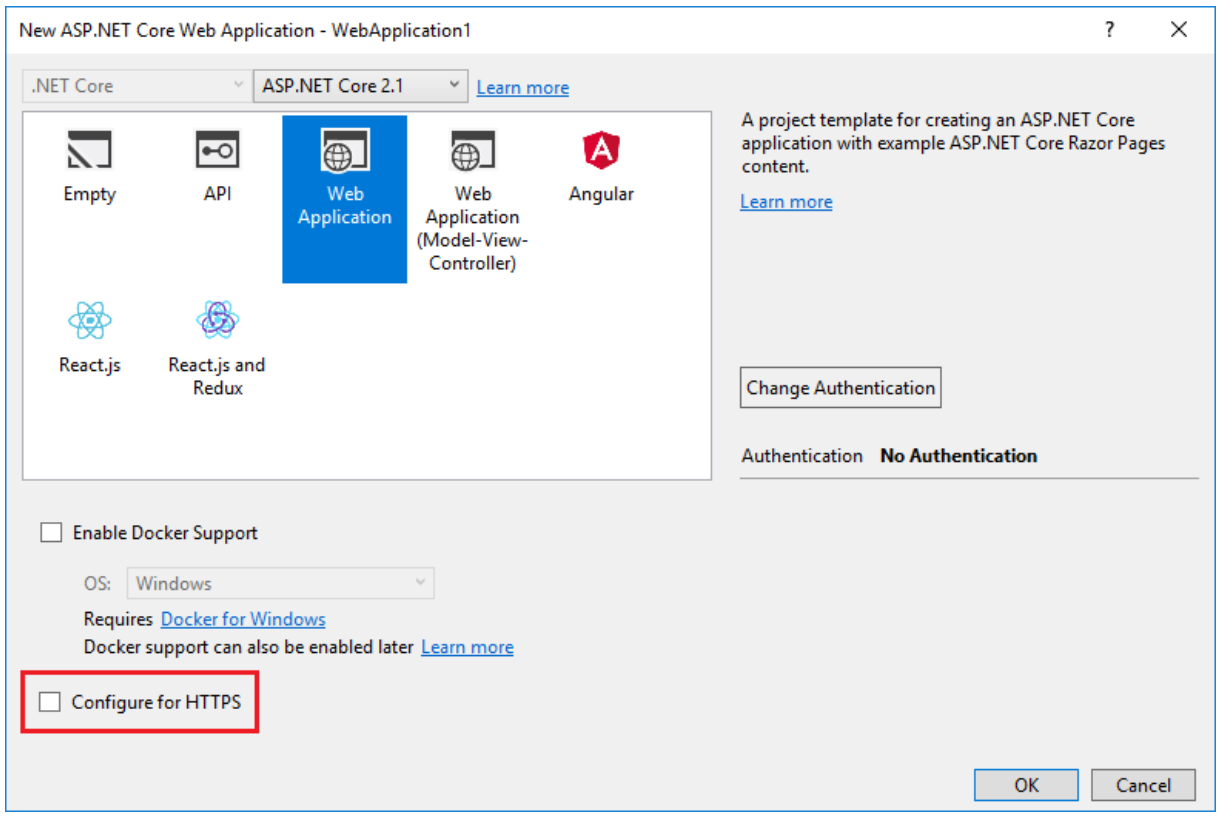
deployments that don't require these scenarios, you can opt-out of HTTPS/HSTS when the app is created from the template.

To opt-out of HTTPS/HSTS:

- Visual Studio
- .NET Core CLI

Uncheck the **Configure for HTTPS** check box.





Trust the ASP.NET Core HTTPS development certificate on Windows and macOS

The .NET Core SDK includes an HTTPS development certificate. The certificate is installed as part of the first-run experience. For example, `dotnet --info` produces a variation of the following output:

```
ASP.NET Core
------------
Successfully installed the ASP.NET Core HTTPS Development Certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and macOS only).
For establishing trust on other platforms refer to the platform specific documentation.
For more information on configuring HTTPS see https://go.microsoft.com/fwlink/?linkid=848054.
```

Installing the .NET Core SDK installs the ASP.NET Core HTTPS development certificate to the local user certificate store. The certificate has been installed, but it's not trusted. To trust the certificate, perform the one-time step to run the dotnet `dev-certs` tool:

```
dotnet dev-certs https --trust
```

The following command provides help on the `dev-certs` tool:

```
dotnet dev-certs https --help
```

How to set up a developer certificate for Docker

See this GitHub issue.

Trust HTTPS certificate on Linux

For instructions on Linux, refer to the distribution documentation.

Trust HTTPS certificate from Windows Subsystem for Linux

The Windows Subsystem for Linux (WSL) generates an HTTPS self-signed cert. To configure the Windows certificate store to trust the WSL certificate:

- Run the following command to export the WSL-generated certificate:

  ```
  dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p <cryptic-password>
  ```

- In a WSL window, run the following command:

  ```
    ASPNETCORE_Kestrel__Certificates__Default__Password="<cryptic-password>"
    ASPNETCORE_Kestrel__Certificates__Default__Path=/mnt/c/Users/user-
  name/.aspnet/https/aspnetapp.pfx
    dotnet watch run
  ```

  The preceding command sets the environment variables so Linux uses the Windows trusted certificate.

Troubleshoot certificate problems

This section provides help when the ASP.NET Core HTTPS development certificate has been installed and

trusted, but you still have browser warnings that the certificate is not trusted. The ASP.NET Core HTTPS development certificate is used by Kestrel.

All platforms - certificate not trusted

Run the following commands:

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app. Certificate trust is cached by browsers.

The preceding commands solve most browser trust issues. If the browser is still not trusting the certificate, follow the platform-specific suggestions that follow.

Docker - certificate not trusted

- Delete the *C:\Users{USER}\AppData\Roaming\ASP.NET\Https* folder.
- Clean the solution. Delete the *bin* and *obj* folders.
- Restart the development tool. For example, Visual Studio, Visual Studio Code, or Visual Studio for Mac.

Windows - certificate not trusted

- Check the certificates in the certificate store. There should be a `localhost` certificate with the `ASP.NET Core HTTPS development certificate` friendly name both under `Current User > Personal > Certificates` and `Current User > Trusted root certification authorities > Certificates`
- Remove all the found certificates from both Personal and Trusted root certification authorities. Do **not** remove the IIS Express localhost certificate.
- Run the following commands:

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app.

OS X - certificate not trusted

- Open KeyChain Access.
- Select the System keychain.
- Check for the presence of a localhost certificate.
- Check that it contains a `+` symbol on the icon to indicate it's trusted for all users.
- Remove the certificate from the system keychain.
- Run the following commands:

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app.

See HTTPS Error using IIS Express (dotnet/AspNetCore #16892) for troubleshooting certificate issues with Visual Studio.

IIS Express SSL certificate used with Visual Studio

To fix problems with the IIS Express certificate, select **Repair** from the Visual Studio installer. For more

information, see this GitHub issue.

Additional information

- Configure ASP.NET Core to work with proxy servers and load balancers
- Host ASP.NET Core on Linux with Apache: HTTPS configuration
- Host ASP.NET Core on Linux with Nginx: HTTPS configuration
- How to Set Up SSL on IIS
- OWASP HSTS browser support

Hosting ASP.NET Core images with Docker over HTTPS

9/22/2020 • 2 minutes to read • Edit Online

By Rick Anderson

ASP.NET Core uses HTTPS by default. HTTPS relies on certificates for trust, identity, and encryption.

This document explains how to run pre-built container images with HTTPS.

See Developing ASP.NET Core Applications with Docker over HTTPS for development scenarios.

This sample requires Docker 17.06 or later of the Docker client.

Prerequisites

The .NET Core 2.2 SDK or later is required for some of the instructions in this document.

Certificates

A certificate from a certificate authority is required for production hosting for a domain. Let's Encrypt is a certificate authority that offers free certificates.

This document uses self-signed development certificates for hosting pre-built images over `localhost`. The instructions are similar to using production certificates.

For production certs:

- The `dotnet dev-certs` tool is not required.
- Certificates do not need to be stored in the location used in the instructions. Any location should work, although storing certs within your site directory is not recommended.

The instructions contained in the following section volume mount certificates into containers using Docker's `-v` command-line option. You could add certificates into container images with a `COPY` command in a *Dockerfile*, but it's not recommended. Copying certificates into an image isn't recommended for the following reasons:

- It makes difficult to use the same image for testing with developer certificates.
- It makes difficult to use the same image for Hosting with production certificates.
- There is significant risk of certificate disclosure.

Running pre-built container images with HTTPS

Use the following instructions for your operating system configuration.

Windows using Linux containers

Generate certificate and configure local machine:

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p { password here }
dotnet dev-certs https --trust
```

In the preceding commands, replace `{ password here }` with a password.

Run the container image with ASP.NET Core configured for HTTPS in a command shell:

```
docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_Kestrel__Certificates__Default__Password="password" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v %USERPROFILE%\.aspnet\https:/https/
mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

The password must match the password used for the certificate.

macOS or Linux

Generate certificate and configure local machine:

```
dotnet dev-certs https -ep ${HOME}/.aspnet/https/aspnetapp.pfx -p { password here }
dotnet dev-certs https --trust
```

`dotnet dev-certs https --trust` is only supported on macOS and Windows. You need to trust certs on Linux in the way that is supported by your distribution. It is likely that you need to trust the certificate in your browser.

In the preceding commands, replace `{ password here }` with a password.

Run the container image with ASP.NET Core configured for HTTPS:

```
docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_Kestrel__Certificates__Default__Password="password" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v ${HOME}/.aspnet/https:/https/
mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

The password must match the password used for the certificate.

Windows using Windows containers

Generate certificate and configure local machine:

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p { password here }
dotnet dev-certs https --trust
```

In the preceding commands, replace `{ password here }` with a password. When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Run the container image with ASP.NET Core configured for HTTPS:

```
docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_Kestrel__Certificates__Default__Password="password" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=\https\aspnetapp.pfx -v %USERPROFILE%\.aspnet\https:C:\https\
mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

The password must match the password used for the certificate. When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Hosting ASP.NET Core images with Docker Compose over HTTPS

9/22/2020 • 2 minutes to read • Edit Online

ASP.NET Core uses HTTPS by default. HTTPS relies on certificates for trust, identity, and encryption.

This document explains how to run pre-built container images with HTTPS.

See Developing ASP.NET Core Applications with Docker over HTTPS for development scenarios.

This sample requires Docker 17.06 or later of the Docker client.

Prerequisites

The .NET Core 2.2 SDK or later is required for some of the instructions in this document.

Certificates

A certificate from a certificate authority is required for production hosting for a domain. Let's Encrypt is a certificate authority that offers free certificates.

This document uses self-signed development certificates for hosting pre-built images over `localhost`. The instructions are similar to using production certificates.

For production certificates:

- The `dotnet dev-certs` tool is not required.
- Certificates don't need to be stored in the location used in the instructions. Store the certificates in any location outside the site directory.

The instructions contained in the following section volume mount certificates into containers using the `volumes` property in *docker-compose.yml*. You could add certificates into container images with a `COPY` command in a *Dockerfile*, but it's not recommended. Copying certificates into an image isn't recommended for the following reasons:

- It makes it difficult to use the same image for testing with developer certificates.
- It makes it difficult to use the same image for Hosting with production certificates.
- There is significant risk of certificate disclosure.

Starting a container with https support using docker compose

Use the following instructions for your operating system configuration.

Windows using Linux containers

Generate certificate and configure local machine:

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p { password here }
dotnet dev-certs https --trust
```

In the preceding commands, replace `{ password here }` with a password.

Create a *docker-compose.debug.yml* file with the following content:

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/core/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
    volumes:
      - ~/.aspnet/https:/https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

```
docker-compose -f "docker-compose.debug.yml" up -d
```

macOS or Linux

Generate certificate and configure local machine:

```
dotnet dev-certs https -ep ${HOME}/.aspnet/https/aspnetapp.pfx -p { password here }
dotnet dev-certs https --trust
```

`dotnet dev-certs https --trust` is only supported on macOS and Windows. You need to trust certificates on Linux in the way that is supported by your distribution. It is likely that you need to trust the certificate in your browser.

In the preceding commands, replace `{ password here }` with a password.

Create a *docker-compose.debug.yml* file with the following content:

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/core/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
    volumes:
      - ~/.aspnet/https:/https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

```
docker-compose -f "docker-compose.debug.yml" up -d
```

Windows using Windows containers

Generate certificate and configure local machine:

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p { password here }
dotnet dev-certs https --trust
```

In the preceding commands, replace `{ password here }` with a password.

Create a *docker-compose.debug.yml* file with the following content:

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/core/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      - ASPNETCORE_Kestrel__Certificates__Default__Path=C:\https\aspnetapp.pfx
    volumes:
      - ${USERPROFILE}\.aspnet\https:C:\https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

```
docker-compose -f "docker-compose.debug.yml" up -d
```

EU General Data Protection Regulation (GDPR) support in ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

By Rick Anderson

ASP.NET Core provides APIs and templates to help meet some of the EU General Data Protection Regulation (GDPR) requirements:

- The project templates include extension points and stubbed markup that you can replace with your privacy and cookie use policy.
- The *Pages/Privacy.cshtml* page or *Views/Home/Privacy.cshtml* view provides a page to detail your site's privacy policy.

To enable the default cookie consent feature like that found in the ASP.NET Core 2.2 templates in an ASP.NET Core 3.0 template generated app:

- Add `using Microsoft.AspNetCore.Http` to the list of using directives.
- Add CookiePolicyOptions to `Startup.ConfigureServices` and UseCookiePolicy to `Startup.Configure`:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            // This lambda determines whether user consent for non-essential
            // cookies is needed for a given request.
            options.CheckConsentNeeded = context => true;
            // requires using Microsoft.AspNetCore.Http;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

- Add the cookie consent partial to the *_Layout.cshtml* file:

```
        @*Previous markup removed for brevity*@
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2019 - RPCC - <a asp-area="" asp-page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

- Add the *_CookieConsentPartial.cshtml* file to the project:

```
@using Microsoft.AspNetCore.Http.Features

@{
    var consentFeature = Context.Features.Get<ITrackingConsentFeature>();
    var showBanner = !consentFeature?.CanTrack ?? false;
    var cookieString = consentFeature?.CreateConsentCookie();
}

@if (showBanner)
{
    <div id="cookieConsent" class="alert alert-info alert-dismissible fade show" role="alert">
        Use this space to summarize your privacy and cookie use policy. <a asp-
page="/Privacy">Learn More</a>.
        <button type="button" class="accept-policy close" data-dismiss="alert" aria-
label="Close" data-cookie-string="@cookieString">
            <span aria-hidden="true">Accept</span>
        </button>
    </div>
    <script>
        (function () {
            var button = document.querySelector("#cookieConsent button[data-cookie-string]");
            button.addEventListener("click", function (event) {
                document.cookie = button.dataset.cookieString;
            }, false);
        })();
    </script>
}
```

- Select the ASP.NET Core 2.2 version of this article to read about the cookie consent feature.

- The project templates include extension points and stubbed markup that you can replace with your privacy and cookie use policy.
- A cookie consent feature allows you to ask for (and track) consent from your users for storing personal information. If a user hasn't consented to data collection and the app has CheckConsentNeeded set to `true`, non-essential cookies aren't sent to the browser.
- Cookies can be marked as essential. Essential cookies are sent to the browser even when the user hasn't

consented and tracking is disabled.

- TempData and Session cookies aren't functional when tracking is disabled.
- The Identity manage page provides a link to download and delete user data.

The sample app allows you test most of the GDPR extension points and APIs added to the ASP.NET Core 2.1 templates. See the ReadMe file for testing instructions.

View or download sample code (how to download)

ASP.NET Core GDPR support in template-generated code

Razor Pages and MVC projects created with the project templates include the following GDPR support:

- CookiePolicyOptions and UseCookiePolicy are set in the `Startup` class.
- The *_CookieConsentPartial.cshtml* partial view. An **Accept** button is included in this file. When the user clicks the **Accept** button, consent to store cookies is provided.
- The *Pages/Privacy.cshtml* page or *Views/Home/Privacy.cshtml* view provides a page to detail your site's privacy policy. The *_CookieConsentPartial.cshtml* file generates a link to the Privacy page.
- For apps created with individual user accounts, the Manage page provides links to download and delete personal user data.

CookiePolicyOptions and UseCookiePolicy

CookiePolicyOptions are initialized in `Startup.ConfigureServices`:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services
    // to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            // This lambda determines whether user consent for non-essential cookies
            // is needed for a given request.
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

        // If the app uses session state, call AddSession.
        // services.AddSession();

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    // This method gets called by the runtime. Use this method to configure the
    // HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseAuthentication();

        // If the app uses session state, call Session Middleware after Cookie
        // Policy Middleware and before MVC Middleware.
        // app.UseSession();

        app.UseMvc();
    }
}
```

UseCookiePolicy is called in `Startup.Configure`:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services
    // to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            // This lambda determines whether user consent for non-essential cookies
            // is needed for a given request.
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

        // If the app uses session state, call AddSession.
        // services.AddSession();

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    // This method gets called by the runtime. Use this method to configure the
    // HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseCookiePolicy();

        app.UseAuthentication();

        // If the app uses session state, call Session Middleware after Cookie
        // Policy Middleware and before MVC Middleware.
        // app.UseSession();

        app.UseMvc();
    }
}
```

_CookieConsentPartial.cshtml partial view

The *_CookieConsentPartial.cshtml* partial view:

```
@using Microsoft.AspNetCore.Http.Features

@{
    var consentFeature = Context.Features.Get<ITrackingConsentFeature>();
    var showBanner = !consentFeature?.CanTrack ?? false;
    var cookieString = consentFeature?.CreateConsentCookie();
}

@if (showBanner)
{
    <nav id="cookieConsent" class="navbar navbar-default navbar-fixed-top" role="alert">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target="#cookieConsent .navbar-collapse">
                    <span class="sr-only">Toggle cookie consent banner</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <span class="navbar-brand"><span class="glyphicon glyphicon-info-sign" aria-
hidden="true"></span></span>
            </div>
            <div class="collapse navbar-collapse">
                <p class="navbar-text">
                    Use this space to summarize your privacy and cookie use policy.
                </p>
                <div class="navbar-right">
                    <a asp-page="/Privacy" class="btn btn-info navbar-btn">Learn More</a>
                    <button type="button" class="btn btn-default navbar-btn" data-cookie-
string="@cookieString">Accept</button>
                </div>
            </div>
        </div>
    </nav>
    <script>
        (function () {
            document.querySelector("#cookieConsent button[data-cookie-
string]").addEventListener("click", function (el) {
                document.cookie = el.target.dataset.cookieString;
                document.querySelector("#cookieConsent").classList.add("hidden");
            }, false);
        })();
    </script>
}
```

This partial:

- Obtains the state of tracking for the user. If the app is configured to require consent, the user must consent before cookies can be tracked. If consent is required, the cookie consent panel is fixed at top of the navigation bar created by the *_Layout.cshtml* file.
- Provides an HTML `<p>` element to summarize your privacy and cookie use policy.
- Provides a link to Privacy page or view where you can detail your site's privacy policy.

Essential cookies

If consent to store cookies hasn't been provided, only cookies marked essential are sent to the browser. The following code makes a cookie essential:

```
public IActionResult OnPostCreateEssentialAsync()
{
    HttpContext.Response.Cookies.Append(Constants.EssentialSec,
        DateTime.Now.Second.ToString(),
        new CookieOptions() { IsEssential = true });

    ResponseCookies = Response.Headers[HeaderNames.SetCookie].ToString();

    return RedirectToPage("./Index");
}
```

TempData provider and session state cookies aren't essential

The TempData provider cookie isn't essential. If tracking is disabled, the TempData provider isn't functional. To enable the TempData provider when tracking is disabled, mark the TempData cookie as essential in `Startup.ConfigureServices`:

```
// The TempData provider cookie is not essential. Make it essential
// so TempData is functional when tracking is disabled.
services.Configure<CookieTempDataProviderOptions>(options => {
    options.Cookie.IsEssential = true;
});
```

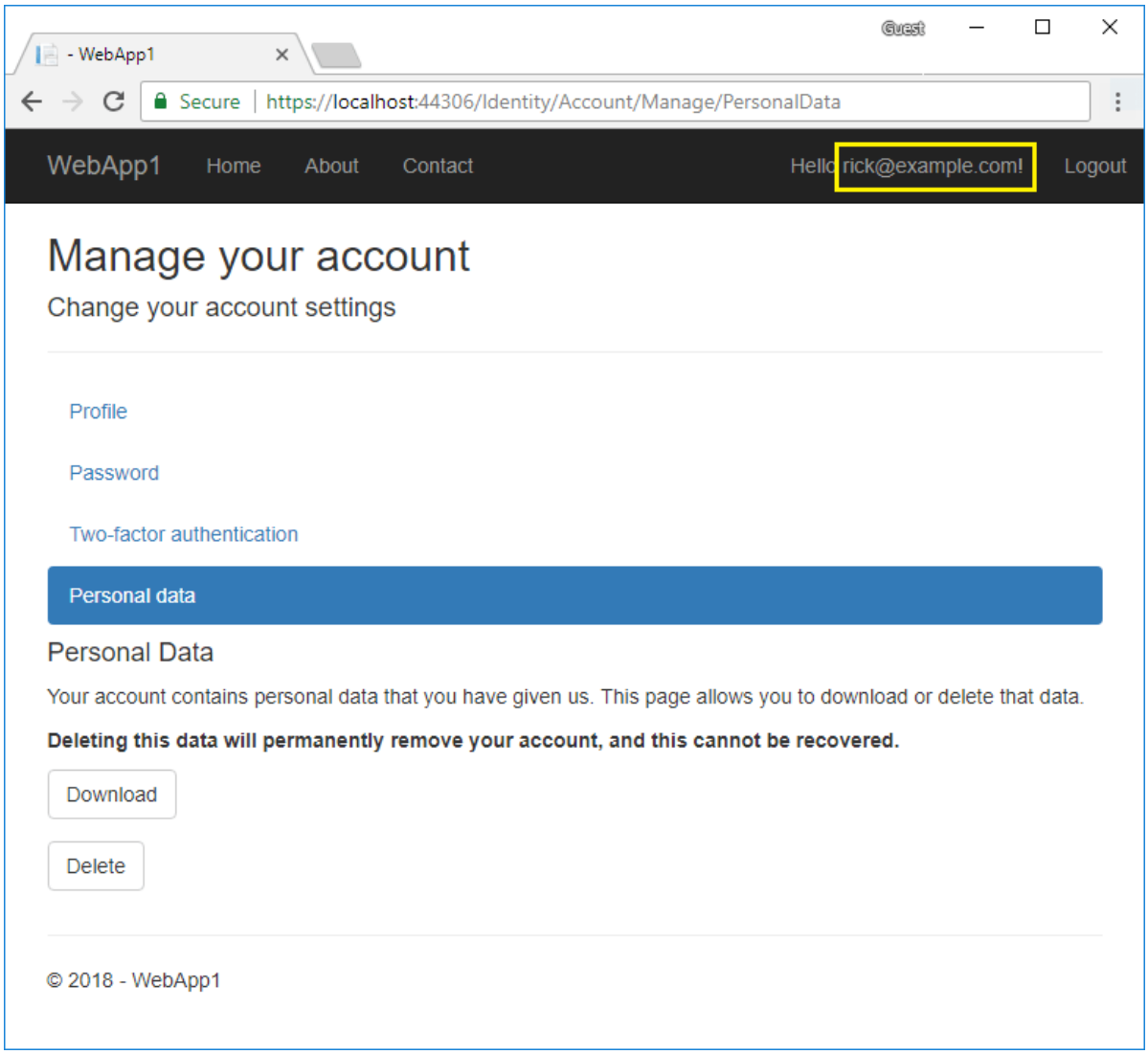
Session state cookies are not essential. Session state isn't functional when tracking is disabled. The following code makes session cookies essential:

```
services.AddSession(options =>
{
    options.Cookie.IsEssential = true;
});
```

Personal data

ASP.NET Core apps created with individual user accounts include code to download and delete personal data.

Select the user name and then select **Personal data**:



Notes:

- To generate the `Account/Manage` code, see Scaffold Identity.
- The **Delete** and **Download** links only act on the default identity data. Apps that create custom user data must be extended to delete/download the custom user data. For more information, see Add, download, and delete custom user data to Identity.
- Saved tokens for the user that are stored in the Identity database table `AspNetUserTokens` are deleted when the user is deleted via the cascading delete behavior due to the foreign key.
- External provider authentication, such as Facebook and Google, isn't available before the cookie policy is accepted.

Encryption at rest

Some databases and storage mechanisms allow for encryption at rest. Encryption at rest:

- Encrypts stored data automatically.
- Encrypts without configuration, programming, or other work for the software that accesses the data.
- Is the easiest and safest option.
- Allows the database to manage keys and encryption.

For example:

- Microsoft SQL and Azure SQL provide Transparent Data Encryption (TDE).
- SQL Azure encrypts the database by default
- Azure Blobs, Files, Table, and Queue Storage are encrypted by default.

For databases that don't provide built-in encryption at rest, you may be able to use disk encryption to provide the same protection. For example:

- BitLocker for Windows Server
- Linux:
  - eCryptfs
  - EncFS.

Additional resources

- Microsoft.com/GDPR
- GDPR - Adding a Revoke Consent Button in ASP.NET Core

Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

9/22/2020 • 14 minutes to read • Edit Online

By Rick Anderson, Fiyaz Hasan, and Steve Smith

Cross-site request forgery (also known as XSRF or CSRF) is an attack against web-hosted apps whereby a malicious web app can influence the interaction between a client browser and a web app that trusts that browser. These attacks are possible because web browsers send some types of authentication tokens automatically with every request to a website. This form of exploit is also known as a *one-click attack* or *session riding* because the attack takes advantage of the user's previously authenticated session.

An example of a CSRF attack:

1. A user signs into `www.good-banking-site.com` using forms authentication. The server authenticates the user and issues a response that includes an authentication cookie. The site is vulnerable to attack because it trusts any request that it receives with a valid authentication cookie.

2. The user visits a malicious site, `www.bad-crook-site.com`.

   The malicious site, `www.bad-crook-site.com`, contains an HTML form similar to the following:

   ```
   <h1>Congratulations! You're a Winner!</h1>
   <form action="http://good-banking-site.com/api/account" method="post">
       <input type="hidden" name="Transaction" value="withdraw">
       <input type="hidden" name="Amount" value="1000000">
       <input type="submit" value="Click to collect your prize!">
   </form>
   ```

   Notice that the form's `action` posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.

3. The user selects the submit button. The browser makes the request and automatically includes the authentication cookie for the requested domain, `www.good-banking-site.com`.

4. The request runs on the `www.good-banking-site.com` server with the user's authentication context and can perform any action that an authenticated user is allowed to perform.

In addition to the scenario where the user selects the button to submit the form, the malicious site could:

- Run a script that automatically submits the form.
- Send the form submission as an AJAX request.
- Hide the form using CSS.

These alternative scenarios don't require any action or input from the user other than initially visiting the malicious site.

Using HTTPS doesn't prevent a CSRF attack. The malicious site can send an `https://www.good-banking-site.com/` request just as easily as it can send an insecure request.

Some attacks target endpoints that respond to GET requests, in which case an image tag can be used to perform the action. This form of attack is common on forum sites that permit images but block JavaScript. Apps that change state on GET requests, where variables or resources are altered, are vulnerable to

malicious attacks. **GET requests that change state are insecure. A best practice is to never change state on a GET request.**

CSRF attacks are possible against web apps that use cookies for authentication because:

- Browsers store cookies issued by a web app.
- Stored cookies include session cookies for authenticated users.
- Browsers send all of the cookies associated with a domain to the web app every request regardless of how the request to app was generated within the browser.

However, CSRF attacks aren't limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user signs in with Basic or Digest authentication, the browser automatically sends the credentials until the session† ends.

†In this context, *session* refers to the client-side session during which the user is authenticated. It's unrelated to server-side sessions or ASP.NET Core Session Middleware.

Users can guard against CSRF vulnerabilities by taking precautions:

- Sign off of web apps when finished using them.
- Clear browser cookies periodically.

However, CSRF vulnerabilities are fundamentally a problem with the web app, not the end user.

Authentication fundamentals

Cookie-based authentication is a popular form of authentication. Token-based authentication systems are growing in popularity, especially for Single Page Applications (SPAs).

Cookie-based authentication

When a user authenticates using their username and password, they're issued a token, containing an authentication ticket that can be used for authentication and authorization. The token is stored as a cookie that accompanies every request the client makes. Generating and validating this cookie is performed by the Cookie Authentication Middleware. The middleware serializes a user principal into an encrypted cookie. On subsequent requests, the middleware validates the cookie, recreates the principal, and assigns the principal to the User property of HttpContext.

Token-based authentication

When a user is authenticated, they're issued a token (not an antiforgery token). The token contains user information in the form of claims or a reference token that points the app to user state maintained in the app. When a user attempts to access a resource requiring authentication, the token is sent to the app with an additional authorization header in form of Bearer token. This makes the app stateless. In each subsequent request, the token is passed in the request for server-side validation. This token isn't *encrypted*; it's *encoded*. On the server, the token is decoded to access its information. To send the token on subsequent requests, store the token in the browser's local storage. Don't be concerned about CSRF vulnerability if the token is stored in the browser's local storage. CSRF is a concern when the token is stored in a cookie. For more information, see the GitHub issue SPA code sample adds two cookies.

Multiple apps hosted at one domain

Shared hosting environments are vulnerable to session hijacking, login CSRF, and other attacks.

Although `example1.contoso.net` and `example2.contoso.net` are different hosts, there's an implicit trust relationship between hosts under the `*.contoso.net` domain. This implicit trust relationship allows potentially untrusted hosts to affect each other's cookies (the same-origin policies that govern AJAX requests don't necessarily apply to HTTP cookies).

Attacks that exploit trusted cookies between apps hosted on the same domain can be prevented by not sharing domains. When each app is hosted on its own domain, there is no implicit cookie trust relationship to exploit.

ASP.NET Core antiforgery configuration

> **WARNING**
> ASP.NET Core implements antiforgery using ASP.NET Core Data Protection. The data protection stack must be configured to work in a server farm. See Configuring data protection for more information.

Antiforgery middleware is added to the Dependency injection container when one of the following APIs is called in `Startup.ConfigureServices`:

- AddMvc
- MapRazorPages
- MapControllerRoute
- MapBlazorHub

Antiforgery middleware is added to the Dependency injection container when AddMvc is called in `Startup.ConfigureServices`

In ASP.NET Core 2.0 or later, the FormTagHelper injects antiforgery tokens into HTML form elements. The following markup in a Razor file automatically generates antiforgery tokens:

```
<form method="post">
    ...
</form>
```

Similarly, IHtmlHelper.BeginForm generates antiforgery tokens by default if the form's method isn't GET.

The automatic generation of antiforgery tokens for HTML form elements happens when the `<form>` tag contains the `method="post"` attribute and either of the following are true:

- The action attribute is empty (`action=""`).
- The action attribute isn't supplied (`<form method="post">`).

Automatic generation of antiforgery tokens for HTML form elements can be disabled:

- Explicitly disable antiforgery tokens with the `asp-antiforgery` attribute:

  ```
  <form method="post" asp-antiforgery="false">
      ...
  </form>
  ```

- The form element is opted-out of Tag Helpers by using the Tag Helper ! opt-out symbol:

  ```
  <!form method="post">
      ...
  </!form>
  ```

- Remove the `FormTagHelper` from the view. The `FormTagHelper` can be removed from a view by adding the following directive to the Razor view:

```
@removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper,
Microsoft.AspNetCore.Mvc.TagHelpers
```

> **NOTE**
>
> Razor Pages are automatically protected from XSRF/CSRF. For more information, see XSRF/CSRF and Razor Pages.

The most common approach to defending against CSRF attacks is to use the *Synchronizer Token Pattern* (STP). STP is used when the user requests a page with form data:

1. The server sends a token associated with the current user's identity to the client.
2. The client sends back the token to the server for verification.
3. If the server receives a token that doesn't match the authenticated user's identity, the request is rejected.

The token is unique and unpredictable. The token can also be used to ensure proper sequencing of a series of requests (for example, ensuring the request sequence of: page 1 > page 2 > page 3). All of the forms in ASP.NET Core MVC and Razor Pages templates generate antiforgery tokens. The following pair of view examples generate antiforgery tokens:

```
<form asp-controller="Manage" asp-action="ChangePassword" method="post">
    ...
</form>

@using (Html.BeginForm("ChangePassword", "Manage"))
{
    ...
}
```

Explicitly add an antiforgery token to a `<form>` element without using Tag Helpers with the HTML helper `@Html.AntiForgeryToken`:

```
<form action="/" method="post">
    @Html.AntiForgeryToken()
</form>
```

In each of the preceding cases, ASP.NET Core adds a hidden form field similar to the following:

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">
```

ASP.NET Core includes three filters for working with antiforgery tokens:

- ValidateAntiForgeryToken
- AutoValidateAntiforgeryToken
- IgnoreAntiforgeryToken

Antiforgery options

Customize antiforgery options in `Startup.ConfigureServices`:

```
services.AddAntiforgery(options =>
{
    // Set Cookie properties using CookieBuilder properties†.
    options.FormFieldName = "AntiforgeryFieldname";
    options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
    options.SuppressXFrameOptionsHeader = false;
});
```

†Set the antiforgery `Cookie` properties using the properties of the CookieBuilder class.

OPTION	DESCRIPTION
Cookie	Determines the settings used to create the antiforgery cookies.
FormFieldName	The name of the hidden form field used by the antiforgery system to render antiforgery tokens in views.
HeaderName	The name of the header used by the antiforgery system. If `null`, the system considers only form data.
SuppressXFrameOptionsHeader	Specifies whether to suppress generation of the `X-Frame-Options` header. By default, the header is generated with a value of "SAMEORIGIN". Defaults to `false`.

```
services.AddAntiforgery(options =>
{
    options.CookieDomain = "contoso.com";
    options.CookieName = "X-CSRF-TOKEN-COOKIENAME";
    options.CookiePath = "Path";
    options.FormFieldName = "AntiforgeryFieldname";
    options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
    options.RequireSsl = false;
    options.SuppressXFrameOptionsHeader = false;
});
```

OPTION	DESCRIPTION
Cookie	Determines the settings used to create the antiforgery cookies.
CookieDomain	The domain of the cookie. Defaults to `null`. This property is obsolete and will be removed in a future version. The recommended alternative is Cookie.Domain.
CookieName	The name of the cookie. If not set, the system generates a unique name beginning with the DefaultCookiePrefix (".AspNetCore.Antiforgery."). This property is obsolete and will be removed in a future version. The recommended alternative is Cookie.Name.
CookiePath	The path set on the cookie. This property is obsolete and will be removed in a future version. The recommended alternative is Cookie.Path.

OPTION	DESCRIPTION
FormFieldName	The name of the hidden form field used by the antiforgery system to render antiforgery tokens in views.
HeaderName	The name of the header used by the antiforgery system. If `null`, the system considers only form data.
RequireSsl	Specifies whether HTTPS is required by the antiforgery system. If `true`, non-HTTPS requests fail. Defaults to `false`. This property is obsolete and will be removed in a future version. The recommended alternative is to set Cookie.SecurePolicy.
SuppressXFrameOptionsHeader	Specifies whether to suppress generation of the `X-Frame-Options` header. By default, the header is generated with a value of "SAMEORIGIN". Defaults to `false`.

For more information, see CookieAuthenticationOptions.

Configure antiforgery features with IAntiforgery

IAntiforgery provides the API to configure antiforgery features. `IAntiforgery` can be requested in the `Configure` method of the `Startup` class. The following example uses middleware from the app's home page to generate an antiforgery token and send it in the response as a cookie (using the default Angular naming convention described later in this topic):

```
public void Configure(IApplicationBuilder app, IAntiforgery antiforgery)
{
    app.Use(next => context =>
    {
        string path = context.Request.Path.Value;

        if (
            string.Equals(path, "/", StringComparison.OrdinalIgnoreCase) ||
            string.Equals(path, "/index.html", StringComparison.OrdinalIgnoreCase))
        {
            // The request token can be sent as a JavaScript-readable cookie,
            // and Angular uses it by default.
            var tokens = antiforgery.GetAndStoreTokens(context);
            context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken,
                new CookieOptions() { HttpOnly = false });
        }

        return next(context);
    });
}
```

Require antiforgery validation

ValidateAntiForgeryToken is an action filter that can be applied to an individual action, a controller, or globally. Requests made to actions that have this filter applied are blocked unless the request includes a valid antiforgery token.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> RemoveLogin(RemoveLoginViewModel account)
{
    ManageMessageId? message = ManageMessageId.Error;
    var user = await GetCurrentUserAsync();

    if (user != null)
    {
        var result =
            await _userManager.RemoveLoginAsync(
                user, account.LoginProvider, account.ProviderKey);

        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            message = ManageMessageId.RemoveLoginSuccess;
        }
    }

    return RedirectToAction(nameof(ManageLogins), new { Message = message });
}
```

The `ValidateAntiForgeryToken` attribute requires a token for requests to the action methods it marks, including HTTP GET requests. If the `ValidateAntiForgeryToken` attribute is applied across the app's controllers, it can be overridden with the `IgnoreAntiforgeryToken` attribute.

> **NOTE**
> ASP.NET Core doesn't support adding antiforgery tokens to GET requests automatically.

Automatically validate antiforgery tokens for unsafe HTTP methods only

ASP.NET Core apps don't generate antiforgery tokens for safe HTTP methods (GET, HEAD, OPTIONS, and TRACE). Instead of broadly applying the `ValidateAntiForgeryToken` attribute and then overriding it with `IgnoreAntiforgeryToken` attributes, the AutoValidateAntiforgeryToken attribute can be used. This attribute works identically to the `ValidateAntiForgeryToken` attribute, except that it doesn't require tokens for requests made using the following HTTP methods:

- GET
- HEAD
- OPTIONS
- TRACE

We recommend use of `AutoValidateAntiforgeryToken` broadly for non-API scenarios. This ensures POST actions are protected by default. The alternative is to ignore antiforgery tokens by default, unless `ValidateAntiForgeryToken` is applied to individual action methods. It's more likely in this scenario for a POST action method to be left unprotected by mistake, leaving the app vulnerable to CSRF attacks. All POSTs should send the antiforgery token.

APIs don't have an automatic mechanism for sending the non-cookie part of the token. The implementation probably depends on the client code implementation. Some examples are shown below:

Class-level example:

```
[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{
```

Global example:

services.AddMvc(options => options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));

```
services.AddControllersWithViews(options =>
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));
```

Override global or controller antiforgery attributes

The IgnoreAntiforgeryToken filter is used to eliminate the need for an antiforgery token for a given action (or controller). When applied, this filter overrides `ValidateAntiForgeryToken` and `AutoValidateAntiforgeryToken` filters specified at a higher level (globally or on a controller).

```
[Authorize]
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
{
    [HttpPost]
    [IgnoreAntiforgeryToken]
    public async Task<IActionResult> DoSomethingSafe(SomeViewModel model)
    {
        // no antiforgery token required
    }
}
```

Refresh tokens after authentication

Tokens should be refreshed after the user is authenticated by redirecting the user to a view or Razor Pages page.

JavaScript, AJAX, and SPAs

In traditional HTML-based apps, antiforgery tokens are passed to the server using hidden form fields. In modern JavaScript-based apps and SPAs, many requests are made programmatically. These AJAX requests may use other techniques (such as request headers or cookies) to send the token.

If cookies are used to store authentication tokens and to authenticate API requests on the server, CSRF is a potential problem. If local storage is used to store the token, CSRF vulnerability might be mitigated because values from local storage aren't sent automatically to the server with every request. Thus, using local storage to store the antiforgery token on the client and sending the token as a request header is a recommended approach.

JavaScript

Using JavaScript with views, the token can be created using a service from within the view. Inject the Microsoft.AspNetCore.Antiforgery.IAntiforgery service into the view and call GetAndStoreTokens:

```
@{
    ViewData["Title"] = "AJAX Demo";
}
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Xsrf
@functions{
    public string GetAntiXsrfRequestToken()
    {
        return Xsrf.GetAndStoreTokens(Context).RequestToken;
    }
}

<input type="hidden" id="RequestVerificationToken"
       name="RequestVerificationToken" value="@GetAntiXsrfRequestToken()">

<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<div class="row">
    <p><input type="button" id="antiforgery" value="Antiforgery"></p>
    <script>
        var xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
            if (xhttp.readyState == XMLHttpRequest.DONE) {
                if (xhttp.status == 200) {
                    alert(xhttp.responseText);
                } else {
                    alert('There was an error processing the AJAX request.');
                }
            }
        };

        document.addEventListener('DOMContentLoaded', function() {
            document.getElementById("antiforgery").onclick = function () {
                xhttp.open('POST', '@Url.Action("Antiforgery", "Home")', true);
                xhttp.setRequestHeader("RequestVerificationToken",
                    document.getElementById('RequestVerificationToken').value);
                xhttp.send();
            }
        });
    </script>
</div>
```

This approach eliminates the need to deal directly with setting cookies from the server or reading them from the client.

The preceding example uses JavaScript to read the hidden field value for the AJAX POST header.

JavaScript can also access tokens in cookies and use the cookie's contents to create a header with the token's value.

```
context.Response.Cookies.Append("CSRF-TOKEN", tokens.RequestToken,
    new Microsoft.AspNetCore.Http.CookieOptions { HttpOnly = false });
```

Assuming the script requests to send the token in a header called `X-CSRF-TOKEN`, configure the antiforgery service to look for the `X-CSRF-TOKEN` header:

```
services.AddAntiforgery(options => options.HeaderName = "X-CSRF-TOKEN");
```

The following example uses JavaScript to make an AJAX request with the appropriate header:

```
function getCookie(cname) {
    var name = cname + "=";
    var decodedCookie = decodeURIComponent(document.cookie);
    var ca = decodedCookie.split(';');
    for(var i = 0; i <ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') {
            c = c.substring(1);
        }
        if (c.indexOf(name) == 0) {
            return c.substring(name.length, c.length);
        }
    }
    return "";
}

var csrfToken = getCookie("CSRF-TOKEN");

var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (xhttp.readyState == XMLHttpRequest.DONE) {
        if (xhttp.status == 200) {
            alert(xhttp.responseText);
        } else {
            alert('There was an error processing the AJAX request.');
        }
    }
};
xhttp.open('POST', '/api/password/changepassword', true);
xhttp.setRequestHeader("Content-type", "application/json");
xhttp.setRequestHeader("X-CSRF-TOKEN", csrfToken);
xhttp.send(JSON.stringify({ "newPassword": "ReallySecurePassword999$$$" }));
```

AngularJS

AngularJS uses a convention to address CSRF. If the server sends a cookie with the name `XSRF-TOKEN`, the AngularJS `$http` service adds the cookie value to a header when it sends a request to the server. This process is automatic. The header doesn't need to be set in the client explicitly. The header name is `X-XSRF-TOKEN`. The server should detect this header and validate its contents.

For ASP.NET Core API to work with this convention in your application startup:

- Configure your app to provide a token in a cookie called `XSRF-TOKEN`.
- Configure the antiforgery service to look for a header named `X-XSRF-TOKEN`.

```
public void Configure(IApplicationBuilder app, IAntiforgery antiforgery)
{
    app.Use(next => context =>
    {
        string path = context.Request.Path.Value;

        if (
            string.Equals(path, "/", StringComparison.OrdinalIgnoreCase) ||
            string.Equals(path, "/index.html", StringComparison.OrdinalIgnoreCase))
        {
            // The request token can be sent as a JavaScript-readable cookie,
            // and Angular uses it by default.
            var tokens = antiforgery.GetAndStoreTokens(context);
            context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken,
                new CookieOptions() { HttpOnly = false });
        }

        return next(context);
    });
}

public void ConfigureServices(IServiceCollection services)
{
    // Angular's default header name for sending the XSRF token.
    services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");
}
```

View or download sample code (how to download)

Extend antiforgery

The IAntiForgeryAdditionalDataProvider type allows developers to extend the behavior of the anti-CSRF system by round-tripping additional data in each token. The GetAdditionalData method is called each time a field token is generated, and the return value is embedded within the generated token. An implementer could return a timestamp, a nonce, or any other value and then call ValidateAdditionalData to validate this data when the token is validated. The client's username is already embedded in the generated tokens, so there's no need to include this information. If a token includes supplemental data but no `IAntiForgeryAdditionalDataProvider` is configured, the supplemental data isn't validated.

Additional resources

- CSRF on Open Web Application Security Project (OWASP).
- Host ASP.NET Core in a web farm

Prevent open redirect attacks in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

A web app that redirects to a URL that's specified via the request such as the querystring or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack.

Whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn't been tampered with. ASP.NET Core has built-in functionality to help protect apps from open redirect (also known as open redirection) attacks.

What is an open redirect attack?

Web applications frequently redirect users to a login page when they access resources that require authentication. The redirection typically includes a `returnUrl` querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in. After the user authenticates, they're redirected to the URL they had originally requested.

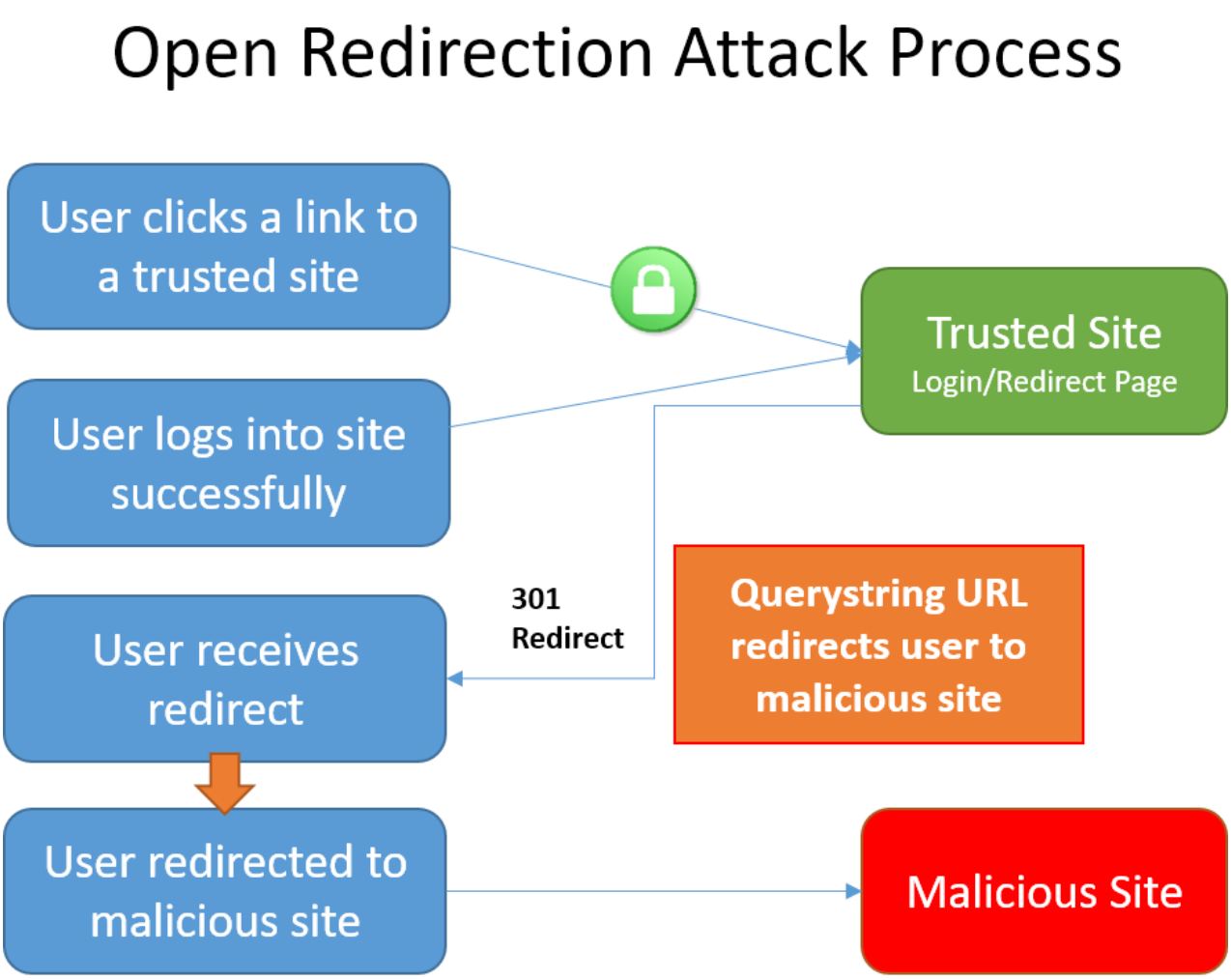
Because the destination URL is specified in the querystring of the request, a malicious user could tamper with the querystring. A tampered querystring could allow the site to redirect the user to an external, malicious site. This technique is called an open redirect (or redirection) attack.

An example attack

A malicious user can develop an attack intended to allow the malicious user access to a user's credentials or sensitive information. To begin the attack, the malicious user convinces the user to click a link to your site's login page with a `returnUrl` querystring value added to the URL. For example, consider an app at `contoso.com` that includes a login page at `http://contoso.com/Account/LogOn?returnUrl=/Home/About`. The attack follows these steps:

1. The user clicks a malicious link to `http://contoso.com/Account/LogOn?returnUrl=http://contoso1.com/Account/LogOn` (the second URL is "contoso**1**.com", not "contoso.com").
2. The user logs in successfully.
3. The user is redirected (by the site) to `http://contoso1.com/Account/LogOn` (a malicious site that looks exactly like real site).
4. The user logs in again (giving malicious site their credentials) and is redirected back to the real site.

The user likely believes that their first attempt to log in failed and that their second attempt is successful. The user most likely remains unaware that their credentials are compromised.



In addition to login pages, some sites provide redirect pages or endpoints. Imagine your app has a page with an open redirect, `/Home/Redirect`. An attacker could create, for example, a link in an email that goes to `[yoursite]/Home/Redirect?url=http://phishingsite.com/Home/Login`. A typical user will look at the URL and see it begins with your site name. Trusting that, they will click the link. The open redirect would then send the user to the phishing site, which looks identical to yours, and the user would likely login to what they believe is your site.

Protecting against open redirect attacks

When developing web applications, treat all user-provided data as untrustworthy. If your application has functionality that redirects the user based on the contents of the URL, ensure that such redirects are only done locally within your app (or to a known URL, not any URL that may be supplied in the querystring).

LocalRedirect

Use the `LocalRedirect` helper method from the base `Controller` class:

```
public IActionResult SomeAction(string redirectUrl)
{
    return LocalRedirect(redirectUrl);
}
```

`LocalRedirect` will throw an exception if a non-local URL is specified. Otherwise, it behaves just like the `Redirect` method.

IsLocalUrl

Use the IsLocalUrl method to test URLs before redirecting:

The following example shows how to check whether a URL is local before redirecting.

```
private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Home");
    }
}
```

The `IsLocalUrl` method protects users from being inadvertently redirected to a malicious site. You can log the details of the URL that was provided when a non-local URL is supplied in a situation where you expected a local URL. Logging redirect URLs may help in diagnosing redirection attacks.

Prevent Cross-Site Scripting (XSS) in ASP.NET Core

9/22/2020 • 7 minutes to read • Edit Online

By Rick Anderson

Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages the attacker's scripts will run, enabling the attacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it to a page without validating, encoding or escaping it.

Protecting your application against XSS

At a basic level XSS works by tricking your application into inserting a `<script>` tag into your rendered page, or by inserting an `On*` event into an element. Developers should use the following prevention steps to avoid introducing XSS into their application.

1. Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by an attacker, HTML form inputs, query strings, HTTP headers, even data sourced from a database as an attacker may be able to breach your database even if they cannot breach your application.
2. Before putting untrusted data inside an HTML element ensure it's HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like <
3. Before putting untrusted data into an HTML attribute ensure it's HTML encoded. HTML attribute encoding is a superset of HTML encoding and encodes additional characters such as " and '.
4. Before putting untrusted data into JavaScript place the data in an HTML element whose contents you retrieve at runtime. If this isn't possible, then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example < would be encoded as `\u003C`.
5. Before putting untrusted data into a URL query string ensure it's URL encoded.

HTML Encoding using Razor

The Razor engine used in MVC automatically encodes all output sourced from variables, unless you work really hard to prevent it doing so. It uses HTML attribute encoding rules whenever you use the @ directive. As HTML attribute encoding is a superset of HTML encoding this means you don't have to concern yourself with whether you should use HTML encoding or HTML attribute encoding. You must ensure that you only use @ in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Tag helpers will also encode input you use in tag parameters.

Take the following Razor view:

```
@{
        var untrustedInput = "<\"123\">";
    }

    @untrustedInput
```

This view outputs the contents of the *untrustedInput* variable. This variable includes some characters which are used in XSS attacks, namely <, " and >. Examining the source shows the rendered output encoded as:

```
&lt;&quot;123&quot;&gt;
```

WARNING

ASP.NET Core MVC provides an `HtmlString` class which isn't automatically encoded upon output. This should never be used in combination with untrusted input as this will expose an XSS vulnerability.

JavaScript Encoding using Razor

There may be times you want to insert a value into JavaScript to process in your view. There are two ways to do this. The safest way to insert values is to place the value in a data attribute of a tag and retrieve it in your JavaScript. For example:

```
@{
    var untrustedInput = "<script>alert(1)</script>";
}

<div id="injectedData"
     data-untrustedinput="@untrustedInput" />

<div id="scriptedWrite" />
<div id="scriptedWrite-html5" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    // Put the injected, untrusted data into the scriptedWrite div tag.
    // Do NOT use document.write() on dynamically generated data as it
    // can lead to XSS.

    document.getElementById("scriptedWrite").innerText += clientSideUntrustedInputOldStyle;

    // Or you can use createElement() to dynamically create document elements
    // This time we're using textContent to ensure the data is properly encoded.
    var x = document.createElement("div");
    x.textContent = clientSideUntrustedInputHtml5;
    document.body.appendChild(x);

    // You can also use createTextNode on an element to ensure data is properly encoded.
    var y = document.createElement("div");
    y.appendChild(document.createTextNode(clientSideUntrustedInputHtml5));
    document.body.appendChild(y);

</script>
```

The preceding markup generates the following HTML:

```
<div id="injectedData"
     data-untrustedinput="&lt;script&gt;alert(1)&lt;/script&gt;" />

<div id="scriptedWrite" />
<div id="scriptedWrite-html5" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    // Put the injected, untrusted data into the scriptedWrite div tag.
    // Do NOT use document.write() on dynamically generated data as it can
    // lead to XSS.

    document.getElementById("scriptedWrite").innerText += clientSideUntrustedInputOldStyle;

    // Or you can use createElement() to dynamically create document elements
    // This time we're using textContent to ensure the data is properly encoded.
    var x = document.createElement("div");
    x.textContent = clientSideUntrustedInputHtml5;
    document.body.appendChild(x);

    // You can also use createTextNode on an element to ensure data is properly encoded.
    var y = document.createElement("div");
    y.appendChild(document.createTextNode(clientSideUntrustedInputHtml5));
    document.body.appendChild(y);

</script>
```

The preceding code generates the following output:

```
<script>alert(1)</script>
<script>alert(1)</script>
<script>alert(1)</script>
```

WARNING

Do ***NOT*** concatenate untrusted input in JavaScript to create DOM elements or use `document.write()` on dynamically generated content.

Use one of the following approaches to prevent code from being exposed to DOM-based XSS:

- `createElement()` and assign property values with appropriate methods or properties such as `node.textContent=` or node.InnerText=`.
- `document.CreateTextNode()` and append it in the appropriate DOM location.
- `element.SetAttribute()`
- `element[attribute]=`

Accessing encoders in code

The HTML, JavaScript and URL encoders are available to your code in two ways, you can inject them via dependency injection or you can use the default encoders contained in the `System.Text.Encodings.Web` namespace. If you use the default encoders then any you applied to character ranges to be treated as safe won't take effect -

the default encoders use the safest encoding rules possible.

To use the configurable encoders via DI your constructors should take an *HtmlEncoder*, *JavaScriptEncoder* and *UrlEncoder* parameter as appropriate. For example;

```
public class HomeController : Controller
   {
       HtmlEncoder _htmlEncoder;
       JavaScriptEncoder _javaScriptEncoder;
       UrlEncoder _urlEncoder;

       public HomeController(HtmlEncoder htmlEncoder,
                             JavaScriptEncoder javascriptEncoder,
                             UrlEncoder urlEncoder)
       {
           _htmlEncoder = htmlEncoder;
           _javaScriptEncoder = javascriptEncoder;
           _urlEncoder = urlEncoder;
       }
   }
```

Encoding URL Parameters

If you want to build a URL query string with untrusted input as a value use the `UrlEncoder` to encode the value. For example,

```
var example = "\"Quoted Value with spaces and &\"";
   var encodedValue = _urlEncoder.Encode(example);
```

After encoding the encodedValue variable will contain `%22Quoted%20Value%20with%20spaces%20and%20%26%22`. Spaces, quotes, punctuation and other unsafe characters will be percent encoded to their hexadecimal value, for example a space character will become %20.

> **WARNING**
> Don't use untrusted input as part of a URL path. Always pass untrusted input as a query string value.

Customizing the Encoders

By default encoders use a safe list limited to the Basic Latin Unicode range and encode all characters outside of that range as their character code equivalents. This behavior also affects Razor TagHelper and HtmlHelper rendering as it will use the encoders to output your strings.

The reasoning behind this is to protect against unknown or future browser bugs (previous browser bugs have tripped up parsing based on the processing of non-English characters). If your web site makes heavy use of non-Latin characters, such as Chinese, Cyrillic or others this is probably not the behavior you want.

You can customize the encoder safe lists to include Unicode ranges appropriate to your application during startup, in `ConfigureServices()`.

For example, using the default configuration you might use a Razor HtmlHelper like so;

```
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

When you view the source of the web page you will see it has been rendered as follows, with the Chinese text

encoded;

```
<p>This link text is in Chinese: <a href="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

To widen the characters treated as safe by the encoder you would insert the following line into the `ConfigureServices()` method in `startup.cs`;

```
services.AddSingleton<HtmlEncoder>(
     HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,
                                               UnicodeRanges.CjkUnifiedIdeographs }));
```

This example widens the safe list to include the Unicode Range CjkUnifiedIdeographs. The rendered output would now become

```
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

Safe list ranges are specified as Unicode code charts, not languages. The Unicode standard has a list of code charts you can use to find the chart containing your characters. Each encoder, Html, JavaScript and Url, must be configured separately.

> **NOTE**
>
> Customization of the safe list only affects encoders sourced via DI. If you directly access an encoder via `System.Text.Encodings.Web.*Encoder.Default` then the default, Basic Latin only safelist will be used.

Where should encoding take place?

The general accepted practice is that encoding takes place at the point of output and encoded values should never be stored in a database. Encoding at the point of output allows you to change the use of data, for example, from HTML to a query string value. It also enables you to easily search your data without having to encode values before searching and allows you to take advantage of any changes or bug fixes made to encoders.

Validation as an XSS prevention technique

Validation can be a useful tool in limiting XSS attacks. For example, a numeric string containing only the characters 0-9 won't trigger an XSS attack. Validation becomes more complicated when accepting HTML in user input. Parsing HTML input is difficult, if not impossible. Markdown, coupled with a parser that strips embedded HTML, is a safer option for accepting rich input. Never rely on validation alone. Always encode untrusted input before output, no matter what validation or sanitization has been performed.

Enable Cross-Origin Requests (CORS) in ASP.NET Core

9/22/2020 • 33 minutes to read • Edit Online

By Rick Anderson and Kirk Larkin

This article shows how to enable CORS in an ASP.NET Core app.

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction is called the *same-origin policy*. The same-origin policy prevents a malicious site from reading sensitive data from another site. Sometimes, you might want to allow other sites to make cross-origin requests to your app. For more information, see the Mozilla CORS article.

Cross Origin Resource Sharing (CORS):

- Is a W3C standard that allows a server to relax the same-origin policy.
- Is **not** a security feature, CORS relaxes security. An API is not safer by allowing CORS. For more information, see How CORS works.
- Allows a server to explicitly allow some cross-origin requests while rejecting others.
- Is safer and more flexible than earlier techniques, such as JSONP.

View or download sample code (how to download)

Same origin

Two URLs have the same origin if they have identical schemes, hosts, and ports (RFC 6454).

These two URLs have the same origin:

- `https://example.com/foo.html`
- `https://example.com/bar.html`

These URLs have different origins than the previous two URLs:

- `https://example.net`: Different domain
- `https://www.example.com/foo.html`: Different subdomain
- `http://example.com/foo.html`: Different scheme
- `https://example.com:9000/foo.html`: Different port

Enable CORS

There are three ways to enable CORS:

- In middleware using a named policy or default policy.
- Using endpoint routing.
- With the [EnableCors] attribute.

Using the [EnableCors] attribute with a named policy provides the finest control in limiting endpoints that support CORS.

WARNING

UseCors must be called before UseResponseCaching when using `UseResponseCaching`.

Each approach is detailed in the following sections.

CORS with named policy and middleware

CORS Middleware handles cross-origin requests. The following code applies a CORS policy to all the app's endpoints with the specified origins:

```
public class Startup
{
    readonly string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(name: MyAllowSpecificOrigins,
                              builder =>
                              {
                                  builder.WithOrigins("http://example.com",
                                                      "http://www.contoso.com");
                              });
        });

        // services.AddResponseCaching();
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseCors(MyAllowSpecificOrigins);

        // app.UseResponseCaching();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

The preceding code:

- Sets the policy name to `_myAllowSpecificOrigins`. The policy name is arbitrary.
- Calls the UseCors extension method and specifies the `_myAllowSpecificOrigins` CORS policy. `UseCors` adds the CORS middleware. The call to `UseCors` must be placed after `UseRouting`, but before `UseAuthorization`. For more information, see Middleware order.

- Calls AddCors with a lambda expression. The lambda takes a CorsPolicyBuilder object. Configuration options, such as `WithOrigins`, are described later in this article.
- Enables the `_myAllowSpecificOrigins` CORS policy for all controller endpoints. See endpoint routing to apply a CORS policy to specific endpoints.
- When using Response Caching Middleware, call UseCors before UseResponseCaching.

With endpoint routing, the CORS middleware **must** be configured to execute between the calls to `UseRouting` and `UseEndpoints`.

See Test CORS for instructions on testing code similar to the preceding code.

The AddCors method call adds CORS services to the app's service container:

```
public class Startup
{
    readonly string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(name: MyAllowSpecificOrigins,
                              builder =>
                              {
                                  builder.WithOrigins("http://example.com",
                                                      "http://www.contoso.com");
                              });
        });

        // services.AddResponseCaching();
        services.AddControllers();
    }
```

For more information, see CORS policy options in this document.

The CorsPolicyBuilder methods can be chained, as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy(MyAllowSpecificOrigins,
                          builder =>
                          {
                              builder.WithOrigins("http://example.com",
                                                  "http://www.contoso.com")
                                                  .AllowAnyHeader()
                                                  .AllowAnyMethod();
                          });
    });

    services.AddControllers();
}
```

Note: The specified URL must **not** contain a trailing slash (`/`). If the URL terminates with `/`, the comparison returns `false` and no header is returned.

CORS with default policy and middleware

The following highlighted code enables the default CORS policy:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddDefaultPolicy(
                builder =>
                {
                    builder.WithOrigins("http://example.com",
                                        "http://www.contoso.com");
                });
        });

        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseCors();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

The preceding code applies the default CORS policy to all controller endpoints.

Enable Cors with endpoint routing

Enabling CORS on a per-endpoint basis using `RequireCors` currently does **not** support automatic preflight requests. For more information, see this GitHub issue and Test CORS with endpoint routing and [HttpOptions].

With endpoint routing, CORS can be enabled on a per-endpoint basis using the RequireCors set of extension methods:

```
public class Startup
{
    readonly string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(name: MyAllowSpecificOrigins,
                              builder =>
                              {
                                  builder.WithOrigins("http://example.com",
                                                      "http://www.contoso.com");
                              });
        });

        services.AddControllers();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseCors();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/echo",
                context => context.Response.WriteAsync("echo"))
                .RequireCors(MyAllowSpecificOrigins);

            endpoints.MapControllers()
                     .RequireCors(MyAllowSpecificOrigins);

            endpoints.MapGet("/echo2",
                context => context.Response.WriteAsync("echo2"));

            endpoints.MapRazorPages();
        });
    }
}
```

In the preceding code:

- `app.UseCors` enables the CORS middleware. Because a default policy hasn't been configured, `app.UseCors()` alone doesn't enable CORS.
- The `/echo` and controller endpoints allow cross-origin requests using the specified policy.
- The `/echo2` and Razor Pages endpoints do **not** allow cross-origin requests because no default policy was specified.

The [DisableCors] attribute does **not** disable CORS that has been enabled by endpoint routing with `RequireCors`.

See Test CORS with endpoint routing and [HttpOptions] for instructions on testing code similar to the

preceding.

Enable CORS with attributes

Enabling CORS with the [EnableCors] attribute and applying a named policy to only those endpoints that require CORS provides the finest control.

The [EnableCors] attribute provides an alternative to applying CORS globally. The `[EnableCors]` attribute enables CORS for selected endpoints, rather than all endpoints:

- `[EnableCors]` specifies the default policy.
- `[EnableCors("{Policy String}")]` specifies a named policy.

The `[EnableCors]` attribute can be applied to:

- Razor Page `PageModel`
- Controller
- Controller action method

Different policies can be applied to controllers, page models, or action methods with the `[EnableCors]` attribute. When the `[EnableCors]` attribute is applied to a controller, page model, or action method, and CORS is enabled in middleware, **both** policies are applied. **We recommend against combining policies. Use the `[EnableCors]` attribute or middleware, not both in the same app.**

The following code applies a different policy to each method:

```
[Route("api/[controller]")]
[ApiController]
public class WidgetController : ControllerBase
{
    // GET api/values
    [EnableCors("AnotherPolicy")]
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "green widget", "red widget" };
    }

    // GET api/values/5
    [EnableCors("Policy1")]
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    {
        return id switch
        {
            1 => "green widget",
            2 => "red widget",
            _ => NotFound(),
        };
    }
}
```

The following code creates two CORS policies:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy("Policy1",
                builder =>
                {
                    builder.WithOrigins("http://example.com",
                                        "http://www.contoso.com");
                });

            options.AddPolicy("AnotherPolicy",
                builder =>
                {
                    builder.WithOrigins("http://www.contoso.com")
                                        .AllowAnyHeader()
                                        .AllowAnyMethod();
                });
        });

        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();

        app.UseRouting();

        app.UseCors();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

For the finest control of limiting CORS requests:

- Use `[EnableCors("MyPolicy")]` with a named policy.
- Don't define a default policy.
- Don't use endpoint routing.

The code in the next section meets the preceding list.

See Test CORS for instructions on testing code similar to the preceding code.

Disable CORS

The [DisableCors] attribute does **not** disable CORS that has been enabled by endpoint routing.

The following code defines the CORS policy `"MyPolicy"`:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(name: "MyPolicy",
                builder =>
                {
                    builder.WithOrigins("http://example.com",
                                        "http://www.contoso.com")
                           .WithMethods("PUT", "DELETE", "GET");
                });
        });

        services.AddControllers();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseCors();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
            endpoints.MapRazorPages();
        });
    }
}
```

The following code disables CORS for the `GetValues2` action:

```
[EnableCors("MyPolicy")]
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public IActionResult Get() =>
        ControllerContext.MyDisplayRouteInfo();

    // GET api/values/5
    [HttpGet("{id}")]
    public IActionResult Get(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // PUT api/values/5
    [HttpPut("{id}")]
    public IActionResult Put(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);


    // GET: api/values/GetValues2
    [DisableCors]
    [HttpGet("{action}")]
    public IActionResult GetValues2() =>
        ControllerContext.MyDisplayRouteInfo();

}
```

The preceding code:

- Doesn't enable CORS with endpoint routing.
- Doesn't define a default CORS policy.
- Uses [EnableCors("MyPolicy")] to enable the `"MyPolicy"` CORS policy for the controller.
- Disables CORS for the `GetValues2` method.

See Test CORS for instructions on testing the preceding code.

CORS policy options

This section describes the various options that can be set in a CORS policy:

- Set the allowed origins
- Set the allowed HTTP methods
- Set the allowed request headers
- Set the exposed response headers
- Credentials in cross-origin requests
- Set the preflight expiration time

AddPolicy is called in `Startup.ConfigureServices`. For some options, it may be helpful to read the How CORS works section first.

Set the allowed origins

AllowAnyOrigin: Allows CORS requests from all origins with any scheme (`http` or `https`). `AllowAnyOrigin` is insecure because *any website* can make cross-origin requests to the app.

> **NOTE**
>
> Specifying `AllowAnyOrigin` and `AllowCredentials` is an insecure configuration and can result in cross-site request forgery. The CORS service returns an invalid CORS response when an app is configured with both methods.

`AllowAnyOrigin` affects preflight requests and the `Access-Control-Allow-Origin` header. For more information, see the Preflight requests section.

SetIsOriginAllowedToAllowWildcardSubdomains: Sets the IsOriginAllowed property of the policy to be a function that allows origins to match a configured wildcard domain when evaluating if the origin is allowed.

```
options.AddPolicy("MyAllowSubdomainPolicy",
    builder =>
    {
        builder.WithOrigins("https://*.example.com")
            .SetIsOriginAllowedToAllowWildcardSubdomains();
    });
```

Set the allowed HTTP methods

AllowAnyMethod:

- Allows any HTTP method:
- Affects preflight requests and the `Access-Control-Allow-Methods` header. For more information, see the Preflight requests section.

Set the allowed request headers

To allow specific headers to be sent in a CORS request, called author request headers, call WithHeaders and specify the allowed headers:

```
options.AddPolicy("MyAllowHeadersPolicy",
    builder =>
    {
        // requires using Microsoft.Net.Http.Headers;
        builder.WithOrigins("http://example.com")
               .WithHeaders(HeaderNames.ContentType, "x-custom-header");
    });
```

To allow all author request headers, call AllowAnyHeader:

```
options.AddPolicy("MyAllowAllHeadersPolicy",
    builder =>
    {
        builder.WithOrigins("https://*.example.com")
               .AllowAnyHeader();
    });
```

`AllowAnyHeader` affects preflight requests and the Access-Control-Request-Headers header. For more information, see the Preflight requests section.

A CORS Middleware policy match to specific headers specified by `WithHeaders` is only possible when the headers sent in `Access-Control-Request-Headers` exactly match the headers stated in `WithHeaders`.

For instance, consider an app configured as follows:

```
app.UseCors(policy => policy.WithHeaders(HeaderNames.CacheControl));
```

CORS Middleware declines a preflight request with the following request header because `Content-Language` (HeaderNames.ContentLanguage) isn't listed in `WithHeaders`:

```
Access-Control-Request-Headers: Cache-Control, Content-Language
```

The app returns a *200 OK* response but doesn't send the CORS headers back. Therefore, the browser doesn't attempt the cross-origin request.

Set the exposed response headers

By default, the browser doesn't expose all of the response headers to the app. For more information, see W3C Cross-Origin Resource Sharing (Terminology): Simple Response Header.

The response headers that are available by default are:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

The CORS specification calls these headers *simple response headers*. To make other headers available to the app, call WithExposedHeaders:

```
options.AddPolicy("MyExposeResponseHeadersPolicy",
    builder =>
    {
        builder.WithOrigins("https://*.example.com")
               .WithExposedHeaders("x-custom-header");
    });
```

Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser doesn't send credentials with a cross-origin request. Credentials include cookies and HTTP authentication schemes. To send credentials with a cross-origin request, the client must set `XMLHttpRequest.withCredentials` to `true`.

Using `XMLHttpRequest` directly:

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'https://www.example.com/api/test');
xhr.withCredentials = true;
```

Using jQuery:

```
$.ajax({
  type: 'get',
  url: 'https://www.example.com/api/test',
  xhrFields: {
    withCredentials: true
  }
});
```

Using the Fetch API:

```
fetch('https://www.example.com/api/test', {
    credentials: 'include'
});
```

The server must allow the credentials. To allow cross-origin credentials, call AllowCredentials:

```
options.AddPolicy("MyMyAllowCredentialsPolicy",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowCredentials();
    });
```

The HTTP response includes an `Access-Control-Allow-Credentials` header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials but the response doesn't include a valid `Access-Control-Allow-Credentials` header, the browser doesn't expose the response to the app, and the cross-origin request fails.

Allowing cross-origin credentials is a security risk. A website at another domain can send a signed-in user's credentials to the app on the user's behalf without the user's knowledge.

The CORS specification also states that setting origins to `"*"` (all origins) is invalid if the `Access-Control-Allow-Credentials` header is present.

Preflight requests

For some CORS requests, the browser sends an additional OPTIONS request before making the actual request. This request is called a preflight request. The browser can skip the preflight request if **all** the following conditions are true:

- The request method is GET, HEAD, or POST.
- The app doesn't set request headers other than `Accept`, `Accept-Language`, `Content-Language`, `Content-Type`, or `Last-Event-ID`.
- The `Content-Type` header, if set, has one of the following values:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`

The rule on request headers set for the client request applies to headers that the app sets by calling `setRequestHeader` on the `XMLHttpRequest` object. The CORS specification calls these headers author request headers. The rule doesn't apply to headers the browser can set, such as `User-Agent`, `Host`, or `Content-Length`.

The following is an example response similar to the preflight request made from the **[Put test]** button in the Test CORS section of this document.

```
General:
Request URL: https://cors3.azurewebsites.net/api/values/5
Request Method: OPTIONS
Status Code: 204 No Content

Response Headers:
Access-Control-Allow-Methods: PUT,DELETE,GET
Access-Control-Allow-Origin: https://cors1.azurewebsites.net
Server: Microsoft-IIS/10.0
Set-Cookie: ARRAffinity=8f8...8;Path=/;HttpOnly;Domain=cors1.azurewebsites.net
Vary: Origin

Request Headers:
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Access-Control-Request-Method: PUT
Connection: keep-alive
Host: cors3.azurewebsites.net
Origin: https://cors1.azurewebsites.net
Referer: https://cors1.azurewebsites.net/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0
```

The preflight request uses the HTTP OPTIONS method. It may include the following headers:

- Access-Control-Request-Method: The HTTP method that will be used for the actual request.
- Access-Control-Request-Headers: A list of request headers that the app sets on the actual request. As stated earlier, this doesn't include headers that the browser sets, such as `User-Agent`.
- Access-Control-Allow-Methods

If the preflight request is denied, the app returns a `200 OK` response but doesn't set the CORS headers. Therefore, the browser doesn't attempt the cross-origin request. For an example of a denied preflight request, see the Test CORS section of this document.

Using the F12 tools, the console app shows an error similar to one of the following, depending on the browser:

- Firefox: Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at `https://cors1.azurewebsites.net/api/TodoItems1/MyDelete2/5`. (Reason: CORS request did not succeed). Learn More
- Chromium based: Access to fetch at 'https://cors1.azurewebsites.net/api/TodoItems1/MyDelete2/5' from origin 'https://cors3.azurewebsites.net' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

To allow specific headers, call WithHeaders:

```
options.AddPolicy("MyAllowHeadersPolicy",
    builder =>
    {
        // requires using Microsoft.Net.Http.Headers;
        builder.WithOrigins("http://example.com")
               .WithHeaders(HeaderNames.ContentType, "x-custom-header");
    });
```

To allow all author request headers, call AllowAnyHeader:

```
options.AddPolicy("MyAllowAllHeadersPolicy",
    builder =>
    {
        builder.WithOrigins("https://*.example.com")
               .AllowAnyHeader();
    });
```

Browsers aren't consistent in how they set `Access-Control-Request-Headers`. If either:

- Headers are set to anything other than `"*"`
- AllowAnyHeader is called: Include at least `Accept`, `Content-Type`, and `Origin`, plus any custom headers that you want to support.

Automatic preflight request code

When the CORS policy is applied either:

- Globally by calling `app.UseCors` in `Startup.Configure`.
- Using the `[EnableCors]` attribute.

ASP.NET Core responds to the preflight OPTIONS request.

Enabling CORS on a per-endpoint basis using `RequireCors` currently does **not** support automatic preflight requests.

The Test CORS section of this document demonstrates this behavior.

[HttpOptions] attribute for preflight requests

When CORS is enabled with the appropriate policy, ASP.NET Core generally responds to CORS preflight requests automatically. In some scenarios, this may not be the case. For example, using CORS with endpoint routing.

The following code uses the [HttpOptions] attribute to create endpoints for OPTIONS requests:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItems2Controller : ControllerBase
{
    // OPTIONS: api/TodoItems2/5
    [HttpOptions("{id}")]
    public IActionResult PreflightRoute(int id)
    {
        return NoContent();
    }

    // OPTIONS: api/TodoItems2
    [HttpOptions]
    public IActionResult PreflightRoute()
    {
        return NoContent();
    }

    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id)
    {
        if (id < 1)
        {
            return BadRequest();
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }
```

See Test CORS with endpoint routing and [HttpOptions] for instructions on testing the preceding code.

Set the preflight expiration time

The `Access-Control-Max-Age` header specifies how long the response to the preflight request can be cached. To set this header, call SetPreflightMaxAge:

```
options.AddPolicy("MySetPreflightExpirationPolicy",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
    });
```

How CORS works

This section describes what happens in a CORS request at the level of the HTTP messages.

- CORS is **not** a security feature. CORS is a W3C standard that allows a server to relax the same-origin policy.
  - For example, a malicious actor could use Cross-Site Scripting (XSS) against your site and execute a cross-site request to their CORS enabled site to steal information.
- An API isn't safer by allowing CORS.
  - It's up to the client (browser) to enforce CORS. The server executes the request and returns the response, it's the client that returns an error and blocks the response. For example, any of the following tools will display the server response:
    - Fiddler
    - Postman
    - .NET HttpClient
    - A web browser by entering the URL in the address bar.
- It's a way for a server to allow browsers to execute a cross-origin XHR or Fetch API request that otherwise

would be forbidden.

- Browsers without CORS can't do cross-origin requests. Before CORS, JSONP was used to circumvent this restriction. JSONP doesn't use XHR, it uses the `<script>` tag to receive the response. Scripts are allowed to be loaded cross-origin.

The CORS specification introduced several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests. Custom JavaScript code isn't required to enable CORS.

The PUT test button on the deployed sample

The following is an example of a cross-origin request from the Values test button to `https://cors1.azurewebsites.net/api/values`. The `Origin` header:

- Provides the domain of the site that's making the request.
- Is required and must be different from the host.

General headers

```
Request URL: https://cors1.azurewebsites.net/api/values
Request Method: GET
Status Code: 200 OK
```

Response headers

```
Content-Encoding: gzip
Content-Type: text/plain; charset=utf-8
Server: Microsoft-IIS/10.0
Set-Cookie: ARRAffinity=8f...;Path=/;HttpOnly;Domain=cors1.azurewebsites.net
Transfer-Encoding: chunked
Vary: Accept-Encoding
X-Powered-By: ASP.NET
```

Request headers

```
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
Host: cors1.azurewebsites.net
Origin: https://cors3.azurewebsites.net
Referer: https://cors3.azurewebsites.net/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0 ...
```

In `OPTIONS` requests, the server sets the **Response headers** `Access-Control-Allow-Origin: {allowed origin}` header in the response. For example, the deployed sample, Delete [EnableCors] button `OPTIONS` request contains the following headers:

General headers

```
Request URL: https://cors3.azurewebsites.net/api/TodoItems2/MyDelete2/5
Request Method: OPTIONS
Status Code: 204 No Content
```

Response headers

```
Access-Control-Allow-Headers: Content-Type,x-custom-header
Access-Control-Allow-Methods: PUT,DELETE,GET,OPTIONS
Access-Control-Allow-Origin: https://cors1.azurewebsites.net
Server: Microsoft-IIS/10.0
Set-Cookie: ARRAffinity=8f...;Path=/;HttpOnly;Domain=cors3.azurewebsites.net
Vary: Origin
X-Powered-By: ASP.NET
```

Request headers

```
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Access-Control-Request-Headers: content-type
Access-Control-Request-Method: DELETE
Connection: keep-alive
Host: cors3.azurewebsites.net
Origin: https://cors1.azurewebsites.net
Referer: https://cors1.azurewebsites.net/test?number=2
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0
```

In the preceding **Response headers**, the server sets the Access-Control-Allow-Origin header in the response. The `https://cors1.azurewebsites.net` value of this header matches the `Origin` header from the request.

If AllowAnyOrigin is called, the `Access-Control-Allow-Origin: *`, the wildcard value, is returned. `AllowAnyOrigin` allows any origin.

If the response doesn't include the `Access-Control-Allow-Origin` header, the cross-origin request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser doesn't make the response available to the client app.

Display OPTIONS requests

By default, the Chrome and Edge browsers don't show OPTIONS requests on the network tab of the F12 tools. To display OPTIONS requests in these browsers:

- `chrome://flags/#out-of-blink-cors` or `edge://flags/#out-of-blink-cors`
- disable the flag.
- restart.

Firefox shows OPTIONS requests by default.

CORS in IIS

When deploying to IIS, CORS has to run before Windows Authentication if the server isn't configured to allow anonymous access. To support this scenario, the IIS CORS module needs to be installed and configured for the app.

Test CORS

The sample download has code to test CORS. See how to download. The sample is an API project with Razor Pages added:

```
public class StartupTest2
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(name: "MyPolicy",
                builder =>
                {
                    builder.WithOrigins("http://example.com",
                        "http://www.contoso.com",
                        "https://cors1.azurewebsites.net",
                        "https://cors3.azurewebsites.net",
                        "https://localhost:44398",
                        "https://localhost:5001")
                            .WithMethods("PUT", "DELETE", "GET");
                });
        });

        services.AddControllers();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseCors();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
            endpoints.MapRazorPages();
        });
    }
}
```

WARNING

`WithOrigins("https://localhost:<port>");` should only be used for testing a sample app similar to the download sample code.

The following `ValuesController` provides the endpoints for testing:

```
[EnableCors("MyPolicy")]
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public IActionResult Get() =>
        ControllerContext.MyDisplayRouteInfo();

    // GET api/values/5
    [HttpGet("{id}")]
    public IActionResult Get(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // PUT api/values/5
    [HttpPut("{id}")]
    public IActionResult Put(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);


    // GET: api/values/GetValues2
    [DisableCors]
    [HttpGet("{action}")]
    public IActionResult GetValues2() =>
        ControllerContext.MyDisplayRouteInfo();

}
```

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

Test the preceding sample code by using one of the following approaches:

- Use the deployed sample app at https://cors3.azurewebsites.net/. There is no need to download the sample.
- Run the sample with `dotnet run` using the default URL of `https://localhost:5001`.
- Run the sample from Visual Studio with the port set to 44398 for a URL of `https://localhost:44398`.

Using a browser with the F12 tools:

- Select the **Values** button and review the headers in the **Network** tab.
- Select the **PUT test** button. See Display OPTIONS requests for instructions on displaying the OPTIONS request. The **PUT test** creates two requests, an OPTIONS preflight request and the PUT request.
- Select the `GetValues2 [DisableCors]` button to trigger a failed CORS request. As mentioned in the document, the response returns 200 success, but the CORS request is not made. Select the **Console** tab to see the CORS error. Depending on the browser, an error similar to the following is displayed:

  Access to fetch at `'https://cors1.azurewebsites.net/api/values/GetValues2'` from origin `'https://cors3.azurewebsites.net'` has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

CORS-enabled endpoints can be tested with a tool, such as curl, Fiddler, or Postman. When using a tool, the origin of the request specified by the `Origin` header must differ from the host receiving the request. If the request isn't *cross-origin* based on the value of the `Origin` header:

- There's no need for CORS Middleware to process the request.
- CORS headers aren't returned in the response.

The following command uses `curl` to issue an OPTIONS request with information:

```
curl -X OPTIONS https://cors3.azurewebsites.net/api/TodoItems2/5 -i
```

Test CORS with endpoint routing and [HttpOptions]

Enabling CORS on a per-endpoint basis using `RequireCors` currently does **not** support automatic preflight requests. Consider the following code which uses endpoint routing to enable CORS:

```
public class StartupEndPointBugTest
{
    readonly string MyPolicy = "_myPolicy";

    // .WithHeaders(HeaderNames.ContentType, "x-custom-header")
    // forces browsers to require a preflight request with GET

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(name: MyPolicy,
                builder =>
                {
                    builder.WithOrigins("http://example.com",
                                        "http://www.contoso.com",
                                        "https://cors1.azurewebsites.net",
                                        "https://cors3.azurewebsites.net",
                                        "https://localhost:44398",
                                        "https://localhost:5001")
                           .WithHeaders(HeaderNames.ContentType, "x-custom-header")
                           .WithMethods("PUT", "DELETE", "GET", "OPTIONS");
                });
        });

        services.AddControllers();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseCors();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers().RequireCors(MyPolicy);
            endpoints.MapRazorPages();
        });
    }
}
```

The following `TodoItems1Controller` provides endpoints for testing:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItems1Controller : ControllerBase
{
    // PUT: api/TodoItems1/5
    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id)
    {
        if (id < 1)
        {
            return Content($"ID = {id}");
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }

    // Delete: api/TodoItems1/5
    [HttpDelete("{id}")]
    public IActionResult MyDelete(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // GET: api/TodoItems1
    [HttpGet]
    public IActionResult GetTodoItems() =>
        ControllerContext.MyDisplayRouteInfo();

    [EnableCors]
    [HttpGet("{action}")]
    public IActionResult GetTodoItems2() =>
        ControllerContext.MyDisplayRouteInfo();

    // Delete: api/TodoItems1/MyDelete2/5
    [EnableCors]
    [HttpDelete("{action}/{id}")]
    public IActionResult MyDelete2(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);
}
```

Test the preceding code from the test page of the deployed sample.

The **Delete [EnableCors]** and **GET [EnableCors]** buttons succeed, because the endpoints have `[EnableCors]` and respond to preflight requests. The other endpoints fails. The **GET** button fails, because the JavaScript sends:

```
 headers: {
      "Content-Type": "x-custom-header"
 },
```

The following `TodoItems2Controller` provides similar endpoints, but includes explicit code to respond to OPTIONS requests:

```
[Route("api/[controller]")]
[ApiController]
public class TodoItems2Controller : ControllerBase
{
    // OPTIONS: api/TodoItems2/5
    [HttpOptions("{id}")]
    public IActionResult PreflightRoute(int id)
    {
        return NoContent();
    }

    // OPTIONS: api/TodoItems2
    [HttpOptions]
    public IActionResult PreflightRoute()
    {
        return NoContent();
    }

    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id)
    {
        if (id < 1)
        {
            return BadRequest();
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }

    // [EnableCors] // Not needed as OPTIONS path provided
    [HttpDelete("{id}")]
    public IActionResult MyDelete(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    [EnableCors]  // Rquired for this path
    [HttpGet]
    public IActionResult GetTodoItems() =>
        ControllerContext.MyDisplayRouteInfo();

    [HttpGet("{action}")]
    public IActionResult GetTodoItems2() =>
        ControllerContext.MyDisplayRouteInfo();

    [EnableCors]  // Rquired for this path
    [HttpDelete("{action}/{id}")]
    public IActionResult MyDelete2(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);
}
```

Test the preceding code from the test page of the deployed sample. In the **Controller** drop down list, select **Preflight** and then **Set Controller**. All the CORS calls to the `TodoItems2Controller` endpoints succeed.

Additional resources

- Cross-Origin Resource Sharing (CORS)
- Getting started with the IIS CORS module

By Rick Anderson

This article shows how to enable CORS in an ASP.NET Core app.

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction is called the *same-origin policy*. The same-origin policy prevents a malicious site from reading sensitive data from another site. Sometimes, you might want to allow other sites make cross-

origin requests to your app. For more information, see the Mozilla CORS article.

Cross Origin Resource Sharing (CORS):

- Is a W3C standard that allows a server to relax the same-origin policy.
- Is **not** a security feature, CORS relaxes security. An API is not safer by allowing CORS. For more information, see How CORS works.
- Allows a server to explicitly allow some cross-origin requests while rejecting others.
- Is safer and more flexible than earlier techniques, such as JSONP.

View or download sample code (how to download)

Same origin

Two URLs have the same origin if they have identical schemes, hosts, and ports (RFC 6454).

These two URLs have the same origin:

- `https://example.com/foo.html`
- `https://example.com/bar.html`

These URLs have different origins than the previous two URLs:

- `https://example.net`: Different domain
- `https://www.example.com/foo.html`: Different subdomain
- `http://example.com/foo.html`: Different scheme
- `https://example.com:9000/foo.html`: Different port

Internet Explorer doesn't consider the port when comparing origins.

CORS with named policy and middleware

CORS Middleware handles cross-origin requests. The following code enables CORS for the entire app with the specified origin:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    readonly string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddPolicy(MyAllowSpecificOrigins,
            builder =>
            {
                builder.WithOrigins("http://example.com",
                                    "http://www.contoso.com");
            });
        });

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseHsts();
        }

        app.UseCors(MyAllowSpecificOrigins);

        app.UseHttpsRedirection();
        app.UseMvc();
    }
}
```

The preceding code:

- Sets the policy name to "_myAllowSpecificOrigins". The policy name is arbitrary.
- Calls the UseCors extension method, which enables CORS.
- Calls AddCors with a lambda expression. The lambda takes a CorsPolicyBuilder object. Configuration options, such as `WithOrigins`, are described later in this article.

The AddCors method call adds CORS services to the app's service container:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy(MyAllowSpecificOrigins,
        builder =>
        {
            builder.WithOrigins("http://example.com",
                                "http://www.contoso.com");
        });
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

For more information, see CORS policy options in this document .

The CorsPolicyBuilder method can chain methods, as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy(MyAllowSpecificOrigins,
        builder =>
        {
            builder.WithOrigins("http://example.com",
                                "http://www.contoso.com")
                                .AllowAnyHeader()
                                .AllowAnyMethod();
        });
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Note: The URL must **not** contain a trailing slash (`/`). If the URL terminates with `/`, the comparison returns `false` and no header is returned.

The following code applies CORS policies to all the apps endpoints via CORS Middleware:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.UseCors();

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

Note: `UseCors` must be called before `UseMvc`.

See Enable CORS in Razor Pages, controllers, and action methods to apply CORS policy at the page/controller/action level.

See Test CORS for instructions on testing code similar to the preceding code.

Enable CORS with attributes

The [EnableCors] attribute provides an alternative to applying CORS globally. The `[EnableCors]` attribute enables CORS for selected end points, rather than all end points.

Use `[EnableCors]` to specify the default policy and `[EnableCors("{Policy String}")]` to specify a policy.

The `[EnableCors]` attribute can be applied to:

- Razor Page `PageModel`
- Controller
- Controller action method

You can apply different policies to controller/page-model/action with the `[EnableCors]` attribute. When the `[EnableCors]` attribute is applied to a controllers/page model/action method, and CORS is enabled in middleware, **both** policies are applied. We recommend **not** combining policies. Use the `[EnableCors]` attribute or middleware, **not both**. When using `[EnableCors]`, do **not** define a default policy.

The following code applies a different policy to each method:

```
[Route("api/[controller]")]
[ApiController]
public class WidgetController : ControllerBase
{
    // GET api/values
    [EnableCors("AnotherPolicy")]
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "green widget", "red widget" };
    }

    // GET api/values/5
    [EnableCors]        // Default policy.
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    {
        switch (id)
        {
            case 1:
                return "green widget";
            case 2:
                return "red widget";
            default:
                return NotFound();
        }
    }
}
```

The following code creates a CORS default policy and a policy named `"AnotherPolicy"`:

```
public class StartupMultiPolicy
{
    public StartupMultiPolicy(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddCors(options =>
        {
            options.AddDefaultPolicy(
                builder =>
                {

                    builder.WithOrigins("http://example.com",
                                        "http://www.contoso.com");
                });

            options.AddPolicy("AnotherPolicy",
                builder =>
                {
                    builder.WithOrigins("http://www.contoso.com")
                                        .AllowAnyHeader()
                                        .AllowAnyMethod();
                });

        });

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseMvc();
    }
}
```

Disable CORS

The [DisableCors] attribute disables CORS for the controller/page-model/action.

CORS policy options

This section describes the various options that can be set in a CORS policy:

- Set the allowed origins
- Set the allowed HTTP methods
- Set the allowed request headers
- Set the exposed response headers
- Credentials in cross-origin requests

- Set the preflight expiration time

AddPolicy is called in `Startup.ConfigureServices`. For some options, it may be helpful to read the How CORS works section first.

Set the allowed origins

AllowAnyOrigin: Allows CORS requests from all origins with any scheme (`http` or `https`). `AllowAnyOrigin` is insecure because *any website* can make cross-origin requests to the app.

> **NOTE**
> Specifying `AllowAnyOrigin` and `AllowCredentials` is an insecure configuration and can result in cross-site request forgery. For a secure app, specify an exact list of origins if the client must authorize itself to access server resources.

`AllowAnyOrigin` affects preflight requests and the `Access-Control-Allow-Origin` header. For more information, see the Preflight requests section.

SetIsOriginAllowedToAllowWildcardSubdomains: Sets the IsOriginAllowed property of the policy to be a function that allows origins to match a configured wildcard domain when evaluating if the origin is allowed.

```
options.AddPolicy("AllowSubdomain",
    builder =>
    {
        builder.WithOrigins("https://*.example.com")
            .SetIsOriginAllowedToAllowWildcardSubdomains();
    });
```

Set the allowed HTTP methods

AllowAnyMethod:

- Allows any HTTP method:
- Affects preflight requests and the `Access-Control-Allow-Methods` header. For more information, see the Preflight requests section.

Set the allowed request headers

To allow specific headers to be sent in a CORS request, called *author request headers*, call WithHeaders and specify the allowed headers:

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithHeaders(HeaderNames.ContentType, "x-custom-header");
    });
```

To allow all author request headers, call AllowAnyHeader:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyHeader();
    });
```

This setting affects preflight requests and the `Access-Control-Request-Headers` header. For more information, see the Preflight requests section.

CORS Middleware always allows four headers in the `Access-Control-Request-Headers` to be sent regardless of the values configured in CorsPolicy.Headers. This list of headers includes:

- `Accept`
- `Accept-Language`
- `Content-Language`
- `Origin`

For instance, consider an app configured as follows:

```
app.UseCors(policy => policy.WithHeaders(HeaderNames.CacheControl));
```

CORS Middleware responds successfully to a preflight request with the following request header because `Content-Language` is always permitted:

```
Access-Control-Request-Headers: Cache-Control, Content-Language
```

Set the exposed response headers

By default, the browser doesn't expose all of the response headers to the app. For more information, see W3C Cross-Origin Resource Sharing (Terminology): Simple Response Header.

The response headers that are available by default are:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

The CORS specification calls these headers *simple response headers*. To make other headers available to the app, call WithExposedHeaders:

```
options.AddPolicy("ExposeResponseHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithExposedHeaders("x-custom-header");
    });
```

Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser doesn't send credentials with a cross-origin request. Credentials include cookies and HTTP authentication schemes. To send credentials with a cross-origin request, the client must set `XMLHttpRequest.withCredentials` to `true`.

Using `XMLHttpRequest` directly:

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'https://www.example.com/api/test');
xhr.withCredentials = true;
```

Using jQuery:

```
$.ajax({
  type: 'get',
  url: 'https://www.example.com/api/test',
  xhrFields: {
    withCredentials: true
  }
});
```

Using the Fetch API:

```
fetch('https://www.example.com/api/test', {
    credentials: 'include'
});
```

The server must allow the credentials. To allow cross-origin credentials, call AllowCredentials:

```
options.AddPolicy("AllowCredentials",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowCredentials();
    });
```

The HTTP response includes an `Access-Control-Allow-Credentials` header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials but the response doesn't include a valid `Access-Control-Allow-Credentials` header, the browser doesn't expose the response to the app, and the cross-origin request fails.

Allowing cross-origin credentials is a security risk. A website at another domain can send a signed-in user's credentials to the app on the user's behalf without the user's knowledge.

The CORS specification also states that setting origins to `"*"` (all origins) is invalid if the `Access-Control-Allow-Credentials` header is present.

Preflight requests

For some CORS requests, the browser sends an additional request before making the actual request. This request is called a *preflight request*. The browser can skip the preflight request if the following conditions are true:

- The request method is GET, HEAD, or POST.
- The app doesn't set request headers other than `Accept`, `Accept-Language`, `Content-Language`, `Content-Type`, or `Last-Event-ID`.
- The `Content-Type` header, if set, has one of the following values:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`

The rule on request headers set for the client request applies to headers that the app sets by calling

`setRequestHeader` on the `XMLHttpRequest` object. The CORS specification calls these headers *author request headers*. The rule doesn't apply to headers the browser can set, such as `User-Agent`, `Host`, or `Content-Length`.

The following is an example of a preflight request:

```
OPTIONS https://myservice.azurewebsites.net/api/test HTTP/1.1
Accept: */*
Origin: https://myclient.azurewebsites.net
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: accept, x-my-custom-header
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
Content-Length: 0
```

The pre-flight request uses the HTTP OPTIONS method. It includes two special headers:

- `Access-Control-Request-Method`: The HTTP method that will be used for the actual request.
- `Access-Control-Request-Headers`: A list of request headers that the app sets on the actual request. As stated earlier, this doesn't include headers that the browser sets, such as `User-Agent`.

When CORS is enabled with the appropriate policy, ASP.NET Core generally automatically responds to CORS preflight requests. See [HttpOptions] attribute for preflight requests.

A CORS preflight request might include an `Access-Control-Request-Headers` header, which indicates to the server the headers that are sent with the actual request.

To allow specific headers, call WithHeaders:

```
options.AddPolicy("AllowHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .WithHeaders(HeaderNames.ContentType, "x-custom-header");
    });
```

To allow all author request headers, call AllowAnyHeader:

```
options.AddPolicy("AllowAllHeaders",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .AllowAnyHeader();
    });
```

Browsers aren't entirely consistent in how they set `Access-Control-Request-Headers`. If you set headers to anything other than `"*"` (or use AllowAnyHeader), you should include at least `Accept`, `Content-Type`, and `Origin`, plus any custom headers that you want to support.

The following is an example response to the preflight request (assuming that the server allows the request):

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 0
Access-Control-Allow-Origin: https://myclient.azurewebsites.net
Access-Control-Allow-Headers: x-my-custom-header
Access-Control-Allow-Methods: PUT
Date: Wed, 20 May 2015 06:33:22 GMT
```

The response includes an `Access-Control-Allow-Methods` header that lists the allowed methods and optionally an `Access-Control-Allow-Headers` header, which lists the allowed headers. If the preflight request succeeds, the browser sends the actual request.

If the preflight request is denied, the app returns a *200 OK* response but doesn't send the CORS headers back. Therefore, the browser doesn't attempt the cross-origin request.

Set the preflight expiration time

The `Access-Control-Max-Age` header specifies how long the response to the preflight request can be cached. To set this header, call SetPreflightMaxAge:

```
options.AddPolicy("SetPreflightExpiration",
    builder =>
    {
        builder.WithOrigins("http://example.com")
               .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
    });
```

How CORS works

This section describes what happens in a CORS request at the level of the HTTP messages.

- CORS is **not** a security feature. CORS is a W3C standard that allows a server to relax the same-origin policy.
  - For example, a malicious actor could use Prevent Cross-Site Scripting (XSS) against your site and execute a cross-site request to their CORS enabled site to steal information.
- Your API is not safer by allowing CORS.
  - It's up to the client (browser) to enforce CORS. The server executes the request and returns the response, it's the client that returns an error and blocks the response. For example, any of the following tools will display the server response:
    - Fiddler
    - Postman
    - .NET HttpClient
    - A web browser by entering the URL in the address bar.
- It's a way for a server to allow browsers to execute a cross-origin XHR or Fetch API request that otherwise would be forbidden.
  - Browsers (without CORS) can't do cross-origin requests. Before CORS, JSONP was used to circumvent this restriction. JSONP doesn't use XHR, it uses the `<script>` tag to receive the response. Scripts are allowed to be loaded cross-origin.

The CORS specification introduced several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests. Custom JavaScript code isn't required to enable CORS.

The following is an example of a cross-origin request. The `Origin` header provides the domain of the site that's making the request. The `Origin` header is required and must be different from the host.

```
GET https://myservice.azurewebsites.net/api/test HTTP/1.1
Referer: https://myclient.azurewebsites.net/
Accept: */*
Accept-Language: en-US
Origin: https://myclient.azurewebsites.net
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)
Host: myservice.azurewebsites.net
```

If the server allows the request, it sets the `Access-Control-Allow-Origin` header in the response. The value of this header either matches the `Origin` header from the request or is the wildcard value `"*"`, meaning that any origin is allowed:

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: https://myclient.azurewebsites.net
Date: Wed, 20 May 2015 06:27:30 GMT
Content-Length: 12

Test message
```

If the response doesn't include the `Access-Control-Allow-Origin` header, the cross-origin request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser doesn't make the response available to the client app.

Test CORS

To test CORS:

1. Create an API project. Alternatively, you can download the sample.
2. Enable CORS using one of the approaches in this document. For example:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    // Shows UseCors with CorsPolicyBuilder.
    app.UseCors(builder =>
    {
        builder.WithOrigins("http://example.com",
                            "http://www.contoso.com",
                            "https://localhost:44375",
                            "https://localhost:5001");
    });

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

WARNING

`WithOrigins("https://localhost:<port>");` should only be used for testing a sample app similar to the download sample code.

1. Create a web app project (Razor Pages or MVC). The sample uses Razor Pages. You can create the web app in the same solution as the API project.
2. Add the following highlighted code to the *Index.cshtml* file:

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">CORS Test</h1>
</div>

<div>
    <input type="button" value="Test"
           onclick="requestVal('https://<web app>.azurewebsites.net/api/values')" />
    <span id='result'></span>
</div>

<script>
    function requestVal(uri) {
        const resultSpan = document.getElementById('result');

        fetch(uri)
            .then(response => response.json())
            .then(data => resultSpan.innerText = data)
            .catch(error => resultSpan.innerText = 'See F12 Console for error');
    }
</script>
```

1. In the preceding code, replace `url: 'https://<web app>.azurewebsites.net/api/values/1',` with the URL to the deployed app.
2. Deploy the API project. For example, deploy to Azure.
3. Run the Razor Pages or MVC app from the desktop and click on the **Test** button. Use the F12 tools to review error messages.
4. Remove the localhost origin from `WithOrigins` and deploy the app. Alternatively, run the client app with a different port. For example, run from Visual Studio.
5. Test with the client app. CORS failures return an error, but the error message isn't available to JavaScript. Use the console tab in the F12 tools to see the error. Depending on the browser, you get an error (in the F12 tools console) similar to the following:

   - Using Microsoft Edge:

     SEC7120: [CORS] The origin `https://localhost:44375` did not find `https://localhost:44375` in the Access-Control-Allow-Origin response header for cross-origin resource at `https://webapi.azurewebsites.net/api/values/1`

   - Using Chrome:

     Access to XMLHttpRequest at `https://webapi.azurewebsites.net/api/values/1` from origin

> `https://localhost:44375` **has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.**

CORS-enabled endpoints can be tested with a tool, such as Fiddler or Postman. When using a tool, the origin of the request specified by the `Origin` header must differ from the host receiving the request. If the request isn't *cross-origin* based on the value of the `Origin` header:

- There's no need for CORS Middleware to process the request.
- CORS headers aren't returned in the response.

CORS in IIS

When deploying to IIS, CORS has to run before Windows Authentication if the server isn't configured to allow anonymous access. To support this scenario, the IIS CORS module needs to be installed and configured for the app.

Additional resources

- Cross-Origin Resource Sharing (CORS)
- Getting started with the IIS CORS module

Share authentication cookies among ASP.NET apps

9/22/2020 • 5 minutes to read • Edit Online

By Rick Anderson

Websites often consist of individual web apps working together. To provide a single sign-on (SSO) experience, web apps within a site must share authentication cookies. To support this scenario, the data protection stack allows sharing Katana cookie authentication and ASP.NET Core cookie authentication tickets.

In the examples that follow:

- The authentication cookie name is set to a common value of `.AspNet.SharedCookie`.
- The `AuthenticationType` is set to `Identity.Application` either explicitly or by default.
- A common app name is used to enable the data protection system to share data protection keys (`SharedCookieApp`).
- `Identity.Application` is used as the authentication scheme. Whatever scheme is used, it must be used consistently *within and across* the shared cookie apps either as the default scheme or by explicitly setting it. The scheme is used when encrypting and decrypting cookies, so a consistent scheme must be used across apps.
- A common data protection key storage location is used.
  - In ASP.NET Core apps, PersistKeysToFileSystem is used to set the key storage location.
  - In .NET Framework apps, Cookie Authentication Middleware uses an implementation of DataProtectionProvider. `DataProtectionProvider` provides data protection services for the encryption and decryption of authentication cookie payload data. The `DataProtectionProvider` instance is isolated from the data protection system used by other parts of the app. DataProtectionProvider.Create(System.IO.DirectoryInfo, Action<IDataProtectionBuilder>) accepts a DirectoryInfo to specify the location for data protection key storage.
- `DataProtectionProvider` requires the Microsoft.AspNetCore.DataProtection.Extensions NuGet package:
  - In ASP.NET Core 2.x apps, reference the Microsoft.AspNetCore.App metapackage.
  - In .NET Framework apps, add a package reference to Microsoft.AspNetCore.DataProtection.Extensions.
- SetApplicationName sets the common app name.

Share authentication cookies with ASP.NET Core Identity

When using ASP.NET Core Identity:

- Data protection keys and the app name must be shared among apps. A common key storage location is provided to the PersistKeysToFileSystem method in the following examples. Use SetApplicationName to configure a common shared app name (`SharedCookieApp` in the following examples). For more information, see Configure ASP.NET Core Data Protection.
- Use the ConfigureApplicationCookie extension method to set up the data protection service for cookies.
- The default authentication type is `Identity.Application`.

In `Startup.ConfigureServices`:

```
services.AddDataProtection()
    .PersistKeysToFileSystem("{PATH TO COMMON KEY RING FOLDER}")
    .SetApplicationName("SharedCookieApp");

services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = ".AspNet.SharedCookie";
});
```

Share authentication cookies without ASP.NET Core Identity

When using cookies directly without ASP.NET Core Identity, configure data protection and authentication in `Startup.ConfigureServices`. In the following example, the authentication type is set to `Identity.Application`:

```
services.AddDataProtection()
    .PersistKeysToFileSystem("{PATH TO COMMON KEY RING FOLDER}")
    .SetApplicationName("SharedCookieApp");

services.AddAuthentication("Identity.Application")
    .AddCookie("Identity.Application", options =>
    {
        options.Cookie.Name = ".AspNet.SharedCookie";
    });
```

Share cookies across different base paths

An authentication cookie uses the HttpRequest.PathBase as its default Cookie.Path. If the app's cookie must be shared across different base paths, `Path` must be overridden:

```
services.AddDataProtection()
    .PersistKeysToFileSystem("{PATH TO COMMON KEY RING FOLDER}")
    .SetApplicationName("SharedCookieApp");

services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = ".AspNet.SharedCookie";
    options.Cookie.Path = "/";
});
```

Share cookies across subdomains

When hosting apps that share cookies across subdomains, specify a common domain in the Cookie.Domain property. To share cookies across apps at `contoso.com`, such as `first_subdomain.contoso.com` and `second_subdomain.contoso.com`, specify the `Cookie.Domain` as `.contoso.com`:

```
options.Cookie.Domain = ".contoso.com";
```

Encrypt data protection keys at rest

For production deployments, configure the `DataProtectionProvider` to encrypt keys at rest with DPAPI or an X509Certificate. For more information, see Key encryption at rest in Windows and Azure using ASP.NET Core. In the following example, a certificate thumbprint is provided to ProtectKeysWithCertificate:

```
services.AddDataProtection()
    .ProtectKeysWithCertificate("{CERTIFICATE THUMBPRINT}");
```

Share authentication cookies between ASP.NET 4.x and ASP.NET Core apps

ASP.NET 4.x apps that use Katana Cookie Authentication Middleware can be configured to generate authentication cookies that are compatible with the ASP.NET Core Cookie Authentication Middleware. This allows upgrading a large site's individual apps in several steps while providing a smooth SSO experience across the site.

When an app uses Katana Cookie Authentication Middleware, it calls `UseCookieAuthentication` in the project's *Startup.Auth.cs* file. ASP.NET 4.x web app projects created with Visual Studio 2013 and later use the Katana Cookie Authentication Middleware by default. Although `UseCookieAuthentication` is obsolete and unsupported for ASP.NET Core apps, calling `UseCookieAuthentication` in an ASP.NET 4.x app that uses Katana Cookie Authentication Middleware is valid.

An ASP.NET 4.x app must target .NET Framework 4.5.1 or later. Otherwise, the necessary NuGet packages fail to install.

To share authentication cookies between an ASP.NET 4.x app and an ASP.NET Core app, configure the ASP.NET Core app as stated in the Share authentication cookies among ASP.NET Core apps section, then configure the ASP.NET 4.x app as follows.

Confirm that the app's packages are updated to the latest releases. Install the Microsoft.Owin.Security.Interop package into each ASP.NET 4.x app.

Locate and modify the call to `UseCookieAuthentication`:

- Change the cookie name to match the name used by the ASP.NET Core Cookie Authentication Middleware (`.AspNet.SharedCookie` in the example).
- In the following example, the authentication type is set to `Identity.Application`.
- Provide an instance of a `DataProtectionProvider` initialized to the common data protection key storage location.
- Confirm that the app name is set to the common app name used by all apps that share authentication cookies (`SharedCookieApp` in the example).

If not setting `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier` and `http://schemas.microsoft.com/accesscontrolservice/2010/07/claims/identityprovider`, set UniqueClaimTypeIdentifier to a claim that distinguishes unique users.

App_Start/Startup.Auth.cs:

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    AuthenticationType = "Identity.Application",
    CookieName = ".AspNet.SharedCookie",
    LoginPath = new PathString("/Account/Login"),
    Provider = new CookieAuthenticationProvider
    {
        OnValidateIdentity =
            SecurityStampValidator
                .OnValidateIdentity<ApplicationUserManager, ApplicationUser>(
                    validateInterval: TimeSpan.FromMinutes(30),
                    regenerateIdentity: (manager, user) =>
                        user.GenerateUserIdentityAsync(manager))
    },
    TicketDataFormat = new AspNetTicketDataFormat(
        new DataProtectorShim(
            DataProtectionProvider.Create("{PATH TO COMMON KEY RING FOLDER}",
                (builder) => { builder.SetApplicationName("SharedCookieApp"); })
            .CreateProtector(
                "Microsoft.AspNetCore.Authentication.Cookies." +
                    "CookieAuthenticationMiddleware",
                "Identity.Application",
                "v2"))),
    CookieManager = new ChunkingCookieManager()
});

System.Web.Helpers.AntiForgeryConfig.UniqueClaimTypeIdentifier =
    "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name";
```

When generating a user identity, the authentication type (`Identity.Application`) must match the type defined in `AuthenticationType` set with `UseCookieAuthentication` in *App_Start/Startup.Auth.cs*.

Models/IdentityModels.cs:

```
public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity> GenerateUserIdentityAsync(
        UserManager<ApplicationUser> manager)
    {
        // The authenticationType must match the one defined in
        // CookieAuthenticationOptions.AuthenticationType
        var userIdentity =
            await manager.CreateIdentityAsync(this, "Identity.Application");

        // Add custom user claims here

        return userIdentity;
    }
}
```

Use a common user database

When apps use the same Identity schema (same version of Identity), confirm that the Identity system for each app is pointed at the same user database. Otherwise, the identity system produces failures at runtime when it attempts to match the information in the authentication cookie against the information in its database.

When the Identity schema is different among apps, usually because apps are using different Identity versions, sharing a common database based on the latest version of Identity isn't possible without remapping and adding columns in other app's Identity schemas. It's often more efficient to upgrade the other apps to use the latest Identity version so that a common database can be shared by the apps.

Additional resources

- Host ASP.NET Core in a web farm

Work with SameSite cookies in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

SameSite is an IETF draft standard designed to provide some protection against cross-site request forgery (CSRF) attacks. Originally drafted in 2016, the draft standard was updated in 2019. The updated standard is not backward compatible with the previous standard, with the following being the most noticeable differences:

- Cookies without SameSite header are treated as `SameSite=Lax` by default.
- `SameSite=None` must be used to allow cross-site cookie use.
- Cookies that assert `SameSite=None` must also be marked as `Secure`.
- Applications that use `<iframe>` may experience issues with `sameSite=Lax` or `sameSite=Strict` cookies because `<iframe>` is treated as cross-site scenarios.
- The value `SameSite=None` is not allowed by the 2016 standard and causes some implementations to treat such cookies as `SameSite=Strict`. See Supporting older browsers in this document.

The `SameSite=Lax` setting works for most application cookies. Some forms of authentication like OpenID Connect (OIDC) and WS-Federation default to POST based redirects. The POST based redirects trigger the SameSite browser protections, so SameSite is disabled for these components. Most OAuth logins are not affected due to differences in how the request flows.

Each ASP.NET Core component that emits cookies needs to decide if SameSite is appropriate.

SameSite and Identity

ASP.NET Core Identity is largely unaffected by SameSite cookies except for advanced scenarios like `IFrames` or `OpenIdConnect` integration.

When using `Identity`, do ***not*** add any cookie providers or call `services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)`, `Identity` takes care of that.

SameSite test sample code

The following samples can be downloaded and tested:

SAMPLE	DOCUMENT
.NET Core MVC	ASP.NET Core 2.1 MVC SameSite cookie sample
.NET Core Razor Pages	ASP.NET Core 2.1 Razor Pages SameSite cookie sample

The following sample can be downloaded and tested:

SAMPLE	DOCUMENT
.NET Core Razor Pages	ASP.NET Core 3.1 Razor Pages SameSite cookie sample

.NET Core support for the sameSite attribute

.NET Core 2.2 and later support the 2019 draft standard for SameSite since the release of updates in December 2019. Developers are able to programmatically control the value of the sameSite attribute using the `HttpCookie.SameSite` property. Setting the `SameSite` property to Strict, Lax, or None results in those values being written on the network with the cookie. Setting it equal to `(SameSiteMode)(-1)` indicates that no sameSite attribute should be included on the network with the cookie

```
var cookieOptions = new CookieOptions
{
    // Set the secure flag, which Chrome's changes will require for SameSite none.
    // Note this will also require you to be running on HTTPS.
    Secure = true,

    // Set the cookie to HTTP only which is good practice unless you really do need
    // to access it client side in scripts.
    HttpOnly = true,

    // Add the SameSite attribute, this will emit the attribute with a value of none.
    // To not emit the attribute at all set
    // SameSite = (SameSiteMode)(-1)
    SameSite = SameSiteMode.None
};

// Add the cookie to the response cookie collection
Response.Cookies.Append("MyCookie", "cookieValue", cookieOptions);
```

.NET Core 3.0 and later support the updated SameSite values and adds an extra enum value, `SameSiteMode.Unspecified` to the `SameSiteMode` enum. This new value indicates no sameSite should be sent with the cookie.

December patch behavior changes

The specific behavior change for .NET Framework and .NET Core 2.1 is how the `SameSite` property interprets the `None` value. Before the patch a value of `None` meant "Do not emit the attribute at all", after the patch it means "Emit the attribute with a value of `None`". After the patch a `SameSite` value of `(SameSiteMode)(-1)` causes the attribute not to be emitted.

The default SameSite value for forms authentication and session state cookies was changed from `None` to `Lax`.

API usage with SameSite

HttpContext.Response.Cookies.Append defaults to `Unspecified`, meaning no SameSite attribute added to the cookie and the client will use its default behavior (Lax for new browsers, None for old ones). The following code shows how to change the cookie SameSite value to `SameSiteMode.Lax`:

```
HttpContext.Response.Cookies.Append(
                    "name", "value",
                    new CookieOptions() { SameSite = SameSiteMode.Lax });
```

All ASP.NET Core components that emit cookies override the preceding defaults with settings appropriate for their scenarios. The overridden preceding default values haven't changed.

COMPONENT	COOKIE	DEFAULT
CookieBuilder	SameSite	`Unspecified`
Session	SessionOptions.Cookie	`Lax`

COMPONENT	COOKIE	DEFAULT
CookieTempDataProvider	CookieTempDataProviderOptions.Cookie	`Lax`
IAntiforgery	AntiforgeryOptions.Cookie	`Strict`
Cookie Authentication	CookieAuthenticationOptions.Cookie	`Lax`
AddTwitter	TwitterOptions.StateCookie	`Lax`
RemoteAuthenticationHandler<TOptions>	RemoteAuthenticationOptions.CorrelationCookie	`None`
AddOpenIdConnect	OpenIdConnectOptions.NonceCookie	`None`
HttpContext.Response.Cookies.Append	CookieOptions	`Unspecified`

ASP.NET Core 3.1 and later provides the following SameSite support:

- Redefines the behavior of `SameSiteMode.None` to emit `SameSite=None`
- Adds a new value `SameSiteMode.Unspecified` to omit the SameSite attribute.
- All cookies APIs default to `Unspecified`. Some components that use cookies set values more specific to their scenarios. See the table above for examples.

In ASP.NET Core 3.0 and later the SameSite defaults were changed to avoid conflicting with inconsistent client defaults. The following APIs have changed the default from `SameSiteMode.Lax` to `-1` to avoid emitting a SameSite attribute for these cookies:

- CookieOptions used with HttpContext.Response.Cookies.Append
- CookieBuilder used as a factory for `CookieOptions`
- CookiePolicyOptions.MinimumSameSitePolicy

History and changes

SameSite support was first implemented in ASP.NET Core in 2.0 using the 2016 draft standard. The 2016 standard was opt-in. ASP.NET Core opted-in by setting several cookies to `Lax` by default. After encountering several issues with authentication, most SameSite usage was disabled.

Patches were issued in November 2019 to update from the 2016 standard to the 2019 standard. The 2019 draft of the SameSite specification:

- Is **not** backwards compatible with the 2016 draft. For more information, see Supporting older browsers in this document.
- Specifies cookies are treated as `SameSite=Lax` by default.
- Specifies cookies that explicitly assert `SameSite=None` in order to enable cross-site delivery should be marked as `Secure`. `None` is a new entry to opt out.
- Is supported by patches issued for ASP.NET Core 2.1, 2.2, and 3.0. ASP.NET Core 3.1 has additional SameSite support.
- Is scheduled to be enabled by Chrome by default in Feb 2020. Browsers started moving to this standard in 2019.

APIs impacted by the change from the 2016 SameSite draft standard to the 2019 draft standard

- Http.SameSiteMode
- CookieOptions.SameSite
- CookieBuilder.SameSite
- CookiePolicyOptions.MinimumSameSitePolicy
- Microsoft.Net.Http.Headers.SameSiteMode
- Microsoft.Net.Http.Headers.SetCookieHeaderValue.SameSite

Supporting older browsers

The 2016 SameSite standard mandated that unknown values must be treated as `SameSite=Strict` values. Apps accessed from older browsers which support the 2016 SameSite standard may break when they get a SameSite property with a value of `None`. Web apps must implement browser detection if they intend to support older browsers. ASP.NET Core doesn't implement browser detection because User-Agents values are highly volatile and change frequently. An extension point in Microsoft.AspNetCore.CookiePolicy allows plugging in User-Agent specific logic.

In `Startup.Configure`, add code that calls UseCookiePolicy before calling UseAuthentication or *any* method that writes cookies:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseCookiePolicy();
    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

In `Startup.ConfigureServices`, add code similar to the following:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    });

    services.AddRazorPages();
}

private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.MinimumSameSitePolicy = (SameSiteMode)(-1);
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    });

    services.AddRazorPages();
}

private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = (SameSiteMode)(-1);
        }

    }
}
```

In the preceding sample, `MyUserAgentDetectionLib.DisallowsSameSiteNone` is a user supplied library that detects if the user agent doesn't support SameSite `None`:

```
if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
{
    options.SameSite = SameSiteMode.Unspecified;
}
```

The following code shows a sample `DisallowsSameSiteNone` method:

WARNING

The following code is for demonstration only:

- It should not be considered complete.
- It is not maintained or supported.

```
public static bool DisallowsSameSiteNone(string userAgent)
{
    // Check if a null or empty string has been passed in, since this
    // will cause further interrogation of the useragent to fail.
     if (String.IsNullOrWhiteSpace(userAgent))
        return false;

    // Cover all iOS based browsers here. This includes:
    // - Safari on iOS 12 for iPhone, iPod Touch, iPad
    // - WkWebview on iOS 12 for iPhone, iPod Touch, iPad
    // - Chrome on iOS 12 for iPhone, iPod Touch, iPad
    // All of which are broken by SameSite=None, because they use the iOS networking
    // stack.
    if (userAgent.Contains("CPU iPhone OS 12") ||
        userAgent.Contains("iPad; CPU OS 12"))
    {
        return true;
    }

    // Cover Mac OS X based browsers that use the Mac OS networking stack.
    // This includes:
    // - Safari on Mac OS X.
    // This does not include:
    // - Chrome on Mac OS X
    // Because they do not use the Mac OS networking stack.
    if (userAgent.Contains("Macintosh; Intel Mac OS X 10_14") &&
        userAgent.Contains("Version/") && userAgent.Contains("Safari"))
    {
        return true;
    }

    // Cover Chrome 50-69, because some versions are broken by SameSite=None,
    // and none in this range require it.
    // Note: this covers some pre-Chromium Edge versions,
    // but pre-Chromium Edge does not require SameSite=None.
    if (userAgent.Contains("Chrome/5") || userAgent.Contains("Chrome/6"))
    {
        return true;
    }

    return false;
}
```

Test apps for SameSite problems

Apps that interact with remote sites such as through third-party login need to:

- Test the interaction on multiple browsers.
- Apply the CookiePolicy browser detection and mitigation discussed in this document.

Test web apps using a client version that can opt-in to the new SameSite behavior. Chrome, Firefox, and Chromium Edge all have new opt-in feature flags that can be used for testing. After your app applies the SameSite patches, test it with older client versions, especially Safari. For more information, see Supporting older browsers in this document.

Test with Chrome

Chrome 78+ gives misleading results because it has a temporary mitigation in place. The Chrome 78+ temporary mitigation allows cookies less than two minutes old. Chrome 76 or 77 with the appropriate test flags enabled provides more accurate results. To test the new SameSite behavior toggle `chrome://flags/#same-site-by-default-cookies` to **Enabled**. Older versions of Chrome (75 and below) are reported to fail with the new `None` setting. See Supporting older browsers in this document.

Google does not make older chrome versions available. Follow the instructions at Download Chromium to test older versions of Chrome. Do **not** download Chrome from links provided by searching for older versions of chrome.

- Chromium 76 Win64
- Chromium 74 Win64

Starting in Canary version `80.0.3975.0`, the Lax+POST temporary mitigation can be disabled for testing purposes using the new flag `--enable-features=SameSiteDefaultChecksMethodRigorously` to allow testing of sites and services in the eventual end state of the feature where the mitigation has been removed. For more information, see The Chromium Projects SameSite Updates

Test with Safari

Safari 12 strictly implemented the prior draft and fails when the new `None` value is in a cookie. `None` is avoided via the browser detection code Supporting older browsers in this document. Test Safari 12, Safari 13, and WebKit based OS style logins using MSAL, ADAL or whatever library you are using. The problem is dependent on the underlying OS version. OSX Mojave (10.14) and iOS 12 are known to have compatibility problems with the new SameSite behavior. Upgrading the OS to OSX Catalina (10.15) or iOS 13 fixes the problem. Safari does not currently have an opt-in flag for testing the new spec behavior.

Test with Firefox

Firefox support for the new standard can be tested on version 68+ by opting in on the `about:config` page with the feature flag `network.cookie.sameSite.laxByDefault`. There haven't been reports of compatibility issues with older versions of Firefox.

Test with Edge browser

Edge supports the old SameSite standard. Edge version 44 doesn't have any known compatibility problems with the new standard.

Test with Edge (Chromium)

SameSite flags are set on the `edge://flags/#same-site-by-default-cookies` page. No compatibility issues were discovered with Edge Chromium.

Test with Electron

Versions of Electron include older versions of Chromium. For example, the version of Electron used by Teams is Chromium 66, which exhibits the older behavior. You must perform your own compatibility testing with the version of Electron your product uses. See Supporting older browsers in the following section.

Additional resources

- Chromium Blog:Developers: Get Ready for New SameSite=None; Secure Cookie Settings
- SameSite cookies explained
- November 2019 Patches

SAMPLE	DOCUMENT
.NET Core MVC	ASP.NET Core 2.1 MVC SameSite cookie sample
.NET Core Razor Pages	ASP.NET Core 2.1 Razor Pages SameSite cookie sample

SAMPLE	DOCUMENT
.NET Core Razor Pages	ASP.NET Core 3.1 Razor Pages SameSite cookie sample

ASP.NET Core 2.1 Razor Pages SameSite cookie sample

9/22/2020 • 3 minutes to read • Edit Online

This sample targets .NET Framework Targeted

ASP.NET Core 2.1 has built-in support for the SameSite attribute, but it was written to the original standard. The patched behavior changed the meaning of `SameSite.None` to emit the sameSite attribute with a value of `None`, rather than not emit the value at all. If you want to not emit the value you can set the `SameSite` property on a cookie to -1.

ASP.NET Core Identity is largely unaffected by SameSite cookies except for advanced scenarios like `IFrames` or `OpenIdConnect` integration.

When using `Identity`, do ***not*** add any cookie providers or call `services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)`, `Identity` takes care of that.

Writing the SameSite attribute

The following code is an example of how to write a SameSite attribute on a cookie:

```
var cookieOptions = new CookieOptions
{
    // Set the secure flag, which Chrome's changes will require for SameSite none.
    // Note this will also require you to be running on HTTPS
    Secure = true,

    // Set the cookie to HTTP only which is good practice unless you really do need
    // to access it client side in scripts.
    HttpOnly = true,

    // Add the SameSite attribute, this will emit the attribute with a value of none.
    // To not emit the attribute at all set the SameSite property to (SameSiteMode)(-1).
    SameSite = SameSiteMode.None
};

// Add the cookie to the response cookie collection
Response.Cookies.Append(CookieName, "cookieValue", cookieOptions);
```

Setting Cookie Authentication and Session State cookies

Cookie authentication, session state and various other components set their sameSite options via Cookie options, for example

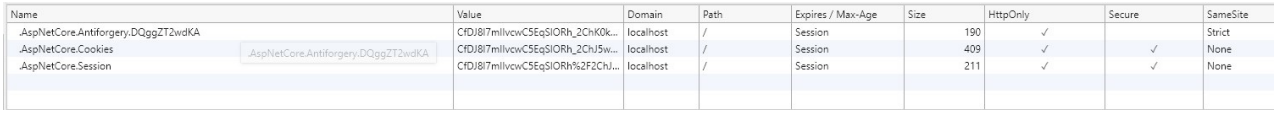
```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.SameSite = SameSiteMode.None;
        options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
        options.Cookie.IsEssential = true;
    });

services.AddSession(options =>
{
    options.Cookie.SameSite = SameSiteMode.None;
    options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
    options.Cookie.IsEssential = true;
});
```

In the preceding code, both cookie authentication and session state set their sameSite attribute to `None`, emitting the attribute with a `None` value, and also set the Secure attribute to `true`.

Run the sample

If you run the sample project, load your browser debugger on the initial page and use it to view the cookie collection for the site. To do so in Edge and Chrome press `F12` then select the `Application` tab and click the site URL under the `Cookies` option in the `Storage` section.



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
.AspNetCore.Antiforgery.DQggZT2wdKA	CfDJ8I7mIlvcwC5EqSIORh_2ChK0k...	localhost	/	Session	190	✓		Strict
.AspNetCore.Cookies	CfDJ8I7mIlvcwC5EqSIORh_2ChJ5w...	localhost	/	Session	409	✓	✓	None
.AspNetCore.Session	CfDJ8I7mIlvcwC5EqSIORh%2F2ChJ...	localhost	/	Session	211	✓	✓	None

You can see from the image above that the cookie created by the sample when you click the "Create SameSite Cookie" button has a SameSite attribute value of `Lax`, matching the value set in the sample code.

Intercepting cookies

In order to intercept cookies, to adjust the none value according to its support in the user's browser agent you must use the `CookiePolicy` middleware. This must be placed into the http request pipeline **before** any components that write cookies and configured within `ConfigureServices()`.

To insert it into the pipeline use `app.UseCookiePolicy()` in the `Configure(IApplicationBuilder, IHostingEnvironment)` method in your Startup.cs. For example

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
       app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseAuthentication();
    app.UseSession();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Then in the `ConfigureServices(IServiceCollection services)` configure the cookie policy to call out to a helper class when cookies are appended or deleted, like so;

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    });
}

private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = (SameSiteMode)(-1);
        }
    }
}
```

The helper function `CheckSameSite(HttpContext, CookieOptions)`:

- Is called when cookies are appended to the request or deleted from the request.
- Checks to see if the `SameSite` property is set to `None`.
- If `SameSite` is set to `None` and the current user agent is known to not support the none attribute value. The check is done using the SameSiteSupport class:
  - Sets `SameSite` to not emit the value by setting the property to `(SameSiteMode)(-1)`

Targeting .NET Framework

ASP.NET Core and System.Web (ASP.NET Classic) have independent implementations of SameSite. The SameSite KB patches for .NET Framework are not required if using ASP.NET Core nor does the System.Web SameSite minimum framework version requirement (.NET 4.7.2) apply to ASP.NET Core.

ASP.NET Core on .NET requires updating NuGet package dependencies to get the appropriate fixes.

To get the ASP.NET Core changes for .NET Framework ensure that you have a direct reference to the patched packages and versions (2.1.14 or later 2.1 versions).

```
<PackageReference Include="Microsoft.Net.Http.Headers" Version="2.1.14" />
<PackageReference Include="Microsoft.AspNetCore.CookiePolicy" Version="2.1.14" />
```

More Information

Chrome Updates ASP.NET Core SameSite Documentation ASP.NET Core 2.1 SameSite Change Announcement

ASP.NET Core 3.1 Razor Pages SameSite cookie sample

9/22/2020 • 3 minutes to read • Edit Online

ASP.NET Core 3.0 has built-in support for the SameSite attribute, including a `SameSiteMode` attribute value of `Unspecified` to suppress writing the attribute.

ASP.NET Core Identity is largely unaffected by SameSite cookies except for advanced scenarios like `IFrames` or `OpenIdConnect` integration.

When using `Identity`, do ***not*** add any cookie providers or call `services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)`, `Identity` takes care of that.

Writing the SameSite attribute

Following is an example of how to write a SameSite attribute on a cookie;

```
var cookieOptions = new CookieOptions
{
    // Set the secure flag, which Chrome's changes will require for SameSite none.
    // Note this will also require you to be running on HTTPS
    Secure = true,

    // Set the cookie to HTTP only which is good practice unless you really do need
    // to access it client side in scripts.
    HttpOnly = true,

    // Add the SameSite attribute, this will emit the attribute with a value of none.
    // To not emit the attribute at all set the SameSite property to SameSiteMode.Unspecified.
    SameSite = SameSiteMode.None
};

// Add the cookie to the response cookie collection
Response.Cookies.Append(CookieName, "cookieValue", cookieOptions);
```

Setting Cookie Authentication and Session State cookies

Cookie authentication, session state and various other components set their sameSite options via Cookie options, for example

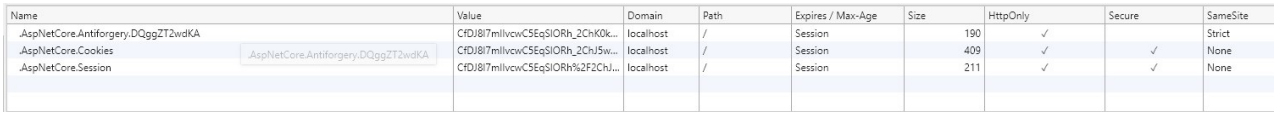
```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.SameSite = SameSiteMode.None;
        options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
        options.Cookie.IsEssential = true;
    });

services.AddSession(options =>
{
    options.Cookie.SameSite = SameSiteMode.None;
    options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
    options.Cookie.IsEssential = true;
});
```

In the code shown above both cookie authentication and session state set their sameSite attribute to None, emitting the attribute with a `None` value, and also set the Secure attribute to true.

Run the sample

If you run the sample project, load your browser debugger on the initial page and use it to view the cookie collection for the site. To do so in Edge and Chrome press `F12` then select the `Application` tab and click the site URL under the `Cookies` option in the `Storage` section.



You can see from the image above that the cookie created by the sample when you click the "Create SameSite Cookie" button has a SameSite attribute value of `Lax`, matching the value set in the sample code.

Intercepting cookies

In order to intercept cookies, to adjust the none value according to its support in the user's browser agent you must use the `CookiePolicy` middleware. This must be placed into the http request pipeline **before** any components that write cookies and configured within `ConfigureServices()`.

To insert it into the pipeline use `app.UseCookiePolicy()` in the `Configure(IApplicationBuilder, IHostingEnvironment)` method in your Startup.cs. For example

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
       app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseAuthentication();
    app.UseSession();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Then in the `ConfigureServices(IServiceCollection services)` configure the cookie policy to call out to a helper class when cookies are appended or deleted, like so;

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    });
}

private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}
```

The helper function `CheckSameSite(HttpContext, CookieOptions)`:

- Is called when cookies are appended to the request or deleted from the request.
- Checks to see if the `SameSite` property is set to `None`.
- If `SameSite` is set to `None` and the current user agent is known to not support the none attribute value. The check is done using the SameSiteSupport class:
  - Sets `SameSite` to not emit the value by setting the property to `(SameSiteMode)(-1)`

More Information

Chrome Updates ASP.NET Core SameSite Documentation

ASP.NET Core 2.1 MVC SameSite cookie sample

9/22/2020 • 3 minutes to read • Edit Online

ASP.NET Core 2.1 has built-in support for the SameSite attribute, but it was written to the original standard. The patched behavior changed the meaning of `SameSite.None` to emit the sameSite attribute with a value of `None`, rather than not emit the value at all. If you want to not emit the value you can set the `SameSite` property on a cookie to -1.

ASP.NET Core Identity is largely unaffected by SameSite cookies except for advanced scenarios like `IFrames` or `OpenIdConnect` integration.

When using `Identity`, do ***not*** add any cookie providers or call `services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)`, `Identity` takes care of that.

Writing the SameSite attribute

Following is an example of how to write a SameSite attribute on a cookie:

```
var cookieOptions = new CookieOptions
{
    // Set the secure flag, which Chrome's changes will require for SameSite none.
    // Note this will also require you to be running on HTTPS
    Secure = true,

    // Set the cookie to HTTP only which is good practice unless you really do need
    // to access it client side in scripts.
    HttpOnly = true,

    // Add the SameSite attribute, this will emit the attribute with a value of none.
    // To not emit the attribute at all set the SameSite property to (SameSiteMode)(-1).
    SameSite = SameSiteMode.None
};

// Add the cookie to the response cookie collection
Response.Cookies.Append(CookieName, "cookieValue", cookieOptions);
```

Setting Cookie Authentication and Session State cookies

Cookie authentication, session state and various other components set their sameSite options via Cookie options, for example

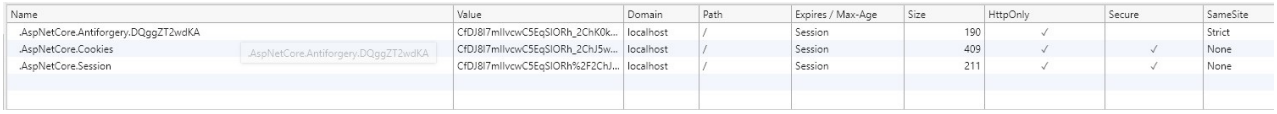
```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.SameSite = SameSiteMode.None;
        options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
        options.Cookie.IsEssential = true;
    });

services.AddSession(options =>
{
    options.Cookie.SameSite = SameSiteMode.None;
    options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
    options.Cookie.IsEssential = true;
});
```

In the preceding code, both cookie authentication and session state set their sameSite attribute to `None`, emitting the attribute with a `None` value, and also set the Secure attribute to true.

Run the sample

If you run the sample project, load your browser debugger on the initial page and use it to view the cookie collection for the site. To do so in Edge and Chrome press `F12` then select the `Application` tab and click the site URL under the `Cookies` option in the `Storage` section.



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
.AspNetCore.Antiforgery.DQggZT2wdKA	CfDJ8I7mIlvcwC5EqSIORh_2ChK0k...	localhost	/	Session	190	✓		Strict
.AspNetCore.Cookies	CfDJ8I7mIlvcwC5EqSIORh_2ChJ5w...	localhost	/	Session	409	✓	✓	None
.AspNetCore.Session	CfDJ8I7mIlvcwC5EqSIORh%2F2ChJ...	localhost	/	Session	211	✓	✓	None

You can see from the image above that the cookie created by the sample when you click the "Create SameSite Cookie" button has a SameSite attribute value of `Lax`, matching the value set in the sample code.

Intercepting cookies

In order to intercept cookies, to adjust the none value according to its support in the user's browser agent you must use the `CookiePolicy` middleware. This must be placed into the http request pipeline **before** any components that write cookies and configured within `ConfigureServices()`.

To insert it into the pipeline use `app.UseCookiePolicy()` in the `Configure(IApplicationBuilder, IHostingEnvironment)` method in Startup.cs. For example:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
       app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseAuthentication();
    app.UseSession();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Then in the `ConfigureServices(IServiceCollection services)` configure the cookie policy to call out to a helper class when cookies are appended or deleted. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    });
}

private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (SameSite.BrowserDetection.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = (SameSiteMode)(-1);
        }
    }
}
```

The helper function `CheckSameSite(HttpContext, CookieOptions)`:

- Is called when cookies are appended to the request or deleted from the request.
- Checks to see if the `SameSite` property is set to `None`.
- If `SameSite` is set to `None` and the current user agent is known to not support the none attribute value. The check is done using the SameSiteSupport class:
  - Sets `SameSite` to not emit the value by setting the property to `(SameSiteMode)(-1)`

Targeting .NET Framework

ASP.NET Core and System.Web (ASP.NET Classic) have independent implementations of SameSite. The SameSite KB patches for .NET Framework are not required if using ASP.NET Core nor does the System.Web SameSite minimum framework version requirement (.NET 4.7.2) apply to ASP.NET Core.

ASP.NET Core on .NET requires updating nuget package dependencies to get the appropriate fixes.

To get the ASP.NET Core changes for .NET Framework ensure that you have a direct reference to the patched packages and versions (2.1.14 or later 2.1 versions).

```
<PackageReference Include="Microsoft.Net.Http.Headers" Version="2.1.14" />
<PackageReference Include="Microsoft.AspNetCore.CookiePolicy" Version="2.1.14" />
```

More Information

Chrome Updates ASP.NET Core SameSite Documentation ASP.NET Core 2.1 SameSite Change Announcement

Client IP safelist for ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By Damien Bowden and Tom Dykstra

This article shows three ways to implement an IP address safelist (also known as an allow list) in an ASP.NET Core app. An accompanying sample app demonstrates all three approaches. You can use:

- Middleware to check the remote IP address of every request.
- MVC action filters to check the remote IP address of requests for specific controllers or action methods.
- Razor Pages filters to check the remote IP address of requests for Razor pages.

In each case, a string containing approved client IP addresses is stored in an app setting. The middleware or filter:

- Parses the string into an array.
- Checks if the remote IP address exists in the array.

Access is allowed if the array contains the IP address. Otherwise, an HTTP 403 Forbidden status code is returned.

View or download sample code (how to download)

IP address safelist

In the sample app, the IP address safelist is:

- Defined by the `AdminSafeList` property in the *appsettings.json* file.
- A semicolon-delimited string that may contain both Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) addresses.

```
{
  "AdminSafeList": "127.0.0.1;192.168.1.5;::1",
  "Logging": {
```

In the preceding example, the IPv4 addresses of `127.0.0.1` and `192.168.1.5` and the IPv6 loopback address of `::1` (compressed format for `0:0:0:0:0:0:0:1`) are allowed.

Middleware

The `Startup.Configure` method adds the custom `AdminSafeListMiddleware` middleware type to the app's request pipeline. The safelist is retrieved with the .NET Core configuration provider and is passed as a constructor parameter.

```
app.UseMiddleware<AdminSafeListMiddleware>(Configuration["AdminSafeList"]);
```

The middleware parses the string into an array and searches for the remote IP address in the array. If the remote IP address isn't found, the middleware returns HTTP 403 Forbidden. This validation process is bypassed for HTTP GET requests.

```
public class AdminSafeListMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<AdminSafeListMiddleware> _logger;
    private readonly string _safelist;

    public AdminSafeListMiddleware(
        RequestDelegate next,
        ILogger<AdminSafeListMiddleware> logger,
        string safelist)
    {
        _safelist = safelist;
        _next = next;
        _logger = logger;
    }

    public async Task Invoke(HttpContext context)
    {
        if (context.Request.Method != HttpMethod.Get.Method)
        {
            var remoteIp = context.Connection.RemoteIpAddress;
            _logger.LogDebug("Request from Remote IP address: {RemoteIp}", remoteIp);

            string[] ip = _safelist.Split(';');

            var bytes = remoteIp.GetAddressBytes();
            var badIp = true;
            foreach (var address in ip)
            {
                var testIp = IPAddress.Parse(address);
                if (testIp.GetAddressBytes().SequenceEqual(bytes))
                {
                    badIp = false;
                    break;
                }
            }

            if (badIp)
            {
                _logger.LogWarning(
                    "Forbidden Request from Remote IP address: {RemoteIp}", remoteIp);
                context.Response.StatusCode = StatusCodes.Status403Forbidden;
                return;
            }
        }

        await _next.Invoke(context);
    }
}
```

Action filter

If you want safelist-driven access control for specific MVC controllers or action methods, use an action filter. For example:

```
public class ClientIpCheckActionFilter : ActionFilterAttribute
{
    private readonly ILogger _logger;
    private readonly string _safelist;

    public ClientIpCheckActionFilter(string safelist, ILogger logger)
    {
        _safelist = safelist;
        _logger = logger;
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
        _logger.LogDebug("Remote IpAddress: {RemoteIp}", remoteIp);
        var ip = _safelist.Split(';');
        var badIp = true;

        if (remoteIp.IsIPv4MappedToIPv6)
        {
            remoteIp = remoteIp.MapToIPv4();
        }

        foreach (var address in ip)
        {
            var testIp = IPAddress.Parse(address);

            if (testIp.Equals(remoteIp))
            {
                badIp = false;
                break;
            }
        }

        if (badIp)
        {
            _logger.LogWarning("Forbidden Request from IP: {RemoteIp}", remoteIp);
            context.Result = new StatusCodeResult(StatusCodes.Status403Forbidden);
            return;
        }

        base.OnActionExecuting(context);
    }
}
```

In `Startup.ConfigureServices`, add the action filter to the MVC filters collection. In the following example, a `ClientIpCheckActionFilter` action filter is added. A safelist and a console logger instance are passed as constructor parameters.

```
services.AddScoped<ClientIpCheckActionFilter>(container =>
{
    var loggerFactory = container.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger<ClientIpCheckActionFilter>();

    return new ClientIpCheckActionFilter(
        Configuration["AdminSafeList"], logger);
});
```

```
services.AddScoped<ClientIpCheckActionFilter>(_ =>
{
    var logger = _loggerFactory.CreateLogger<ClientIpCheckActionFilter>();

    return new ClientIpCheckActionFilter(
        Configuration["AdminSafeList"], logger);
});
```

The action filter can then be applied to a controller or action method with the [ServiceFilter] attribute:

```
[ServiceFilter(typeof(ClientIpCheckActionFilter))]
[HttpGet]
public IEnumerable<string> Get()
```

In the sample app, the action filter is applied to the controller's `Get` action method. When you test the app by sending:

- An HTTP GET request, the `[ServiceFilter]` attribute validates the client IP address. If access is allowed to the `Get` action method, a variation of the following console output is produced by the action filter and action method:

  ```
  dbug: ClientIpSafelistComponents.Filters.ClientIpCheckActionFilter[0]
        Remote IpAddress: ::1
  dbug: ClientIpAspNetCore.Controllers.ValuesController[0]
        successful HTTP GET
  ```

- An HTTP request verb other than GET, the `AdminSafeListMiddleware` middleware validates the client IP address.

Razor Pages filter

If you want safelist-driven access control for a Razor Pages app, use a Razor Pages filter. For example:

```
public class ClientIpCheckPageFilter : IPageFilter
{
    private readonly ILogger _logger;
    private readonly string _safelist;

    public ClientIpCheckPageFilter(
        string safelist,
        ILogger logger)
    {
        _safelist = safelist;
        _logger = logger;
    }

    public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
        _logger.LogDebug(
            "Remote IpAddress: {RemoteIp}", remoteIp);

        string[] ip = _safelist.Split(';');

        var badIp = true;
        foreach (var address in ip)
        {
            if (remoteIp.IsIPv4MappedToIPv6)
            {
                remoteIp = remoteIp.MapToIPv4();
            }
            var testIp = IPAddress.Parse(address);
            if (testIp.Equals(remoteIp))
            {
                badIp = false;
                break;
            }
        }

        if (badIp)
        {
            _logger.LogWarning(
                "Forbidden Request from Remote IP address: {RemoteIp}", remoteIp);
            context.Result = new StatusCodeResult(StatusCodes.Status403Forbidden);
            return;
        }
    }

    public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
    {
    }

    public void OnPageHandlerSelected(PageHandlerSelectedContext context)
    {
    }
}
```

In `Startup.ConfigureServices`, enable the Razor Pages filter by adding it to the MVC filters collection. In the following example, a `ClientIpCheckPageFilter` Razor Pages filter is added. A safelist and a console logger instance are passed as constructor parameters.

```
services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        var logger = LoggerFactory.Create(builder => builder.AddConsole())
                        .CreateLogger<ClientIpCheckPageFilter>();
        var filter = new ClientIpCheckPageFilter(
            Configuration["AdminSafeList"], logger);

        options.Filters.Add(filter);
    });
```

```
services.AddMvc(options =>
{
    var logger = _loggerFactory.CreateLogger<ClientIpCheckPageFilter>();
    var clientIpCheckPageFilter = new ClientIpCheckPageFilter(
        Configuration["AdminSafeList"], logger);

    options.Filters.Add(clientIpCheckPageFilter);
}).SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
```

When the sample app's *Index* Razor page is requested, the Razor Pages filter validates the client IP address. The filter produces a variation of the following console output:

```
dbug: ClientIpSafelistComponents.Filters.ClientIpCheckPageFilter[0]
      Remote IpAddress: ::1
```

Additional resources

- ASP.NET Core Middleware
- Action filters
- Filter methods for Razor Pages in ASP.NET Core

ASP.NET Core Performance Best Practices

9/22/2020 • 17 minutes to read • Edit Online

By Mike Rousos

This article provides guidelines for performance best practices with ASP.NET Core.

Cache aggressively

Caching is discussed in several parts of this document. For more information, see Response caching in ASP.NET Core.

Understand hot code paths

In this document, a *hot code path* is defined as a code path that is frequently called and where much of the execution time occurs. Hot code paths typically limit app scale-out and performance and are discussed in several parts of this document.

Avoid blocking calls

ASP.NET Core apps should be designed to process many requests simultaneously. Asynchronous APIs allow a small pool of threads to handle thousands of concurrent requests by not waiting on blocking calls. Rather than waiting on a long-running synchronous task to complete, the thread can work on another request.

A common performance problem in ASP.NET Core apps is blocking calls that could be asynchronous. Many synchronous blocking calls lead to Thread Pool starvation and degraded response times.

Do not:

- Block asynchronous execution by calling Task.Wait or Task.Result.
- Acquire locks in common code paths. ASP.NET Core apps are most performant when architected to run code in parallel.
- Call Task.Run and immediately await it. ASP.NET Core already runs app code on normal Thread Pool threads, so calling Task.Run only results in extra unnecessary Thread Pool scheduling. Even if the scheduled code would block a thread, Task.Run does not prevent that.

Do:

- Make hot code paths asynchronous.
- Call data access, I/O, and long-running operations APIs asynchronously if an asynchronous API is available. Do **not** use Task.Run to make a synchronous API asynchronous.
- Make controller/Razor Page actions asynchronous. The entire call stack is asynchronous in order to benefit from async/await patterns.

A profiler, such as PerfView, can be used to find threads frequently added to the Thread Pool. The `Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Start` event indicates a thread added to the thread pool.

Return IEnumerable<T> or IAsyncEnumerable<T>

Returning `IEnumerable<T>` from an action results in synchronous collection iteration by the serializer. The result is the blocking of calls and a potential for thread pool starvation. To avoid synchronous enumeration, use

`ToListAsync` before returning the enumerable.

Beginning with ASP.NET Core 3.0, `IAsyncEnumerable<T>` can be used as an alternative to `IEnumerable<T>` that enumerates asynchronously. For more information, see Controller action return types.

Minimize large object allocations

The .NET Core garbage collector manages allocation and release of memory automatically in ASP.NET Core apps. Automatic garbage collection generally means that developers don't need to worry about how or when memory is freed. However, cleaning up unreferenced objects takes CPU time, so developers should minimize allocating objects in hot code paths. Garbage collection is especially expensive on large objects (> 85 K bytes). Large objects are stored on the large object heap and require a full (generation 2) garbage collection to clean up. Unlike generation 0 and generation 1 collections, a generation 2 collection requires a temporary suspension of app execution. Frequent allocation and de-allocation of large objects can cause inconsistent performance.

Recommendations:

- **Do** consider caching large objects that are frequently used. Caching large objects prevents expensive allocations.
- **Do** pool buffers by using an ArrayPool<T> to store large arrays.
- **Do not** allocate many, short-lived large objects on hot code paths.

Memory issues, such as the preceding, can be diagnosed by reviewing garbage collection (GC) stats in PerfView and examining:

- Garbage collection pause time.
- What percentage of the processor time is spent in garbage collection.
- How many garbage collections are generation 0, 1, and 2.

For more information, see Garbage Collection and Performance.

Optimize data access and I/O

Interactions with a data store and other remote services are often the slowest parts of an ASP.NET Core app. Reading and writing data efficiently is critical for good performance.

Recommendations:

- **Do** call all data access APIs asynchronously.
- **Do not** retrieve more data than is necessary. Write queries to return just the data that's necessary for the current HTTP request.
- **Do** consider caching frequently accessed data retrieved from a database or remote service if slightly out-of-date data is acceptable. Depending on the scenario, use a MemoryCache or a DistributedCache. For more information, see Response caching in ASP.NET Core.
- **Do** minimize network round trips. The goal is to retrieve the required data in a single call rather than several calls.
- **Do** use no-tracking queries in Entity Framework Core when accessing data for read-only purposes. EF Core can return the results of no-tracking queries more efficiently.
- **Do** filter and aggregate LINQ queries (with `.Where`, `.Select`, or `.Sum` statements, for example) so that the filtering is performed by the database.
- **Do** consider that EF Core resolves some query operators on the client, which may lead to inefficient query execution. For more information, see Client evaluation performance issues.
- **Do not** use projection queries on collections, which can result in executing "N + 1" SQL queries. For more information, see Optimization of correlated subqueries.

See EF High Performance for approaches that may improve performance in high-scale apps:

- DbContext pooling
- Explicitly compiled queries

We recommend measuring the impact of the preceding high-performance approaches before committing the code base. The additional complexity of compiled queries may not justify the performance improvement.

Query issues can be detected by reviewing the time spent accessing data with Application Insights or with profiling tools. Most databases also make statistics available concerning frequently executed queries.

Pool HTTP connections with HttpClientFactory

Although HttpClient implements the `IDisposable` interface, it's designed for reuse. Closed `HttpClient` instances leave sockets open in the `TIME_WAIT` state for a short period of time. If a code path that creates and disposes of `HttpClient` objects is frequently used, the app may exhaust available sockets. HttpClientFactory was introduced in ASP.NET Core 2.1 as a solution to this problem. It handles pooling HTTP connections to optimize performance and reliability.

Recommendations:

- **Do not** create and dispose of `HttpClient` instances directly.
- **Do** use HttpClientFactory to retrieve `HttpClient` instances. For more information, see Use HttpClientFactory to implement resilient HTTP requests.

Keep common code paths fast

You want all of your code to be fast. Frequently-called code paths are the most critical to optimize. These include:

- Middleware components in the app's request processing pipeline, especially middleware run early in the pipeline. These components have a large impact on performance.
- Code that's executed for every request or multiple times per request. For example, custom logging, authorization handlers, or initialization of transient services.

Recommendations:

- **Do not** use custom middleware components with long-running tasks.
- **Do** use performance profiling tools, such as Visual Studio Diagnostic Tools or PerfView), to identify hot code paths.

Complete long-running Tasks outside of HTTP requests

Most requests to an ASP.NET Core app can be handled by a controller or page model calling necessary services and returning an HTTP response. For some requests that involve long-running tasks, it's better to make the entire request-response process asynchronous.

Recommendations:

- **Do not** wait for long-running tasks to complete as part of ordinary HTTP request processing.
- **Do** consider handling long-running requests with background services or out of process with an Azure Function. Completing work out-of-process is especially beneficial for CPU-intensive tasks.
- **Do** use real-time communication options, such as SignalR, to communicate with clients asynchronously.

Minify client assets

ASP.NET Core apps with complex front-ends frequently serve many JavaScript, CSS, or image files. Performance of

initial load requests can be improved by:

- Bundling, which combines multiple files into one.
- Minifying, which reduces the size of files by removing whitespace and comments.

Recommendations:

- **Do** use ASP.NET Core's built-in support for bundling and minifying client assets.
- **Do** consider other third-party tools, such as Webpack, for complex client asset management.

Compress responses

Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses. For more information, see Response compression.

Use the latest ASP.NET Core release

Each new release of ASP.NET Core includes performance improvements. Optimizations in .NET Core and ASP.NET Core mean that newer versions generally outperform older versions. For example, .NET Core 2.1 added support for compiled regular expressions and benefitted from Span<T>. ASP.NET Core 2.2 added support for HTTP/2. ASP.NET Core 3.0 adds many improvements that reduce memory usage and improve throughput. If performance is a priority, consider upgrading to the current version of ASP.NET Core.

Minimize exceptions

Exceptions should be rare. Throwing and catching exceptions is slow relative to other code flow patterns. Because of this, exceptions shouldn't be used to control normal program flow.

Recommendations:

- **Do not** use throwing or catching exceptions as a means of normal program flow, especially in hot code paths.
- **Do** include logic in the app to detect and handle conditions that would cause an exception.
- **Do** throw or catch exceptions for unusual or unexpected conditions.

App diagnostic tools, such as Application Insights, can help to identify common exceptions in an app that may affect performance.

Performance and reliability

The following sections provide performance tips and known reliability problems and solutions.

Avoid synchronous read or write on HttpRequest/HttpResponse body

All I/O in ASP.NET Core is asynchronous. Servers implement the `Stream` interface, which has both synchronous and asynchronous overloads. The asynchronous ones should be preferred to avoid blocking thread pool threads. Blocking threads can lead to thread pool starvation.

Do not do this: The following example uses the ReadToEnd. It blocks the current thread to wait for the result. This is an example of sync over async.

```
public class BadStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public ActionResult<ContosoData> Get()
    {
        var json = new StreamReader(Request.Body).ReadToEnd();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }
}
```

In the preceding code, `Get` synchronously reads the entire HTTP request body into memory. If the client is slowly uploading, the app is doing sync over async. The app does sync over async because Kestrel does **NOT** support synchronous reads.

Do this: The following example uses ReadToEndAsync and does not block the thread while reading.

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        var json = await new StreamReader(Request.Body).ReadToEndAsync();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }

}
```

The preceding code asynchronously reads the entire HTTP request body into memory.

> **WARNING**
> If the request is large, reading the entire HTTP request body into memory could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see Avoid reading large request bodies or response bodies into memory in this document.

Do this: The following example is fully asynchronous using a non buffered request body:

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        return await JsonSerializer.DeserializeAsync<ContosoData>(Request.Body);
    }
}
```

The preceding code asynchronously de-serializes the request body into a C# object.

Prefer ReadFormAsync over Request.Form

Use `HttpContext.Request.ReadFormAsync` instead of `HttpContext.Request.Form`. `HttpContext.Request.Form` can be safely read only with the following conditions:

- The form has been read by a call to `ReadFormAsync`, and
- The cached form value is being read using `HttpContext.Request.Form`

Do not do this: The following example uses `HttpContext.Request.Form`. `HttpContext.Request.Form` uses sync over async and can lead to thread pool starvation.

```
public class BadReadController : Controller
{
    [HttpPost("/form-body")]
    public IActionResult Post()
    {
        var form =  HttpContext.Request.Form;

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Do this: The following example uses `HttpContext.Request.ReadFormAsync` to read the form body asynchronously.

```
public class GoodReadController : Controller
{
    [HttpPost("/form-body")]
    public async Task<IActionResult> Post()
    {
       var form = await HttpContext.Request.ReadFormAsync();

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Avoid reading large request bodies or response bodies into memory

In .NET, every object allocation greater than 85 KB ends up in the large object heap (LOH). Large objects are expensive in two ways:

- The allocation cost is high because the memory for a newly allocated large object has to be cleared. The CLR guarantees that memory for all newly allocated objects is cleared.
- LOH is collected with the rest of the heap. LOH requires a full garbage collection or Gen2 collection.

This blog post describes the problem succinctly:

> When a large object is allocated, it's marked as Gen 2 object. Not Gen 0 as for small objects. The consequences are that if you run out of memory in LOH, GC cleans up the whole managed heap, not only LOH. So it cleans up Gen 0, Gen 1 and Gen 2 including LOH. This is called full garbage collection and is the most time-consuming garbage collection. For many applications, it can be acceptable. But definitely not for high-performance web servers, where few big memory buffers are needed to handle an average web request (read from a socket, decompress, decode JSON & more).

Naively storing a large request or response body into a single `byte[]` or `string`:

- May result in quickly running out of space in the LOH.
- May cause performance issues for the app because of full GCs running.

Working with a synchronous data processing API

When using a serializer/de-serializer that only supports synchronous reads and writes (for example, JSON.NET):

- Buffer the data into memory asynchronously before passing it into the serializer/de-serializer.

> **WARNING**
> If the request is large, it could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see Avoid reading large request bodies or response bodies into memory in this document.

ASP.NET Core 3.0 uses System.Text.Json by default for JSON serialization. System.Text.Json:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically higher performance than `Newtonsoft.Json`.

Do not store IHttpContextAccessor.HttpContext in a field

The IHttpContextAccessor.HttpContext returns the `HttpContext` of the active request when accessed from the request thread. The `IHttpContextAccessor.HttpContext` should **not** be stored in a field or variable.

Do not do this: The following example stores the `HttpContext` in a field and then attempts to use it later.

```
public class MyBadType
{
    private readonly HttpContext _context;
    public MyBadType(IHttpContextAccessor accessor)
    {
        _context = accessor.HttpContext;
    }

    public void CheckAdmin()
    {
        if (!_context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't an admin");
        }
    }
}
```

The preceding code frequently captures a null or incorrect `HttpContext` in the constructor.

Do this: The following example:

- Stores the IHttpContextAccessor in a field.
- Uses the `HttpContext` field at the correct time and checks for `null`.

```
public class MyGoodType
{
    private readonly IHttpContextAccessor _accessor;
    public MyGoodType(IHttpContextAccessor accessor)
    {
        _accessor = accessor;
    }

    public void CheckAdmin()
    {
        var context = _accessor.HttpContext;
        if (context != null && !context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't an admin");
        }
    }
}
```

Do not access HttpContext from multiple threads

`HttpContext` is *NOT* thread-safe. Accessing `HttpContext` from multiple threads in parallel can result in undefined behavior such as hangs, crashes, and data corruption.

Do not do this: The following example makes three parallel requests and logs the incoming request path before and after the outgoing HTTP request. The request path is accessed from multiple threads, potentially in parallel.

```
public class AsyncBadSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        var query1 = SearchAsync(SearchEngine.Google, query);
        var query2 = SearchAsync(SearchEngine.Bing, query);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine, string query)
    {
        var searchResults = _searchService.Empty();
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   HttpContext.Request.Path);
            searchResults = _searchService.Search(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
                                   HttpContext.Request.Path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}",
                             HttpContext.Request.Path);
        }

        return await searchResults;
    }
```

Do this: The following example copies all data from the incoming request before making the three parallel requests.

```
public class AsyncGoodSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        string path = HttpContext.Request.Path;
        var query1 = SearchAsync(SearchEngine.Google, query,
                                 path);
        var query2 = SearchAsync(SearchEngine.Bing, query, path);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query, path);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine, string query,
                                                  string path)
    {
        var searchResults = _searchService.Empty();
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   path);
            searchResults = await _searchService.SearchAsync(engine, query);
            _logger.LogInformation("Finishing search query from {path}.", path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}", path);
        }

        return await searchResults;
    }
```

Do not use the HttpContext after the request is complete

`HttpContext` is only valid as long as there is an active HTTP request in the ASP.NET Core pipeline. The entire ASP.NET Core pipeline is an asynchronous chain of delegates that executes every request. When the `Task` returned from this chain completes, the `HttpContext` is recycled.

Do not do this: The following example uses `async void` which makes the HTTP request complete when the first `await` is reached:

- Which is **ALWAYS** a bad practice in ASP.NET Core apps.
- Accesses the `HttpResponse` after the HTTP request is complete.
- Crashes the process.

```
public class AsyncBadVoidController : Controller
{
    [HttpGet("/async")]
    public async void Get()
    {
        await Task.Delay(1000);

        // The following line will crash the process because of writing after the
        // response has completed on a background thread. Notice async void Get()

        await Response.WriteAsync("Hello World");
    }
}
```

Do this: The following example returns a `Task` to the framework, so the HTTP request doesn't complete until the action completes.

```
public class AsyncGoodTaskController : Controller
{
    [HttpGet("/async")]
    public async Task Get()
    {
        await Task.Delay(1000);

        await Response.WriteAsync("Hello World");
    }
}
```

Do not capture the HttpContext in background threads

Do not do this: The following example shows a closure is capturing the `HttpContext` from the `Controller` property. This is a bad practice because the work item could:

- Run outside of the request scope.
- Attempt to read the wrong `HttpContext`.

```
[HttpGet("/fire-and-forget-1")]
public IActionResult BadFireAndForget()
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        var path = HttpContext.Request.Path;
        Log(path);
    });

    return Accepted();
}
```

Do this: The following example:

- Copies the data required in the background task during the request.
- Doesn't reference anything from the controller.

```
[HttpGet("/fire-and-forget-3")]
public IActionResult GoodFireAndForget()
{
    string path = HttpContext.Request.Path;
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        Log(path);
    });

    return Accepted();
}
```

Background tasks should be implemented as hosted services. For more information, see Background tasks with hosted services.

Do not capture services injected into the controllers on background threads

Do not do this: The following example shows a closure is capturing the `DbContext` from the `Controller` action parameter. This is a bad practice. The work item could run outside of the request scope. The `ContosoDbContext` is scoped to the request, resulting in an `ObjectDisposedException`.

```
[HttpGet("/fire-and-forget-1")]
public IActionResult FireAndForget1([FromServices]ContosoDbContext context)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        context.Contoso.Add(new Contoso());
        await context.SaveChangesAsync();
    });

    return Accepted();
}
```

Do this: The following example:

- Injects an IServiceScopeFactory in order to create a scope in the background work item. `IServiceScopeFactory` is a singleton.
- Creates a new dependency injection scope in the background thread.
- Doesn't reference anything from the controller.
- Doesn't capture the `ContosoDbContext` from the incoming request.

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        using (var scope = serviceScopeFactory.CreateScope())
        {
            var context = scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });

    return Accepted();
}
```

The following highlighted code:

- Creates a scope for the lifetime of the background operation and resolves services from it.
- Uses `ContosoDbContext` from the correct scope.

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        using (var scope = serviceScopeFactory.CreateScope())
        {
            var context = scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });

    return Accepted();
}
```

Do not modify the status code or headers after the response body has started

ASP.NET Core does not buffer the HTTP response body. The first time the response is written:

- The headers are sent along with that chunk of the body to the client.
- It's no longer possible to change response headers.

Do not do this: The following code tries to add response headers after the response has already started:

```
app.Use(async (context, next) =>
{
    await next();

    context.Response.Headers["test"] = "test value";
});
```

In the preceding code, `context.Response.Headers["test"] = "test value";` will throw an exception if `next()` has written to the response.

Do this: The following example checks if the HTTP response has started before modifying the headers.

```
app.Use(async (context, next) =>
{
    await next();

    if (!context.Response.HasStarted)
    {
        context.Response.Headers["test"] = "test value";
    }
});
```

Do this: The following example uses `HttpResponse.OnStarting` to set the headers before the response headers are flushed to the client.

Checking if the response has not started allows registering a callback that will be invoked just before response headers are written. Checking if the response has not started:

- Provides the ability to append or override headers just in time.
- Doesn't require knowledge of the next middleware in the pipeline.

```
app.Use(async (context, next) =>
{
    context.Response.OnStarting(() =>
    {
        context.Response.Headers["someheader"] = "somevalue";
        return Task.CompletedTask;
    });

    await next();
});
```

Do not call next() if you have already started writing to the response body

Components only expect to be called if it's possible for them to handle and manipulate the response.

Use In-process hosting with IIS

Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter. The loopback adapter is a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the Windows Process Activation Service (WAS).

Projects default to the in-process hosting model in ASP.NET Core 3.0 and later.

For more information, see Host ASP.NET Core on Windows with IIS

Memory management and garbage collection (GC) in ASP.NET Core

9/22/2020 • 12 minutes to read • Edit Online

By Sébastien Ros and Rick Anderson

Memory management is complex, even in a managed framework like .NET. Analyzing and understanding memory issues can be challenging. This article:

- Was motivated by many *memory leak* and *GC not working* issues. Most of these issues were caused by not understanding how memory consumption works in .NET Core, or not understanding how it's measured.
- Demonstrates problematic memory use, and suggests alternative approaches.

How garbage collection (GC) works in .NET Core

The GC allocates heap segments where each segment is a contiguous range of memory. Objects placed in the heap are categorized into one of 3 generations: 0, 1, or 2. The generation determines the frequency the GC attempts to release memory on managed objects that are no longer referenced by the app. Lower numbered generations are GC'd more frequently.

Objects are moved from one generation to another based on their lifetime. As objects live longer, they are moved into a higher generation. As mentioned previously, higher generations are GC'd less often. Short term lived objects always remain in generation 0. For example, objects that are referenced during the life of a web request are short lived. Application level singletons generally migrate to generation 2.

When an ASP.NET Core app starts, the GC:

- Reserves some memory for the initial heap segments.
- Commits a small portion of memory when the runtime is loaded.

The preceding memory allocations are done for performance reasons. The performance benefit comes from heap segments in contiguous memory.

Call GC.Collect

Calling GC.Collect explicitly:

- Should **not** be done by production ASP.NET Core apps.
- Is useful when investigating memory leaks.
- When investigating, verifies the GC has removed all dangling objects from memory so memory can be measured.

Analyzing the memory usage of an app

Dedicated tools can help analyzing memory usage:

- Counting object references
- Measuring how much impact the GC has on CPU usage
- Measuring memory space used for each generation

Use the following tools to analyze memory usage:

- dotnet-trace: Can be used on production machines.

- Analyze memory usage without the Visual Studio debugger
- Profile memory usage in Visual Studio

Detecting memory issues

Task Manager can be used to get an idea of how much memory an ASP.NET app is using. The Task Manager memory value:

- Represents the amount of memory that is used by the ASP.NET process.
- Includes the app's living objects and other memory consumers such as native memory usage.

If the Task Manager memory value increases indefinitely and never flattens out, the app has a memory leak. The following sections demonstrate and explain several memory usage patterns.

Sample display memory usage app

The MemoryLeak sample app is available on GitHub. The MemoryLeak app:

- Includes a diagnostic controller that gathers real-time memory and GC data for the app.
- Has an Index page that displays the memory and GC data. The Index page is refreshed every second.
- Contains an API controller that provides various memory load patterns.
- Is not a supported tool, however, it can be used to display memory usage patterns of ASP.NET Core apps.

Run MemoryLeak. Allocated memory slowly increases until a GC occurs. Memory increases because the tool allocates custom object to capture data. The following image shows the MemoryLeak Index page when a Gen 0 GC occurs. The chart shows 0 RPS (Requests per second) because no API endpoints from the API controller have been called.

The chart displays two values for the memory usage:

- Allocated: the amount of memory occupied by managed objects
- Working set: The set of pages in the virtual address space of the process that are currently resident in physical memory. The working set shown is the same value Task Manager displays.

Transient objects

The following API creates a 10-KB String instance and returns it to the client. On each request, a new object is

allocated in memory and written to the response. Strings are stored as UTF-16 characters in .NET so each character takes 2 bytes in memory.

```
[HttpGet("bigstring")]
public ActionResult<string> GetBigString()
{
    return new String('x', 10 * 1024);
}
```

The following graph is generated with a relatively small load in to show how memory allocations are impacted by the GC.

The preceding chart shows:

- 4K RPS (Requests per second).
- Generation 0 GC collections occur about every two seconds.
- The working set is constant at approximately 500 MB.
- CPU is 12%.
- The memory consumption and release (through GC) is stable.

The following chart is taken at the max throughput that can be handled by the machine.

The preceding chart shows:

- 22K RPS
- Generation 0 GC collections occur several times per second.
- Generation 1 collections are triggered because the app allocated significantly more memory per second.
- The working set is constant at approximately 500 MB.
- CPU is 33%.
- The memory consumption and release (through GC) is stable.
- The CPU (33%) is not over-utilized, therefore the garbage collection can keep up with a high number of allocations.

Workstation GC vs. Server GC

The .NET Garbage Collector has two different modes:

- **Workstation GC**: Optimized for the desktop.
- **Server GC**. The default GC for ASP.NET Core apps. Optimized for the server.

The GC mode can be set explicitly in the project file or in the *runtimeconfig.json* file of the published app. The following markup shows setting `ServerGarbageCollection` in the project file:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Changing `ServerGarbageCollection` in the project file requires the app to be rebuilt.

Note: Server garbage collection is **not** available on machines with a single core. For more information, see IsServerGC.

The following image shows the memory profile under a 5K RPS using the Workstation GC.

The differences between this chart and the server version are significant:

- The working set drops from 500 MB to 70 MB.
- The GC does generation 0 collections multiple times per second instead of every two seconds.
- GC drops from 300 MB to 10 MB.

On a typical web server environment, CPU usage is more important than memory, therefore the Server GC is better. If memory utilization is high and CPU usage is relatively low, the Workstation GC might be more performant. For example, high density hosting several web apps where memory is scarce.

GC using Docker and small containers

When multiple containerized apps are running on one machine, Workstation GC might be more preformant than Server GC. For more information, see Running with Server GC in a Small Container and Running with Server GC in a Small Container Scenario Part 1 – Hard Limit for the GC Heap.

Persistent object references

The GC cannot free objects that are referenced. Objects that are referenced but no longer needed result in a memory leak. If the app frequently allocates objects and fails to free them after they are no longer needed, memory usage will increase over time.

The following API creates a 10-KB String instance and returns it to the client. The difference with the previous example is that this instance is referenced by a static member, which means it's never available for collection.

```
private static ConcurrentBag<string> _staticStrings = new ConcurrentBag<string>();

[HttpGet("staticstring")]
public ActionResult<string> GetStaticString()
{
    var bigString = new String('x', 10 * 1024);
    _staticStrings.Add(bigString);
    return bigString;
}
```

The preceding code:

- Is an example of a typical memory leak.
- With frequent calls, causes app memory to increases until the process crashes with an `OutOfMemory` exception.

In the preceding image:

- Load testing the `/api/staticstring` endpoint causes a linear increase in memory.
- The GC tries to free memory as the memory pressure grows, by calling a generation 2 collection.
- The GC cannot free the leaked memory. Allocated and working set increase with time.

Some scenarios, such as caching, require object references to be held until memory pressure forces them to be released. The WeakReference class can be used for this type of caching code. A `WeakReference` object is collected under memory pressures. The default implementation of IMemoryCache uses `WeakReference`.

Native memory

Some .NET Core objects rely on native memory. Native memory can **not** be collected by the GC. The .NET object using native memory must free it using native code.

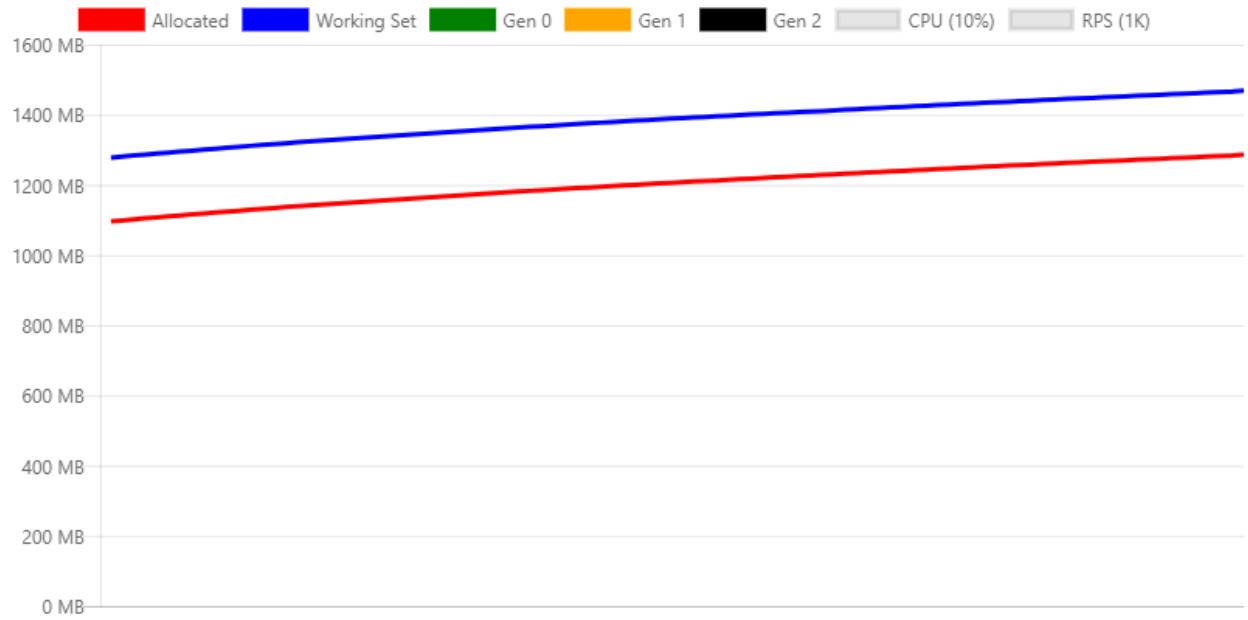
.NET provides the IDisposable interface to let developers release native memory. Even if Dispose is not called, correctly implemented classes call `Dispose` when the finalizer runs.

Consider the following code:

```
[HttpGet("fileprovider")]
public void GetFileProvider()
{
    var fp = new PhysicalFileProvider(TempPath);
    fp.Watch("*.*");
}
```

PhysicalFileProvider is a managed class, so any instance will be collected at the end of the request.

The following image shows the memory profile while invoking the `fileprovider` API continuously.



The preceding chart shows an obvious issue with the implementation of this class, as it keeps increasing memory usage. This is a known problem that is being tracked in this issue.

The same leak could be happen in user code, by one of the following:

- Not releasing the class correctly.
- Forgetting to invoke the `Dispose` method of the dependent objects that should be disposed.

Large objects heap

Frequent memory allocation/free cycles can fragment memory, especially when allocating large chunks of memory. Objects are allocated in contiguous blocks of memory. To mitigate fragmentation, when the GC frees memory, it tries to defragment it. This process is called **compaction**. Compaction involves moving objects. Moving large objects imposes a performance penalty. For this reason the GC creates a special memory zone for *large* objects, called the large object heap (LOH). Objects that are greater than 85,000 bytes (approximately 83 KB) are:

- Placed on the LOH.
- Not compacted.
- Collected during generation 2 GCs.

When the LOH is full, the GC will trigger a generation 2 collection. Generation 2 collections:

- Are inherently slow.
- Additionally incur the cost of triggering a collection on all other generations.

The following code compacts the LOH immediately:

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

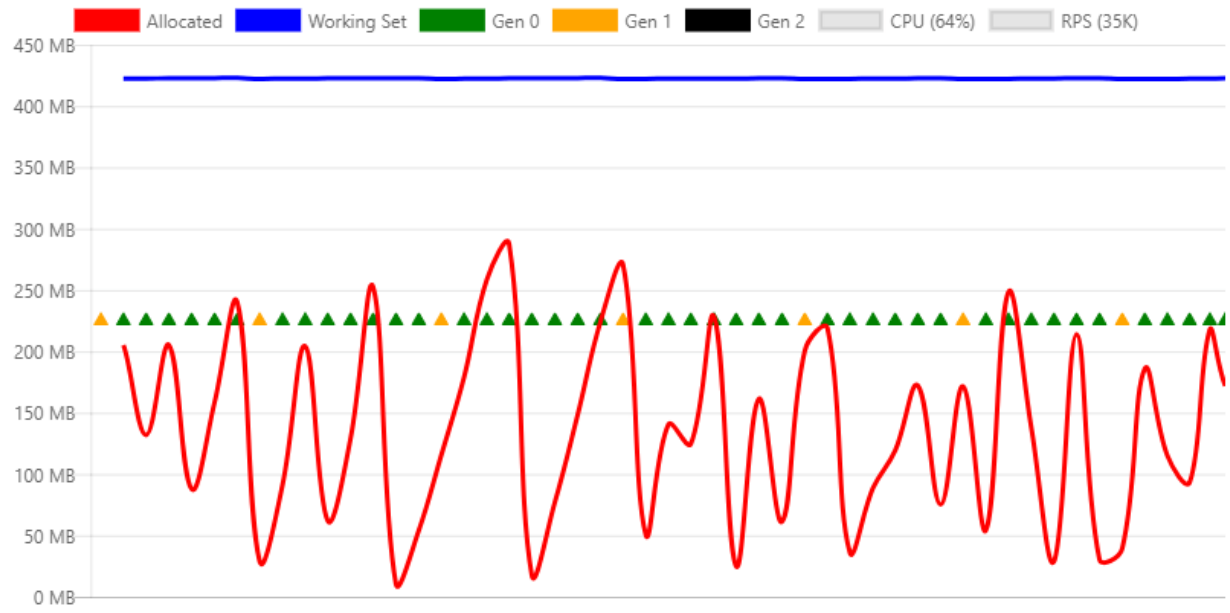
See LargeObjectHeapCompactionMode for information on compacting the LOH.

In containers using .NET Core 3.0 and later, the LOH is automatically compacted.

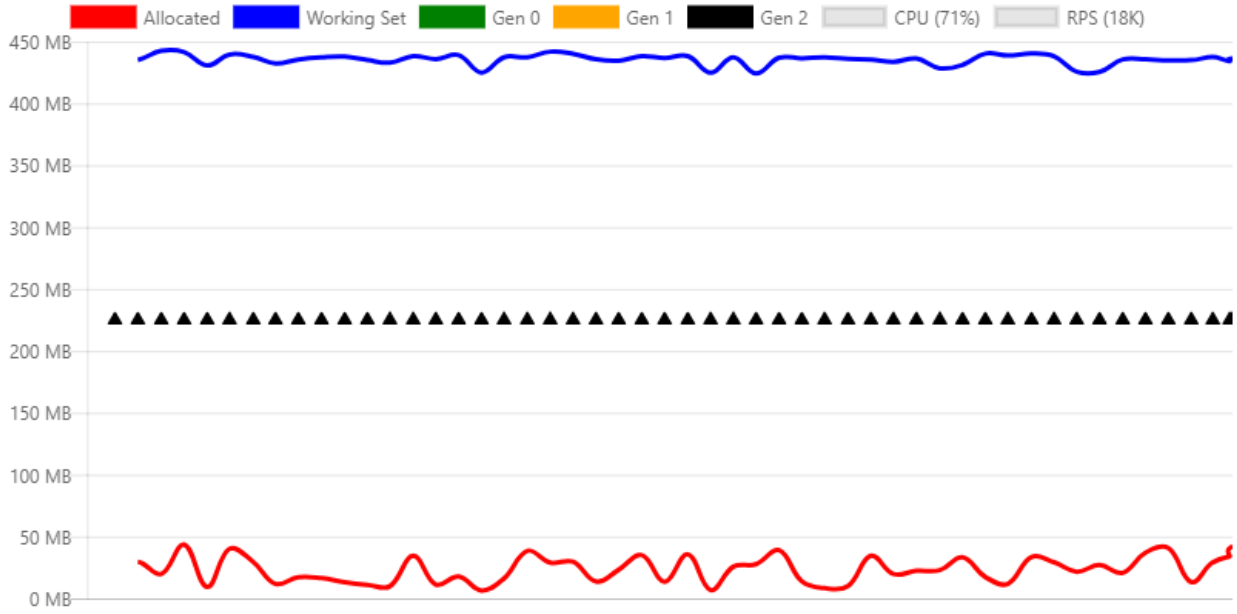
The following API that illustrates this behavior:

```
[HttpGet("loh/{size=85000}")]
public int GetLOH1(int size)
{
   return new byte[size].Length;
}
```

The following chart shows the memory profile of calling the `/api/loh/84975` endpoint, under maximum load:



The following chart shows the memory profile of calling the `/api/loh/84976` endpoint, allocating *just one more byte*:



Note: The `byte[]` structure has overhead bytes. That's why 84,976 bytes triggers the 85,000 limit.

Comparing the two preceding charts:

- The working set is similar for both scenarios, about 450 MB.
- The under LOH requests (84,975 bytes) shows mostly generation 0 collections.
- The over LOH requests generate constant generation 2 collections. Generation 2 collections are expensive. More CPU is required and throughput drops almost 50%.

Temporary large objects are particularly problematic because they cause gen2 GCs.

For maximum performance, large object use should be minimized. If possible, split up large objects. For example, Response Caching middleware in ASP.NET Core split the cache entries into blocks less than 85,000 bytes.

The following links show the ASP.NET Core approach to keeping objects under the LOH limit:

- ResponseCaching/Streams/StreamUtilities.cs
- ResponseCaching/MemoryResponseCache.cs

For more information, see:

- Large Object Heap Uncovered
- Large object heap

HttpClient

Incorrectly using HttpClient can result in a resource leak. System resources, such as database connections, sockets, file handles, etc.:

- Are more scarce than memory.
- Are more problematic when leaked than memory.

Experienced .NET developers know to call Dispose on objects that implement IDisposable. Not disposing objects that implement `IDisposable` typically results in leaked memory or leaked system resources.

`HttpClient` implements `IDisposable`, but should **not** be disposed on every invocation. Rather, `HttpClient` should be reused.

The following endpoint creates and disposes a new `HttpClient` instance on every request:

```
[HttpGet("httpclient1")]
public async Task<int> GetHttpClient1(string url)
{
    using (var httpClient = new HttpClient())
    {
        var result = await httpClient.GetAsync(url);
        return (int)result.StatusCode;
    }
}
```

Under load, the following error messages are logged:

```
fail: Microsoft.AspNetCore.Server.Kestrel[13]
      Connection id "0HLG70PBE1CR1", Request id "0HLG70PBE1CR1:00000031":
      An unhandled exception was thrown by the application.
System.Net.Http.HttpRequestException: Only one usage of each socket address
    (protocol/network address/port) is normally permitted --->
    System.Net.Sockets.SocketException: Only one usage of each socket address
    (protocol/network address/port) is normally permitted
   at System.Net.Http.ConnectHelper.ConnectAsync(String host, Int32 port,
    CancellationToken cancellationToken)
```

Even though the `HttpClient` instances are disposed, the actual network connection takes some time to be released by the operating system. By continuously creating new connections, *ports exhaustion* occurs. Each client connection requires its own client port.

One way to prevent port exhaustion is to reuse the same `HttpClient` instance:

```
private static readonly HttpClient _httpClient = new HttpClient();

[HttpGet("httpclient2")]
public async Task<int> GetHttpClient2(string url)
{
    var result = await _httpClient.GetAsync(url);
    return (int)result.StatusCode;
}
```

The `HttpClient` instance is released when the app stops. This example shows that not every disposable resource should be disposed after each use.

See the following for a better way to handle the lifetime of an `HttpClient` instance:

- HttpClient and lifetime management
- HTTPClient factory blog

Object pooling

The previous example showed how the `HttpClient` instance can be made static and reused by all requests. Reuse prevents running out of resources.

Object pooling:

- Uses the reuse pattern.
- Is designed for objects that are expensive to create.

A pool is a collection of pre-initialized objects that can be reserved and released across threads. Pools can define allocation rules such as limits, predefined sizes, or growth rate.

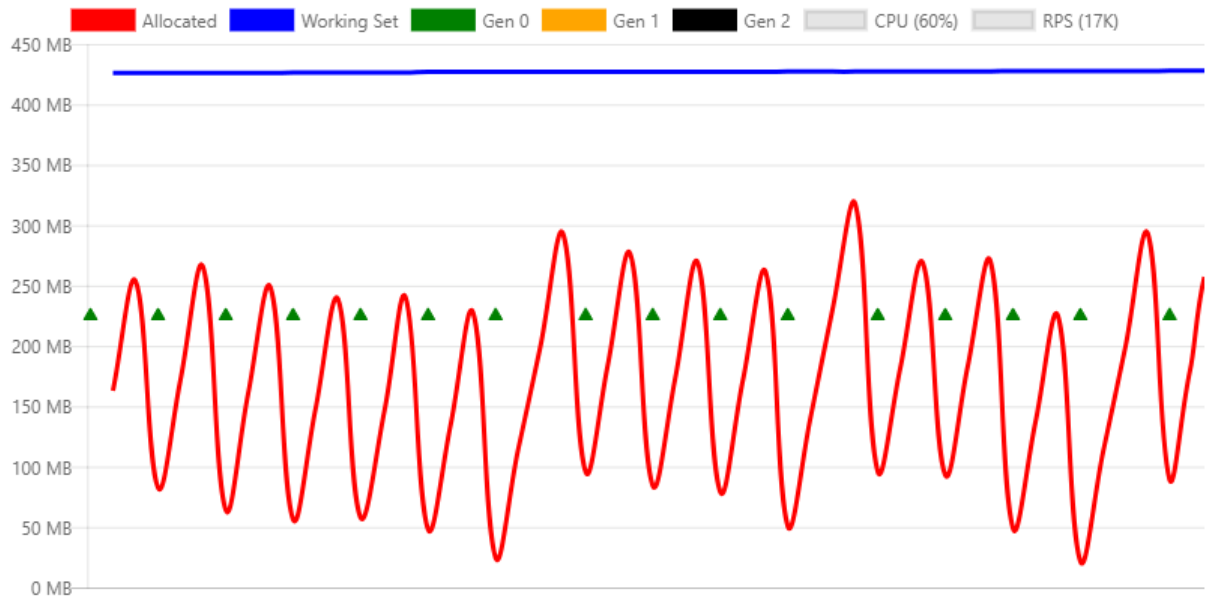
The NuGet package Microsoft.Extensions.ObjectPool contains classes that help to manage such pools.

The following API endpoint instantiates a `byte` buffer that is filled with random numbers on each request:

```
        [HttpGet("array/{size}")]
        public byte[] GetArray(int size)
        {
            var random = new Random();
            var array = new byte[size];
            random.NextBytes(array);

            return array;
        }
```

The following chart display calling the preceding API with moderate load:



In the preceding chart, generation 0 collections happen approximately once per second.

The preceding code can be optimized by pooling the `byte` buffer by using ArrayPool<T>. A static instance is reused across requests.

What's different with this approach is that a pooled object is returned from the API. That means:

- The object is out of your control as soon as you return from the method.
- You can't release the object.

To set up disposal of the object:

- Encapsulate the pooled array in a disposable object.
- Register the pooled object with HttpContext.Response.RegisterForDispose.

`RegisterForDispose` will take care of calling `Dispose` on the target object so that it's only released when the HTTP request is complete.

```
private static ArrayPool<byte> _arrayPool = ArrayPool<byte>.Create();

private class PooledArray : IDisposable
{
    public byte[] Array { get; private set; }

    public PooledArray(int size)
    {
        Array = _arrayPool.Rent(size);
    }

    public void Dispose()
    {
        _arrayPool.Return(Array);
    }
}

[HttpGet("pooledarray/{size}")]
public byte[] GetPooledArray(int size)
{
    var pooledArray = new PooledArray(size);

    var random = new Random();
    random.NextBytes(pooledArray.Array);

    HttpContext.Response.RegisterForDispose(pooledArray);

    return pooledArray.Array;
}
```

Applying the same load as the non-pooled version results in the following chart:

The main difference is allocated bytes, and as a consequence much fewer generation 0 collections.

Additional resources

- Garbage Collection
- Understanding different GC modes with Concurrency Visualizer
- Large Object Heap Uncovered
- Large object heap

Response caching in ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

By John Luo, Rick Anderson, and Steve Smith

View or download sample code (how to download)

Response caching reduces the number of requests a client or proxy makes to a web server. Response caching also reduces the amount of work the web server performs to generate a response. Response caching is controlled by headers that specify how you want client, proxy, and middleware to cache responses.

The ResponseCache attribute participates in setting response caching headers. Clients and intermediate proxies should honor the headers for caching responses under the HTTP 1.1 Caching specification.

For server-side caching that follows the HTTP 1.1 Caching specification, use Response Caching Middleware. The middleware can use the ResponseCacheAttribute properties to influence server-side caching behavior.

HTTP-based response caching

The HTTP 1.1 Caching specification describes how Internet caches should behave. The primary HTTP header used for caching is Cache-Control, which is used to specify cache *directives*. The directives control caching behavior as requests make their way from clients to servers and as responses make their way from servers back to clients. Requests and responses move through proxy servers, and proxy servers must also conform to the HTTP 1.1 Caching specification.

Common `Cache-Control` directives are shown in the following table.

DIRECTIVE	ACTION
public	A cache may store the response.
private	The response must not be stored by a shared cache. A private cache may store and reuse the response.
max-age	The client doesn't accept a response whose age is greater than the specified number of seconds. Examples: `max-age=60` (60 seconds), `max-age=2592000` (1 month)
no-cache	**On requests**: A cache must not use a stored response to satisfy the request. The origin server regenerates the response for the client, and the middleware updates the stored response in its cache. **On responses**: The response must not be used for a subsequent request without validation on the origin server.
no-store	**On requests**: A cache must not store the request. **On responses**: A cache must not store any part of the response.

Other cache headers that play a role in caching are shown in the following table.

HEADER	FUNCTION
Age	An estimate of the amount of time in seconds since the response was generated or successfully validated at the origin server.
Expires	The time after which the response is considered stale.
Pragma	Exists for backwards compatibility with HTTP/1.0 caches for setting `no-cache` behavior. If the `Cache-Control` header is present, the `Pragma` header is ignored.
Vary	Specifies that a cached response must not be sent unless all of the `Vary` header fields match in both the cached response's original request and the new request.

HTTP-based caching respects request Cache-Control directives

The HTTP 1.1 Caching specification for the Cache-Control header requires a cache to honor a valid `Cache-Control` header sent by the client. A client can make requests with a `no-cache` header value and force the server to generate a new response for every request.

Always honoring client `Cache-Control` request headers makes sense if you consider the goal of HTTP caching. Under the official specification, caching is meant to reduce the latency and network overhead of satisfying requests across a network of clients, proxies, and servers. It isn't necessarily a way to control the load on an origin server.

There's no developer control over this caching behavior when using the Response Caching Middleware because the middleware adheres to the official caching specification. Planned enhancements to the middleware are an opportunity to configure the middleware to ignore a request's `Cache-Control` header when deciding to serve a cached response. Planned enhancements provide an opportunity to better control server load.

Other caching technology in ASP.NET Core

In-memory caching

In-memory caching uses server memory to store cached data. This type of caching is suitable for a single server or multiple servers using *sticky sessions*. Sticky sessions means that the requests from a client are always routed to the same server for processing.

For more information, see Cache in-memory in ASP.NET Core.

Distributed Cache

Use a distributed cache to store data in memory when the app is hosted in a cloud or server farm. The cache is shared across the servers that process requests. A client can submit a request that's handled by any server in the group if cached data for the client is available. ASP.NET Core works with SQL Server, Redis, and NCache distributed caches.

For more information, see Distributed caching in ASP.NET Core.

Cache Tag Helper

Cache the content from an MVC view or Razor Page with the Cache Tag Helper. The Cache Tag Helper uses in-memory caching to store data.

For more information, see Cache Tag Helper in ASP.NET Core MVC.

Distributed Cache Tag Helper

Cache the content from an MVC view or Razor Page in distributed cloud or web farm scenarios with the Distributed Cache Tag Helper. The Distributed Cache Tag Helper uses SQL Server, Redis, or NCache to store data.

For more information, see Distributed Cache Tag Helper in ASP.NET Core.

ResponseCache attribute

The ResponseCacheAttribute specifies the parameters necessary for setting appropriate headers in response caching.

> **WARNING**
> Disable caching for content that contains information for authenticated clients. Caching should only be enabled for content that doesn't change based on a user's identity or whether a user is signed in.

VaryByQueryKeys varies the stored response by the values of the given list of query keys. When a single value of `*` is provided, the middleware varies responses by all request query string parameters.

Response Caching Middleware must be enabled to set the VaryByQueryKeys property. Otherwise, a runtime exception is thrown. There isn't a corresponding HTTP header for the VaryByQueryKeys property. The property is an HTTP feature handled by Response Caching Middleware. For the middleware to serve a cached response, the query string and query string value must match a previous request. For example, consider the sequence of requests and results shown in the following table.

REQUEST	RESULT
`http://example.com?key1=value1`	Returned from the server.
`http://example.com?key1=value1`	Returned from middleware.
`http://example.com?key1=value2`	Returned from the server.

The first request is returned by the server and cached in middleware. The second request is returned by middleware because the query string matches the previous request. The third request isn't in the middleware cache because the query string value doesn't match a previous request.

The ResponseCacheAttribute is used to configure and create (via IFilterFactory) a `Microsoft.AspNetCore.Mvc.Internal.ResponseCacheFilter`. The `ResponseCacheFilter` performs the work of updating the appropriate HTTP headers and features of the response. The filter:

- Removes any existing headers for `Vary`, `Cache-Control`, and `Pragma`.
- Writes out the appropriate headers based on the properties set in the ResponseCacheAttribute.
- Updates the response caching HTTP feature if VaryByQueryKeys is set.

Vary

This header is only written when the VaryByHeader property is set. The property set to the `Vary` property's value. The following sample uses the VaryByHeader property:

```
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public class Cache1Model : PageModel
{
```

Using the sample app, view the response headers with the browser's network tools. The following response

headers are sent with the Cache1 page response:

```
Cache-Control: public,max-age=30
Vary: User-Agent
```

NoStore and Location.None

NoStore overrides most of the other properties. When this property is set to `true`, the `Cache-Control` header is set to `no-store`. If Location is set to `None`:

- `Cache-Control` is set to `no-store,no-cache`.
- `Pragma` is set to `no-cache`.

If NoStore is `false` and Location is `None`, `Cache-Control`, and `Pragma` are set to `no-cache`.

NoStore is typically set to `true` for error pages. The Cache2 page in the sample app produces response headers that instruct the client not to store the response.

```
[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
public class Cache2Model : PageModel
{
```

The sample app returns the Cache2 page with the following headers:

```
Cache-Control: no-store,no-cache
Pragma: no-cache
```

Location and Duration

To enable caching, Duration must be set to a positive value and Location must be either `Any` (the default) or `Client`. The framework sets the `Cache-Control` header to the location value followed by the `max-age` of the response.

Location's options of `Any` and `Client` translate into `Cache-Control` header values of `public` and `private`, respectively. As noted in the NoStore and Location.None section, setting Location to `None` sets both `Cache-Control` and `Pragma` headers to `no-cache`.

`Location.Any` (`Cache-Control` set to `public`) indicates that the *client or any intermediate proxy* may cache the value, including Response Caching Middleware.

`Location.Client` (`Cache-Control` set to `private`) indicates that *only the client* may cache the value. No intermediate cache should cache the value, including Response Caching Middleware.

Cache control headers merely provide guidance to clients and intermediary proxies when and how to cache responses. There's no guarantee that clients and proxies will honor the HTTP 1.1 Caching specification. Response Caching Middleware always follows the caching rules laid out by the specification.

The following example shows the Cache3 page model from the sample app and the headers produced by setting Duration and leaving the default Location value:

```
[ResponseCache(Duration = 10, Location = ResponseCacheLocation.Any, NoStore = false)]
public class Cache3Model : PageModel
{
```

The sample app returns the Cache3 page with the following header:

```
Cache-Control: public,max-age=10
```

Cache profiles

Instead of duplicating response cache settings on many controller action attributes, cache profiles can be configured as options when setting up MVC/Razor Pages in `Startup.ConfigureServices`. Values found in a referenced cache profile are used as the defaults by the ResponseCacheAttribute and are overridden by any properties specified on the attribute.

Set up a cache profile. The following example shows a 30 second cache profile in the sample app's `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Default30",
            new CacheProfile()
            {
                Duration = 30
            });
    });
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("Default30",
            new CacheProfile()
            {
                Duration = 30
            });
    }).SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

The sample app's Cache4 page model references the `Default30` cache profile:

```
[ResponseCache(CacheProfileName = "Default30")]
public class Cache4Model : PageModel
{
```

The ResponseCacheAttribute can be applied to:

- Razor Pages: Attributes can't be applied to handler methods.
- MVC controllers.
- MVC action methods: Method-level attributes override the settings specified in class-level attributes.

The resulting header applied to the Cache4 page response by the `Default30` cache profile:

```
Cache-Control: public,max-age=30
```

Additional resources

- Storing Responses in Caches

- Cache-Control
- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Cache in-memory in ASP.NET Core

9/22/2020 • 21 minutes to read • Edit Online

By Rick Anderson, John Luo, and Steve Smith

View or download sample code (how to download)

Caching basics

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently **and** is expensive to generate. Caching makes a copy of data that can be returned much faster than from the source. Apps should be written and tested to **never** depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the IMemoryCache. `IMemoryCache` represents a cache stored in the memory of the web server. Apps running on a server farm (multiple servers) should ensure sessions are sticky when using the in-memory cache. Sticky sessions ensure that subsequent requests from a client all go to the same server. For example, Azure Web apps use Application Request Routing (ARR) to route all subsequent requests to the same server.

Non-sticky sessions in a web farm require a distributed cache to avoid cache consistency problems. For some apps, a distributed cache can support higher scale-out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

The in-memory cache can store any object. The distributed cache interface is limited to `byte[]`. The in-memory and distributed cache store cache items as key-value pairs.

System.Runtime.Caching/MemoryCache

System.Runtime.Caching/MemoryCache (NuGet package) can be used with:

- .NET Standard 2.0 or later.
- Any .NET implementation that targets .NET Standard 2.0 or later. For example, ASP.NET Core 2.0 or later.
- .NET Framework 4.5 or later.

Microsoft.Extensions.Caching.Memory/ `IMemoryCache` (described in this article) is recommended over `System.Runtime.Caching` / `MemoryCache` because it's better integrated into ASP.NET Core. For example, `IMemoryCache` works natively with ASP.NET Core dependency injection.

Use `System.Runtime.Caching` / `MemoryCache` as a compatibility bridge when porting code from ASP.NET 4.x to ASP.NET Core.

Cache guidelines

- Code should always have a fallback option to fetch data and **not** depend on a cached value being available.
- The cache uses a scarce resource, memory. Limit cache growth:
  - Do **not** use external input as cache keys.
  - Use expirations to limit cache growth.
  - Use SetSize, Size, and SizeLimit to limit cache size. The ASP.NET Core runtime does **not** limit cache size based on memory pressure. It's up to the developer to limit cache size.

Use IMemoryCache

> **WARNING**
>
> Using a *shared* memory cache from Dependency Injection and calling `SetSize`, `Size`, or `SizeLimit` to limit cache size can cause the app to fail. When a size limit is set on a cache, all entries must specify a size when being added. This can lead to issues since developers may not have full control on what uses the shared cache. For example, Entity Framework Core uses the shared cache and does not specify a size. If an app sets a cache size limit and uses EF Core, the app throws an `InvalidOperationException`. When using `SetSize`, `Size`, or `SizeLimit` to limit cache, create a cache singleton for caching. For more information and an example, see Use SetSize, Size, and SizeLimit to limit cache size. A shared cache is one shared by other frameworks or libraries. For example, EF Core uses the shared cache and does not specify a size.

In-memory caching is a *service* that's referenced from an app using Dependency Injection. Request the `IMemoryCache` instance in the constructor:

```
public class HomeController : Controller
{
    private IMemoryCache _cache;

    public HomeController(IMemoryCache memoryCache)
    {
        _cache = memoryCache;
    }
```

The following code uses TryGetValue to check if a time is in the cache. If a time isn't cached, a new entry is created and added to the cache with Set. The `CacheKeys` class is part of the download sample.

```
public static class CacheKeys
{
    public static string Entry { get { return "_Entry"; } }
    public static string CallbackEntry { get { return "_Callback"; } }
    public static string CallbackMessage { get { return "_CallbackMessage"; } }
    public static string Parent { get { return "_Parent"; } }
    public static string Child { get { return "_Child"; } }
    public static string DependentMessage { get { return "_DependentMessage"; } }
    public static string DependentCTS { get { return "_DependentCTS"; } }
    public static string Ticks { get { return "_Ticks"; } }
    public static string CancelMsg { get { return "_CancelMsg"; } }
    public static string CancelTokenSource { get { return "_CancelTokenSource"; } }
}
```

```
public IActionResult CacheTryGetValueSet()
{
    DateTime cacheEntry;

    // Look for cache key.
    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now;

        // Set cache options.
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Save data in cache.
        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }

    return View("Cache", cacheEntry);
}
```

The current time and the cached time are displayed:

```
@model DateTime?

<div>
    <h2>Actions</h2>
    <ul>
        <li><a asp-controller="Home" asp-action="CacheTryGetValueSet">TryGetValue and Set</a></li>
        <li><a asp-controller="Home" asp-action="CacheGet">Get</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreate">GetOrCreate</a></li>
        <li><a asp-controller="Home" asp-
action="CacheGetOrCreateAsynchronous">CacheGetOrCreateAsynchronous</a></li>
        <li><a asp-controller="Home" asp-action="CacheRemove">Remove</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAbs">CacheGetOrCreateAbs</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAbsSliding">CacheGetOrCreateAbsSliding</a>
</li>

    </ul>
</div>

<h3>Current Time: @DateTime.Now.TimeOfDay.ToString()</h3>
<h3>Cached Time: @(Model == null ? "No cached entry found" : Model.Value.TimeOfDay.ToString())</h3>
```

The cached `DateTime` value remains in the cache while there are requests within the timeout period.

The following code uses GetOrCreate and GetOrCreateAsync to cache data.

```
public IActionResult CacheGetOrCreate()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}

public async Task<IActionResult> CacheGetOrCreateAsynchronous()
{
    var cacheEntry = await
        _cache.GetOrCreateAsync(CacheKeys.Entry, entry =>
        {
            entry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return Task.FromResult(DateTime.Now);
        });

    return View("Cache", cacheEntry);
}
```

The following code calls Get to fetch the cached time:

```
public IActionResult CacheGet()
{
    var cacheEntry = _cache.Get<DateTime?>(CacheKeys.Entry);
    return View("Cache", cacheEntry);
}
```

The following code gets or creates a cached item with absolute expiration:

```
public IActionResult CacheGetOrCreateAbs()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromSeconds(10);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}
```

A cached item set with a sliding expiration only is at risk of becoming stale. If it's accessed more frequently than the sliding expiration interval, the item will never expire. Combine a sliding expiration with an absolute expiration to guarantee that the item expires once its absolute expiration time passes. The absolute expiration sets an upper bound to how long the item can be cached while still allowing the item to expire earlier if it isn't requested within the sliding expiration interval. When both absolute and sliding expiration are specified, the expirations are logically ORed. If either the sliding expiration interval *or* the absolute expiration time pass, the item is evicted from the cache.

The following code gets or creates a cached item with both sliding *and* absolute expiration:

```
public IActionResult CacheGetOrCreateAbsSliding()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.SetSlidingExpiration(TimeSpan.FromSeconds(3));
        entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromSeconds(20);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}
```

The preceding code guarantees the data will not be cached longer than the absolute time.

GetOrCreate, GetOrCreateAsync, and Get are extension methods in the CacheExtensions class. These methods extend the capability of IMemoryCache.

MemoryCacheEntryOptions

The following sample:

- Sets a sliding expiration time. Requests that access this cached item will reset the sliding expiration clock.
- Sets the cache priority to CacheItemPriority.NeverRemove.
- Sets a PostEvictionDelegate that will be called after the entry is evicted from the cache. The callback is run on a different thread from the code that removes the item from the cache.

```
public IActionResult CreateCallbackEntry()
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        // Pin to cache.
        .SetPriority(CacheItemPriority.NeverRemove)
        // Add eviction callback
        .RegisterPostEvictionCallback(callback: EvictionCallback, state: this);

    _cache.Set(CacheKeys.CallbackEntry, DateTime.Now, cacheEntryOptions);

    return RedirectToAction("GetCallbackEntry");
}

public IActionResult GetCallbackEntry()
{
    return View("Callback", new CallbackViewModel
    {
        CachedTime = _cache.Get<DateTime?>(CacheKeys.CallbackEntry),
        Message = _cache.Get<string>(CacheKeys.CallbackMessage)
    });
}

public IActionResult RemoveCallbackEntry()
{
    _cache.Remove(CacheKeys.CallbackEntry);
    return RedirectToAction("GetCallbackEntry");
}

private static void EvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.CallbackMessage, message);
}
```

Use SetSize, Size, and SizeLimit to limit cache size

A `MemoryCache` instance may optionally specify and enforce a size limit. The cache size limit does not have a defined unit of measure because the cache has no mechanism to measure the size of entries. If the cache size limit is set, all entries must specify size. The ASP.NET Core runtime does not limit cache size based on memory pressure. It's up to the developer to limit cache size. The size specified is in units the developer chooses.

For example:

- If the web app was primarily caching strings, each cache entry size could be the string length.
- The app could specify the size of all entries as 1, and the size limit is the count of entries.

If SizeLimit isn't set, the cache grows without bound. The ASP.NET Core runtime doesn't trim the cache when system memory is low. Apps must be architected to:

- Limit cache growth.
- Call Compact or Remove when available memory is limited:

The following code creates a unitless fixed size MemoryCache accessible by dependency injection:

```
// using Microsoft.Extensions.Caching.Memory;
public class MyMemoryCache
{
    public MemoryCache Cache { get; set; }
    public MyMemoryCache()
    {
        Cache = new MemoryCache(new MemoryCacheOptions
        {
            SizeLimit = 1024
        });
    }
}
```

`SizeLimit` does not have units. Cached entries must specify size in whatever units they deem most appropriate if the cache size limit has been set. All users of a cache instance should use the same unit system. An entry will not be cached if the sum of the cached entry sizes exceeds the value specified by `SizeLimit`. If no cache size limit is set, the cache size set on the entry will be ignored.

The following code registers `MyMemoryCache` with the dependency injection container.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    services.AddSingleton<MyMemoryCache>();
}
```

`MyMemoryCache` is created as an independent memory cache for components that are aware of this size limited cache and know how to set cache entry size appropriately.

The following code uses `MyMemoryCache`:

```
public class SetSize : PageModel
{
    private MemoryCache _cache;
    public static readonly string MyKey = "_MyKey";

    public SetSize(MyMemoryCache memoryCache)
    {
        _cache = memoryCache.Cache;
    }

    [TempData]
    public string DateTime_Now { get; set; }

    public IActionResult OnGet()
    {
        if (!_cache.TryGetValue(MyKey, out string cacheEntry))
        {
            // Key not in cache, so get data.
            cacheEntry = DateTime.Now.TimeOfDay.ToString();

            var cacheEntryOptions = new MemoryCacheEntryOptions()
                // Set cache entry size by extension method.
                .SetSize(1)
                // Keep in cache for this time, reset time if accessed.
                .SetSlidingExpiration(TimeSpan.FromSeconds(3));

            // Set cache entry size via property.
            // cacheEntryOptions.Size = 1;

            // Save data in cache.
            _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
        }

        DateTime_Now = cacheEntry;

        return RedirectToPage("./Index");
    }
}
```

The size of the cache entry can be set by Size or the SetSize extension methods:

```
public IActionResult OnGet()
{
    if (!_cache.TryGetValue(MyKey, out string cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now.TimeOfDay.ToString();

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Set cache entry size by extension method.
            .SetSize(1)
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Set cache entry size via property.
        // cacheEntryOptions.Size = 1;

        // Save data in cache.
        _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
    }

    DateTime_Now = cacheEntry;

    return RedirectToPage("./Index");
}
```

MemoryCache.Compact

`MemoryCache.Compact` attempts to remove the specified percentage of the cache in the following order:

- All expired items.
- Items by priority. Lowest priority items are removed first.
- Least recently used objects.
- Items with the earliest absolute expiration.
- Items with the earliest sliding expiration.

Pinned items with priority NeverRemove are never removed. The following code removes a cache item and calls `Compact`:

```
_cache.Remove(MyKey);

// Remove 33% of cached items.
_cache.Compact(.33);
cache_size = _cache.Count;
```

See Compact source on GitHub for more information.

Cache dependencies

The following sample shows how to expire a cache entry if a dependent entry expires. A CancellationChangeToken is added to the cached item. When `Cancel` is called on the `CancellationTokenSource`, both cache entries are evicted.

```
public IActionResult CreateDependentEntries()
{
    var cts = new CancellationTokenSource();
    _cache.Set(CacheKeys.DependentCTS, cts);

    using (var entry = _cache.CreateEntry(CacheKeys.Parent))
    {
        // expire this entry if the dependant entry expires.
        entry.Value = DateTime.Now;
        entry.RegisterPostEvictionCallback(DependentEvictionCallback, this);

        _cache.Set(CacheKeys.Child,
            DateTime.Now,
            new CancellationChangeToken(cts.Token));
    }

    return RedirectToAction("GetDependentEntries");
}

public IActionResult GetDependentEntries()
{
    return View("Dependent", new DependentViewModel
    {
        ParentCachedTime = _cache.Get<DateTime?>(CacheKeys.Parent),
        ChildCachedTime = _cache.Get<DateTime?>(CacheKeys.Child),
        Message = _cache.Get<string>(CacheKeys.DependentMessage)
    });
}

public IActionResult RemoveChildEntry()
{
    _cache.Get<CancellationTokenSource>(CacheKeys.DependentCTS).Cancel();
    return RedirectToAction("GetDependentEntries");
}

private static void DependentEvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Parent entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.DependentMessage, message);
}
```

Using a CancellationTokenSource allows multiple cache entries to be evicted as a group. With the `using` pattern in the code above, cache entries created inside the `using` block will inherit triggers and expiration settings.

Additional notes

- Expiration doesn't happen in the background. There is no timer that actively scans the cache for expired items. Any activity on the cache (`Get` , `Set` , `Remove`) can trigger a background scan for expired items. A timer on the `CancellationTokenSource` (CancelAfter) also removes the entry and trigger a scan for expired items. The following example uses CancellationTokenSource(TimeSpan) for the registered token. When this token fires it removes the entry immediately and fires the eviction callbacks:

```
public IActionResult CacheAutoExpiringTryGetValueSet()
{
    DateTime cacheEntry;

    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        cacheEntry = DateTime.Now;

        var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .AddExpirationToken(new CancellationChangeToken(cts.Token));

        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }

    return View("Cache", cacheEntry);
}
```

- When using a callback to repopulate a cache item:
  - Multiple requests can find the cached key value empty because the callback hasn't completed.
  - This can result in several threads repopulating the cached item.
- When one cache entry is used to create another, the child copies the parent entry's expiration tokens and time-based expiration settings. The child isn't expired by manual removal or updating of the parent entry.
- Use PostEvictionCallbacks to set the callbacks that will be fired after the cache entry is evicted from the cache.
- For most apps, `IMemoryCache` is enabled. For example, calling `AddMvc`, `AddControllersWithViews`, `AddRazorPages`, `AddMvcCore().AddRazorViewEngine`, and many other `Add{Service}` methods in `ConfigureServices`, enables `IMemoryCache`. For apps that are not calling one of the preceding `Add{Service}` methods, it may be necessary to call AddMemoryCache in `ConfigureServices`.

Background cache update

Use a background service such as IHostedService to update the cache. The background service can recompute the entries and then assign them to the cache only when they're ready.

Additional resources

- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

By Rick Anderson, John Luo, and Steve Smith

View or download sample code (how to download)

Caching basics

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently. Caching makes a copy of data that

can be returned much faster than from the original source. Code should be written and tested to **never** depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the IMemoryCache, which represents a cache stored in the memory of the web server. Apps that run on a server farm (multiple servers) should ensure that sessions are sticky when using the in-memory cache. Sticky sessions ensure that later requests from a client all go to the same server. For example, Azure Web apps use Application Request Routing (ARR) to route all requests from a user agent to the same server.

Non-sticky sessions in a web farm require a distributed cache to avoid cache consistency problems. For some apps, a distributed cache can support higher scale-out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

The in-memory cache can store any object. The distributed cache interface is limited to `byte[]`. The in-memory and distributed cache store cache items as key-value pairs.

System.Runtime.Caching/MemoryCache

System.Runtime.Caching/MemoryCache (NuGet package) can be used with:

- .NET Standard 2.0 or later.
- Any .NET implementation that targets .NET Standard 2.0 or later. For example, ASP.NET Core 2.0 or later.
- .NET Framework 4.5 or later.

Microsoft.Extensions.Caching.Memory/ `IMemoryCache` (described in this article) is recommended over `System.Runtime.Caching` / `MemoryCache` because it's better integrated into ASP.NET Core. For example, `IMemoryCache` works natively with ASP.NET Core dependency injection.

Use `System.Runtime.Caching` / `MemoryCache` as a compatibility bridge when porting code from ASP.NET 4.x to ASP.NET Core.

Cache guidelines

- Code should always have a fallback option to fetch data and **not** depend on a cached value being available.
- The cache uses a scarce resource, memory. Limit cache growth:
  - Do **not** use external input as cache keys.
  - Use expirations to limit cache growth.
  - Use SetSize, Size, and SizeLimit to limit cache size. The ASP.NET Core runtime does not limit cache size based on memory pressure. It's up to the developer to limit cache size.

Using IMemoryCache

> **WARNING**
> Using a *shared* memory cache from Dependency Injection and calling `SetSize`, `Size`, or `SizeLimit` to limit cache size can cause the app to fail. When a size limit is set on a cache, all entries must specify a size when being added. This can lead to issues since developers may not have full control on what uses the shared cache. For example, Entity Framework Core uses the shared cache and does not specify a size. If an app sets a cache size limit and uses EF Core, the app throws an `InvalidOperationException`. When using `SetSize`, `Size`, or `SizeLimit` to limit cache, create a cache singleton for caching. For more information and an example, see Use SetSize, Size, and SizeLimit to limit cache size.

In-memory caching is a *service* that's referenced from your app using Dependency Injection. Call `AddMemoryCache` in `ConfigureServices`:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMemoryCache();

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseMvcWithDefaultRoute();
    }
}
```

Request the `IMemoryCache` instance in the constructor:

```
public class HomeController : Controller
{
    private IMemoryCache _cache;

    public HomeController(IMemoryCache memoryCache)
    {
        _cache = memoryCache;
    }
```

`IMemoryCache` requires NuGet package Microsoft.Extensions.Caching.Memory, which is available in the Microsoft.AspNetCore.App metapackage.

The following code uses TryGetValue to check if a time is in the cache. If a time isn't cached, a new entry is created and added to the cache with Set.

```
public static class CacheKeys
{
    public static string Entry { get { return "_Entry"; } }
    public static string CallbackEntry { get { return "_Callback"; } }
    public static string CallbackMessage { get { return "_CallbackMessage"; } }
    public static string Parent { get { return "_Parent"; } }
    public static string Child { get { return "_Child"; } }
    public static string DependentMessage { get { return "_DependentMessage"; } }
    public static string DependentCTS { get { return "_DependentCTS"; } }
    public static string Ticks { get { return "_Ticks"; } }
    public static string CancelMsg { get { return "_CancelMsg"; } }
    public static string CancelTokenSource { get { return "_CancelTokenSource"; } }
}
```

```
public IActionResult CacheTryGetValueSet()
{
    DateTime cacheEntry;

    // Look for cache key.
    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now;

        // Set cache options.
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Save data in cache.
        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }

    return View("Cache", cacheEntry);
}
```

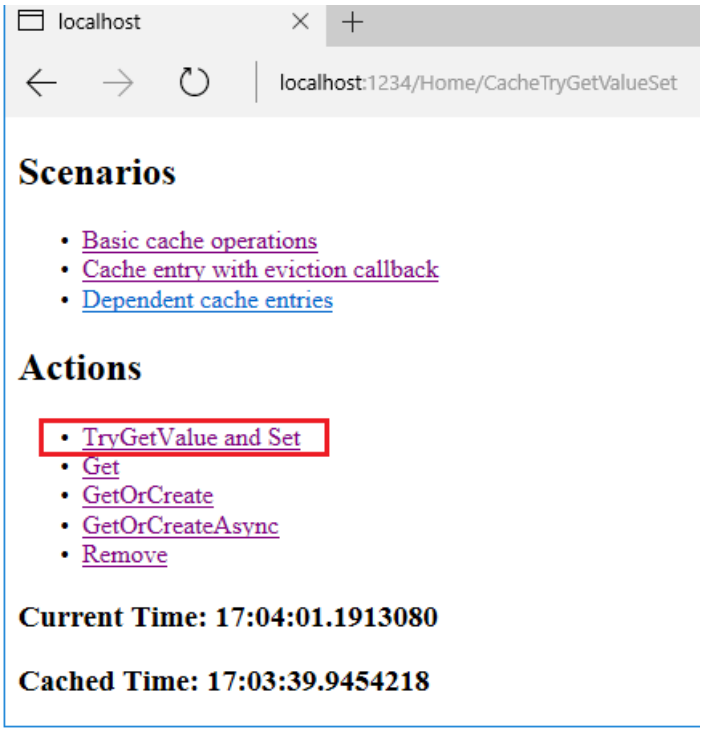
The current time and the cached time are displayed:

```
@model DateTime?

<div>
    <h2>Actions</h2>
    <ul>
        <li><a asp-controller="Home" asp-action="CacheTryGetValueSet">TryGetValue and Set</a></li>
        <li><a asp-controller="Home" asp-action="CacheGet">Get</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreate">GetOrCreate</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAsync">GetOrCreateAsync</a></li>
        <li><a asp-controller="Home" asp-action="CacheRemove">Remove</a></li>
    </ul>
</div>

<h3>Current Time: @DateTime.Now.TimeOfDay.ToString()</h3>
<h3>Cached Time: @(Model == null ? "No cached entry found" : Model.Value.TimeOfDay.ToString())</h3>
```

The cached `DateTime` value remains in the cache while there are requests within the timeout period. The following image shows the current time and an older time retrieved from the cache:



The following code uses GetOrCreate and GetOrCreateAsync to cache data.

```
public IActionResult CacheGetOrCreate()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}

public async Task<IActionResult> CacheGetOrCreateAsync()
{
    var cacheEntry = await
        _cache.GetOrCreateAsync(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return Task.FromResult(DateTime.Now);
    });

    return View("Cache", cacheEntry);
}
```

The following code calls Get to fetch the cached time:

```
public IActionResult CacheGet()
{
    var cacheEntry = _cache.Get<DateTime?>(CacheKeys.Entry);
    return View("Cache", cacheEntry);
}
```

GetOrCreate , GetOrCreateAsync, and Get are extension methods part of the CacheExtensions class that extends the capability of IMemoryCache. See IMemoryCache methods and CacheExtensions methods for a description of other cache methods.

MemoryCacheEntryOptions

The following sample:

- Sets a sliding expiration time. Requests that access this cached item will reset the sliding expiration clock.
- Sets the cache priority to `CacheItemPriority.NeverRemove`.
- Sets a PostEvictionDelegate that will be called after the entry is evicted from the cache. The callback is run on a different thread from the code that removes the item from the cache.

```
public IActionResult CreateCallbackEntry()
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        // Pin to cache.
        .SetPriority(CacheItemPriority.NeverRemove)
        // Add eviction callback
        .RegisterPostEvictionCallback(callback: EvictionCallback, state: this);

    _cache.Set(CacheKeys.CallbackEntry, DateTime.Now, cacheEntryOptions);

    return RedirectToAction("GetCallbackEntry");
}

public IActionResult GetCallbackEntry()
{
    return View("Callback", new CallbackViewModel
    {
        CachedTime = _cache.Get<DateTime?>(CacheKeys.CallbackEntry),
        Message = _cache.Get<string>(CacheKeys.CallbackMessage)
    });
}

public IActionResult RemoveCallbackEntry()
{
    _cache.Remove(CacheKeys.CallbackEntry);
    return RedirectToAction("GetCallbackEntry");
}

private static void EvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.CallbackMessage, message);
}
```

Use SetSize, Size, and SizeLimit to limit cache size

A `MemoryCache` instance may optionally specify and enforce a size limit. The cache size limit does not have a defined unit of measure because the cache has no mechanism to measure the size of entries. If the cache size limit is set, all entries must specify size. The ASP.NET Core runtime does not limit cache size based on memory pressure. It's up to the developer to limit cache size. The size specified is in units the developer chooses.

For example:

- If the web app was primarily caching strings, each cache entry size could be the string length.
- The app could specify the size of all entries as 1, and the size limit is the count of entries.

If SizeLimit is not set, the cache grows without bound. The ASP.NET Core runtime does not trim the cache when system memory is low. Apps much be architected to:

- Limit cache growth.
- Call Compact or Remove when available memory is limited:

The following code creates a unitless fixed size MemoryCache accessible by dependency injection:

```
// using Microsoft.Extensions.Caching.Memory;
public class MyMemoryCache
{
    public MemoryCache Cache { get; set; }
    public MyMemoryCache()
    {
        Cache = new MemoryCache(new MemoryCacheOptions
        {
            SizeLimit = 1024
        });
    }
}
```

`SizeLimit` does not have units. Cached entries must specify size in whatever units they deem most appropriate if the cache size limit has been set. All users of a cache instance should use the same unit system. An entry will not be cached if the sum of the cached entry sizes exceeds the value specified by `SizeLimit`. If no cache size limit is set, the cache size set on the entry will be ignored.

The following code registers `MyMemoryCache` with the dependency injection container.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddSingleton<MyMemoryCache>();
}
```

`MyMemoryCache` is created as an independent memory cache for components that are aware of this size limited cache and know how to set cache entry size appropriately.

The following code uses `MyMemoryCache`:

```
public class AboutModel : PageModel
{
    private MemoryCache _cache;
    public static readonly string MyKey = "_MyKey";

    public AboutModel(MyMemoryCache memoryCache)
    {
        _cache = memoryCache.Cache;
    }

    [TempData]
    public string DateTime_Now { get; set; }

    public IActionResult OnGet()
    {
        if (!_cache.TryGetValue(MyKey, out string cacheEntry))
        {
            // Key not in cache, so get data.
            cacheEntry = DateTime.Now.TimeOfDay.ToString();

            var cacheEntryOptions = new MemoryCacheEntryOptions()
                // Set cache entry size by extension method.
                .SetSize(1)
                // Keep in cache for this time, reset time if accessed.
                .SetSlidingExpiration(TimeSpan.FromSeconds(3));

            // Set cache entry size via property.
            // cacheEntryOptions.Size = 1;

            // Save data in cache.
            _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
        }

        DateTime_Now = cacheEntry;

        return RedirectToPage("./Index");
    }
}
```

The size of the cache entry can be set by Size or the SetSize extension method:

```
public IActionResult OnGet()
{
    if (!_cache.TryGetValue(MyKey, out string cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now.TimeOfDay.ToString();

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Set cache entry size by extension method.
            .SetSize(1)
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Set cache entry size via property.
        // cacheEntryOptions.Size = 1;

        // Save data in cache.
        _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
    }

    DateTime_Now = cacheEntry;

    return RedirectToPage("./Index");
}
```

MemoryCache.Compact

`MemoryCache.Compact` attempts to remove the specified percentage of the cache in the following order:

- All expired items.
- Items by priority. Lowest priority items are removed first.
- Least recently used objects.
- Items with the earliest absolute expiration.
- Items with the earliest sliding expiration.

Pinned items with priority NeverRemove are never removed.

```
_cache.Remove(MyKey);

// Remove 33% of cached items.
_cache.Compact(.33);
cache_size = _cache.Count;
```

See Compact source on GitHub for more information.

Cache dependencies

The following sample shows how to expire a cache entry if a dependent entry expires. A CancellationChangeToken is added to the cached item. When `Cancel` is called on the `CancellationTokenSource`, both cache entries are evicted.

```
public IActionResult CreateDependentEntries()
{
    var cts = new CancellationTokenSource();
    _cache.Set(CacheKeys.DependentCTS, cts);

    using (var entry = _cache.CreateEntry(CacheKeys.Parent))
    {
        // expire this entry if the dependant entry expires.
        entry.Value = DateTime.Now;
        entry.RegisterPostEvictionCallback(DependentEvictionCallback, this);

        _cache.Set(CacheKeys.Child,
            DateTime.Now,
            new CancellationChangeToken(cts.Token));
    }

    return RedirectToAction("GetDependentEntries");
}

public IActionResult GetDependentEntries()
{
    return View("Dependent", new DependentViewModel
    {
        ParentCachedTime = _cache.Get<DateTime?>(CacheKeys.Parent),
        ChildCachedTime = _cache.Get<DateTime?>(CacheKeys.Child),
        Message = _cache.Get<string>(CacheKeys.DependentMessage)
    });
}

public IActionResult RemoveChildEntry()
{
    _cache.Get<CancellationTokenSource>(CacheKeys.DependentCTS).Cancel();
    return RedirectToAction("GetDependentEntries");
}

private static void DependentEvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Parent entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.DependentMessage, message);
}
```

Using a `CancellationTokenSource` allows multiple cache entries to be evicted as a group. With the `using` pattern in the code above, cache entries created inside the `using` block will inherit triggers and expiration settings.

Additional notes

- When using a callback to repopulate a cache item:
  - Multiple requests can find the cached key value empty because the callback hasn't completed.
  - This can result in several threads repopulating the cached item.
- When one cache entry is used to create another, the child copies the parent entry's expiration tokens and time-based expiration settings. The child isn't expired by manual removal or updating of the parent entry.
- Use PostEvictionCallbacks to set the callbacks that will be fired after the cache entry is evicted from the cache.

Background cache update

Use a background service such as IHostedService to update the cache. The background service can recompute

the entries and then assign them to the cache only when they're ready.

Additional resources

- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Distributed caching in ASP.NET Core

9/22/2020 • 20 minutes to read • Edit Online

By Mohsin Nasir and Steve Smith

A distributed cache is a cache shared by multiple app servers, typically maintained as an external service to the app servers that access it. A distributed cache can improve the performance and scalability of an ASP.NET Core app, especially when the app is hosted by a cloud service or a server farm.

A distributed cache has several advantages over other caching scenarios where cached data is stored on individual app servers.

When cached data is distributed, the data:

- Is *coherent* (consistent) across requests to multiple servers.
- Survives server restarts and app deployments.
- Doesn't use local memory.

Distributed cache configuration is implementation specific. This article describes how to configure SQL Server and Redis distributed caches. Third party implementations are also available, such as NCache (NCache on GitHub). Regardless of which implementation is selected, the app interacts with the cache using the IDistributedCache interface.

View or download sample code (how to download)

Prerequisites

To use a SQL Server distributed cache, add a package reference to the Microsoft.Extensions.Caching.SqlServer package.

To use a Redis distributed cache, add a package reference to the Microsoft.Extensions.Caching.StackExchangeRedis package.

To use NCache distributed cache, add a package reference to the NCache.Microsoft.Extensions.Caching.OpenSource package.

IDistributedCache interface

The IDistributedCache interface provides the following methods to manipulate items in the distributed cache implementation:

- Get, GetAsync: Accepts a string key and retrieves a cached item as a `byte[]` array if found in the cache.
- Set, SetAsync: Adds an item (as `byte[]` array) to the cache using a string key.
- Refresh, RefreshAsync: Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).
- Remove, RemoveAsync: Removes a cache item based on its string key.

Establish distributed caching services

Register an implementation of IDistributedCache in `Startup.ConfigureServices`. Framework-provided implementations described in this topic include:

- Distributed Memory Cache

- Distributed SQL Server cache
- Distributed Redis cache
- Distributed NCache cache

Distributed Memory Cache

The Distributed Memory Cache (AddDistributedMemoryCache) is a framework-provided implementation of IDistributedCache that stores items in memory. The Distributed Memory Cache isn't an actual distributed cache. Cached items are stored by the app instance on the server where the app is running.

The Distributed Memory Cache is a useful implementation:

- In development and testing scenarios.
- When a single server is used in production and memory consumption isn't an issue. Implementing the Distributed Memory Cache abstracts cached data storage. It allows for implementing a true distributed caching solution in the future if multiple nodes or fault tolerance become necessary.

The sample app makes use of the Distributed Memory Cache when the app is run in the Development environment in `Startup.ConfigureServices`:

```
services.AddDistributedMemoryCache();
```

Distributed SQL Server Cache

The Distributed SQL Server Cache implementation (AddDistributedSqlServerCache) allows the distributed cache to use a SQL Server database as its backing store. To create a SQL Server cached item table in a SQL Server instance, you can use the `sql-cache` tool. The tool creates a table with the name and schema that you specify.

Create a table in SQL Server by running the `sql-cache create` command. Provide the SQL Server instance (`Data Source`), database (`Initial Catalog`), schema (for example, `dbo`), and table name (for example, `TestCache`):

```
dotnet sql-cache create "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=DistCache;Integrated
Security=True;" dbo TestCache
```

A message is logged to indicate that the tool was successful:

```
Table and index were created successfully.
```

The table created by the `sql-cache` tool has the following schema:

	Name	Data Type	Allow Nulls
🔑	Id	nvarchar(449)	☐
	Value	varbinary(MAX)	☐
	ExpiresAtTime	datetimeoffset(7)	☐
	SlidingExpirationInSeconds	bigint	☑
	AbsoluteExpiration	datetimeoffset(7)	☑

> **NOTE**
>
> An app should manipulate cache values using an instance of IDistributedCache, not a SqlServerCache.

The sample app implements SqlServerCache in a non-Development environment in `Startup.ConfigureServices`:

```
services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString =
        _config["DistCache_ConnectionString"];
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});
```

> **NOTE**
>
> A ConnectionString (and optionally, SchemaName and TableName) are typically stored outside of source control (for example, stored by the Secret Manager or in *appsettings.json*/*appsettings.{ENVIRONMENT}.json* files). The connection string may contain credentials that should be kept out of source control systems.

Distributed Redis Cache

Redis is an open source in-memory data store, which is often used as a distributed cache. You can configure an Azure Redis Cache for an Azure-hosted ASP.NET Core app, and use an Azure Redis Cache for local development.

An app configures the cache implementation using a RedisCache instance (AddStackExchangeRedisCache).

For more information, see Azure Cache for Redis.

See this GitHub issue for a discussion on alternative approaches to a local Redis cache.

Distributed NCache Cache

NCache is an open source in-memory distributed cache developed natively in .NET and .NET Core. NCache works both locally and configured as a distributed cache cluster for an ASP.NET Core app running in Azure or on other hosting platforms.

To install and configure NCache on your local machine, see NCache Getting Started Guide for Windows.

To configure NCache:

1. Install NCache open source NuGet.
2. Configure the cache cluster in client.ncconf.
3. Add the following code to `Startup.ConfigureServices`:

```
services.AddNCacheDistributedCache(configuration =>
{
    configuration.CacheName = "demoClusteredCache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});
```

Use the distributed cache

To use the IDistributedCache interface, request an instance of IDistributedCache from any constructor in the app. The instance is provided by dependency injection (DI).

When the sample app starts, IDistributedCache is injected into `Startup.Configure`. The current time is cached using IHostApplicationLifetime (for more information, see Generic Host: IHostApplicationLifetime):

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    IHostApplicationLifetime lifetime, IDistributedCache cache)
{
    lifetime.ApplicationStarted.Register(() =>
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        cache.Set("cachedTimeUTC", encodedCurrentTimeUTC, options);
    });
```

The sample app injects IDistributedCache into the `IndexModel` for use by the Index page.

Each time the Index page is loaded, the cache is checked for the cached time in `OnGetAsync`. If the cached time hasn't expired, the time is displayed. If 20 seconds have elapsed since the last time the cached time was accessed (the last time this page was loaded), the page displays *Cached Time Expired*.

Immediately update the cached time to the current time by selecting the **Reset Cached Time** button. The button triggers the `OnPostResetCachedTime` handler method.

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public string CachedTimeUTC { get; set; }

    public async Task OnGetAsync()
    {
        CachedTimeUTC = "Cached Time Expired";
        var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");

        if (encodedCachedTimeUTC != null)
        {
            CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
        }
    }

    public async Task<IActionResult> OnPostResetCachedTime()
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        await _cache.SetAsync("cachedTimeUTC", encodedCurrentTimeUTC, options);

        return RedirectToPage();
    }
}
```

> **NOTE**
>
> There's no need to use a Singleton or Scoped lifetime for IDistributedCache instances (at least for the built-in implementations).
>
> You can also create an IDistributedCache instance wherever you might need one instead of using DI, but creating an instance in code can make your code harder to test and violates the Explicit Dependencies Principle.

Recommendations

When deciding which implementation of IDistributedCache is best for your app, consider the following:

- Existing infrastructure
- Performance requirements
- Cost
- Team experience

Caching solutions usually rely on in-memory storage to provide fast retrieval of cached data, but memory is a limited resource and costly to expand. Only store commonly used data in a cache.

Generally, a Redis cache provides higher throughput and lower latency than a SQL Server cache. However, benchmarking is usually required to determine the performance characteristics of caching strategies.

When SQL Server is used as a distributed cache backing store, use of the same database for the cache and the app's ordinary data storage and retrieval can negatively impact the performance of both. We recommend using a dedicated SQL Server instance for the distributed cache backing store.

Additional resources

- Redis Cache on Azure
- SQL Database on Azure
- ASP.NET Core IDistributedCache Provider for NCache in Web Farms (NCache on GitHub)
- Cache in-memory in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core
- Host ASP.NET Core in a web farm

A distributed cache is a cache shared by multiple app servers, typically maintained as an external service to the app servers that access it. A distributed cache can improve the performance and scalability of an ASP.NET Core app, especially when the app is hosted by a cloud service or a server farm.

A distributed cache has several advantages over other caching scenarios where cached data is stored on individual app servers.

When cached data is distributed, the data:

- Is *coherent* (consistent) across requests to multiple servers.
- Survives server restarts and app deployments.
- Doesn't use local memory.

Distributed cache configuration is implementation specific. This article describes how to configure SQL Server

and Redis distributed caches. Third party implementations are also available, such as NCache (NCache on GitHub). Regardless of which implementation is selected, the app interacts with the cache using the IDistributedCache interface.

View or download sample code (how to download)

Prerequisites

To use a SQL Server distributed cache, reference the Microsoft.AspNetCore.App metapackage or add a package reference to the Microsoft.Extensions.Caching.SqlServer package.

To use a Redis distributed cache, reference the Microsoft.AspNetCore.App metapackage and add a package reference to the Microsoft.Extensions.Caching.StackExchangeRedis package. The Redis package isn't included in the `Microsoft.AspNetCore.App` package, so you must reference the Redis package separately in your project file.

To use NCache distributed cache, reference the Microsoft.AspNetCore.App metapackage and add a package reference to the NCache.Microsoft.Extensions.Caching.OpenSource package. The NCache package isn't included in the `Microsoft.AspNetCore.App` package, so you must reference the NCache package separately in your project file.

IDistributedCache interface

The IDistributedCache interface provides the following methods to manipulate items in the distributed cache implementation:

- Get, GetAsync: Accepts a string key and retrieves a cached item as a `byte[]` array if found in the cache.
- Set, SetAsync: Adds an item (as `byte[]` array) to the cache using a string key.
- Refresh, RefreshAsync: Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).
- Remove, RemoveAsync: Removes a cache item based on its string key.

Establish distributed caching services

Register an implementation of IDistributedCache in `Startup.ConfigureServices`. Framework-provided implementations described in this topic include:

- Distributed Memory Cache
- Distributed SQL Server cache
- Distributed Redis cache
- Distributed NCache cache

Distributed Memory Cache

The Distributed Memory Cache (AddDistributedMemoryCache) is a framework-provided implementation of IDistributedCache that stores items in memory. The Distributed Memory Cache isn't an actual distributed cache. Cached items are stored by the app instance on the server where the app is running.

The Distributed Memory Cache is a useful implementation:

- In development and testing scenarios.
- When a single server is used in production and memory consumption isn't an issue. Implementing the Distributed Memory Cache abstracts cached data storage. It allows for implementing a true distributed caching solution in the future if multiple nodes or fault tolerance become necessary.

The sample app makes use of the Distributed Memory Cache when the app is run in the Development

environment in `Startup.ConfigureServices`:

```
services.AddDistributedMemoryCache();
```

Distributed SQL Server Cache

The Distributed SQL Server Cache implementation (AddDistributedSqlServerCache) allows the distributed cache to use a SQL Server database as its backing store. To create a SQL Server cached item table in a SQL Server instance, you can use the `sql-cache` tool. The tool creates a table with the name and schema that you specify.

Create a table in SQL Server by running the `sql-cache create` command. Provide the SQL Server instance (`Data Source`), database (`Initial Catalog`), schema (for example, `dbo`), and table name (for example, `TestCache`):

```
dotnet sql-cache create "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=DistCache;Integrated
Security=True;" dbo TestCache
```

A message is logged to indicate that the tool was successful:

```
Table and index were created successfully.
```

The table created by the `sql-cache` tool has the following schema:

	Name	Data Type	Allow Nulls
🔑	Id	nvarchar(449)	☐
	Value	varbinary(MAX)	☐
	ExpiresAtTime	datetimeoffset(7)	☐
	SlidingExpirationInSeconds	bigint	☑
	AbsoluteExpiration	datetimeoffset(7)	☑

> **NOTE**
> An app should manipulate cache values using an instance of IDistributedCache, not a SqlServerCache.

The sample app implements SqlServerCache in a non-Development environment in `Startup.ConfigureServices`:

```
services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString =
        _config["DistCache_ConnectionString"];
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});
```

> **NOTE**
>
> A ConnectionString (and optionally, SchemaName and TableName) are typically stored outside of source control (for example, stored by the Secret Manager or in *appsettings.json*/*appsettings.{ENVIRONMENT}.json* files). The connection string may contain credentials that should be kept out of source control systems.

Distributed Redis Cache

Redis is an open source in-memory data store, which is often used as a distributed cache. You can use Redis locally, and you can configure an Azure Redis Cache for an Azure-hosted ASP.NET Core app.

An app configures the cache implementation using a RedisCache instance (AddStackExchangeRedisCache) in a non-Development environment in `Startup.ConfigureServices`:

```
services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "localhost";
    options.InstanceName = "SampleInstance";
});
```

To install Redis on your local machine:

1. Install the Chocolatey Redis package.
2. Run `redis-server` from a command prompt.

Distributed NCache Cache

NCache is an open source in-memory distributed cache developed natively in .NET and .NET Core. NCache works both locally and configured as a distributed cache cluster for an ASP.NET Core app running in Azure or on other hosting platforms.

To install and configure NCache on your local machine, see NCache Getting Started Guide for Windows.

To configure NCache:

1. Install NCache open source NuGet.
2. Configure the cache cluster in client.ncconf.
3. Add the following code to `Startup.ConfigureServices`:

```
services.AddNCacheDistributedCache(configuration =>
{
    configuration.CacheName = "demoClusteredCache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});
```

Use the distributed cache

To use the IDistributedCache interface, request an instance of IDistributedCache from any constructor in the app. The instance is provided by dependency injection (DI).

When the sample app starts, IDistributedCache is injected into `Startup.Configure`. The current time is cached using IApplicationLifetime (for more information, see Web Host: IApplicationLifetime interface):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IApplicationLifetime lifetime, IDistributedCache cache)
{
    lifetime.ApplicationStarted.Register(() =>
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        cache.Set("cachedTimeUTC", encodedCurrentTimeUTC, options);
    });
```

The sample app injects IDistributedCache into the `IndexModel` for use by the Index page.

Each time the Index page is loaded, the cache is checked for the cached time in `OnGetAsync`. If the cached time hasn't expired, the time is displayed. If 20 seconds have elapsed since the last time the cached time was accessed (the last time this page was loaded), the page displays *Cached Time Expired*.

Immediately update the cached time to the current time by selecting the **Reset Cached Time** button. The button triggers the `OnPostResetCachedTime` handler method.

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public string CachedTimeUTC { get; set; }

    public async Task OnGetAsync()
    {
        CachedTimeUTC = "Cached Time Expired";
        var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");

        if (encodedCachedTimeUTC != null)
        {
            CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
        }
    }

    public async Task<IActionResult> OnPostResetCachedTime()
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        await _cache.SetAsync("cachedTimeUTC", encodedCurrentTimeUTC, options);

        return RedirectToPage();
    }
}
```

> **NOTE**
>
> There's no need to use a Singleton or Scoped lifetime for IDistributedCache instances (at least for the built-in implementations).
>
> You can also create an IDistributedCache instance wherever you might need one instead of using DI, but creating an instance in code can make your code harder to test and violates the Explicit Dependencies Principle.

Recommendations

When deciding which implementation of IDistributedCache is best for your app, consider the following:

- Existing infrastructure
- Performance requirements
- Cost
- Team experience

Caching solutions usually rely on in-memory storage to provide fast retrieval of cached data, but memory is a limited resource and costly to expand. Only store commonly used data in a cache.

Generally, a Redis cache provides higher throughput and lower latency than a SQL Server cache. However, benchmarking is usually required to determine the performance characteristics of caching strategies.

When SQL Server is used as a distributed cache backing store, use of the same database for the cache and the app's ordinary data storage and retrieval can negatively impact the performance of both. We recommend using a dedicated SQL Server instance for the distributed cache backing store.

Additional resources

- Redis Cache on Azure
- SQL Database on Azure
- ASP.NET Core IDistributedCache Provider for NCache in Web Farms (NCache on GitHub)
- Cache in-memory in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core
- Host ASP.NET Core in a web farm

A distributed cache is a cache shared by multiple app servers, typically maintained as an external service to the app servers that access it. A distributed cache can improve the performance and scalability of an ASP.NET Core app, especially when the app is hosted by a cloud service or a server farm.

A distributed cache has several advantages over other caching scenarios where cached data is stored on individual app servers.

When cached data is distributed, the data:

- Is *coherent* (consistent) across requests to multiple servers.
- Survives server restarts and app deployments.
- Doesn't use local memory.

Distributed cache configuration is implementation specific. This article describes how to configure SQL Server

and Redis distributed caches. Third party implementations are also available, such as NCache (NCache on GitHub). Regardless of which implementation is selected, the app interacts with the cache using the IDistributedCache interface.

View or download sample code (how to download)

Prerequisites

To use a SQL Server distributed cache, reference the Microsoft.AspNetCore.App metapackage or add a package reference to the Microsoft.Extensions.Caching.SqlServer package.

To use a Redis distributed cache, reference the Microsoft.AspNetCore.App metapackage and add a package reference to the Microsoft.Extensions.Caching.Redis package. The Redis package isn't included in the `Microsoft.AspNetCore.App` package, so you must reference the Redis package separately in your project file.

To use NCache distributed cache, reference the Microsoft.AspNetCore.App metapackage and add a package reference to the NCache.Microsoft.Extensions.Caching.OpenSource package. The NCache package isn't included in the `Microsoft.AspNetCore.App` package, so you must reference the NCache package separately in your project file.

IDistributedCache interface

The IDistributedCache interface provides the following methods to manipulate items in the distributed cache implementation:

- Get, GetAsync: Accepts a string key and retrieves a cached item as a `byte[]` array if found in the cache.
- Set, SetAsync: Adds an item (as `byte[]` array) to the cache using a string key.
- Refresh, RefreshAsync: Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).
- Remove, RemoveAsync: Removes a cache item based on its string key.

Establish distributed caching services

Register an implementation of IDistributedCache in `Startup.ConfigureServices`. Framework-provided implementations described in this topic include:

- Distributed Memory Cache
- Distributed SQL Server cache
- Distributed Redis cache
- Distributed NCache cache

Distributed Memory Cache

The Distributed Memory Cache (AddDistributedMemoryCache) is a framework-provided implementation of IDistributedCache that stores items in memory. The Distributed Memory Cache isn't an actual distributed cache. Cached items are stored by the app instance on the server where the app is running.

The Distributed Memory Cache is a useful implementation:

- In development and testing scenarios.
- When a single server is used in production and memory consumption isn't an issue. Implementing the Distributed Memory Cache abstracts cached data storage. It allows for implementing a true distributed caching solution in the future if multiple nodes or fault tolerance become necessary.

The sample app makes use of the Distributed Memory Cache when the app is run in the Development environment in `Startup.ConfigureServices`:

```
services.AddDistributedMemoryCache();
```

Distributed SQL Server Cache

The Distributed SQL Server Cache implementation (AddDistributedSqlServerCache) allows the distributed cache to use a SQL Server database as its backing store. To create a SQL Server cached item table in a SQL Server instance, you can use the `sql-cache` tool. The tool creates a table with the name and schema that you specify.

Create a table in SQL Server by running the `sql-cache create` command. Provide the SQL Server instance (`Data Source`), database (`Initial Catalog`), schema (for example, `dbo`), and table name (for example, `TestCache`):

```
dotnet sql-cache create "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=DistCache;Integrated
Security=True;" dbo TestCache
```

A message is logged to indicate that the tool was successful:

```
Table and index were created successfully.
```

The table created by the `sql-cache` tool has the following schema:

	Name	Data Type	Allow Nulls
🔑	Id	nvarchar(449)	☐
	Value	varbinary(MAX)	☐
	ExpiresAtTime	datetimeoffset(7)	☐
	SlidingExpirationInSeconds	bigint	☑
	AbsoluteExpiration	datetimeoffset(7)	☑

> **NOTE**
> An app should manipulate cache values using an instance of IDistributedCache, not a SqlServerCache.

The sample app implements SqlServerCache in a non-Development environment in `Startup.ConfigureServices` :

```
services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString =
        _config["DistCache_ConnectionString"];
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});
```

> **NOTE**
> A ConnectionString (and optionally, SchemaName and TableName) are typically stored outside of source control (for example, stored by the Secret Manager or in *appsettings.json*/*appsettings.{ENVIRONMENT}.json* files). The connection string may contain credentials that should be kept out of source control systems.

Distributed Redis Cache

Redis is an open source in-memory data store, which is often used as a distributed cache. You can use Redis locally, and you can configure an Azure Redis Cache for an Azure-hosted ASP.NET Core app.

An app configures the cache implementation using a RedisCache instance (AddDistributedRedisCache):

```
services.AddDistributedRedisCache(options =>
{
    options.Configuration = "localhost";
    options.InstanceName = "SampleInstance";
});
```

To install Redis on your local machine:

1. Install the Chocolatey Redis package.
2. Run `redis-server` from a command prompt.

Distributed NCache Cache

NCache is an open source in-memory distributed cache developed natively in .NET and .NET Core. NCache works both locally and configured as a distributed cache cluster for an ASP.NET Core app running in Azure or on other hosting platforms.

To install and configure NCache on your local machine, see NCache Getting Started Guide for Windows.

To configure NCache:

1. Install NCache open source NuGet.
2. Configure the cache cluster in client.ncconf.
3. Add the following code to `Startup.ConfigureServices`:

```
services.AddNCacheDistributedCache(configuration =>
{
    configuration.CacheName = "demoClusteredCache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});
```

Use the distributed cache

To use the IDistributedCache interface, request an instance of IDistributedCache from any constructor in the app. The instance is provided by dependency injection (DI).

When the sample app starts, IDistributedCache is injected into `Startup.Configure`. The current time is cached using IApplicationLifetime (for more information, see Web Host: IApplicationLifetime interface):

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IApplicationLifetime lifetime, IDistributedCache cache)
{
    lifetime.ApplicationStarted.Register(() =>
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        cache.Set("cachedTimeUTC", encodedCurrentTimeUTC, options);
    });
```

The sample app injects IDistributedCache into the `IndexModel` for use by the Index page.

Each time the Index page is loaded, the cache is checked for the cached time in `OnGetAsync`. If the cached time hasn't expired, the time is displayed. If 20 seconds have elapsed since the last time the cached time was accessed (the last time this page was loaded), the page displays *Cached Time Expired*.

Immediately update the cached time to the current time by selecting the **Reset Cached Time** button. The button triggers the `OnPostResetCachedTime` handler method.

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public string CachedTimeUTC { get; set; }

    public async Task OnGetAsync()
    {
        CachedTimeUTC = "Cached Time Expired";
        var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");

        if (encodedCachedTimeUTC != null)
        {
            CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
        }
    }

    public async Task<IActionResult> OnPostResetCachedTime()
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        await _cache.SetAsync("cachedTimeUTC", encodedCurrentTimeUTC, options);

        return RedirectToPage();
    }
}
```

> **NOTE**
>
> There's no need to use a Singleton or Scoped lifetime for IDistributedCache instances (at least for the built-in implementations).
>
> You can also create an IDistributedCache instance wherever you might need one instead of using DI, but creating an instance in code can make your code harder to test and violates the Explicit Dependencies Principle.

Recommendations

When deciding which implementation of IDistributedCache is best for your app, consider the following:

- Existing infrastructure
- Performance requirements
- Cost
- Team experience

Caching solutions usually rely on in-memory storage to provide fast retrieval of cached data, but memory is a limited resource and costly to expand. Only store commonly used data in a cache.

Generally, a Redis cache provides higher throughput and lower latency than a SQL Server cache. However, benchmarking is usually required to determine the performance characteristics of caching strategies.

When SQL Server is used as a distributed cache backing store, use of the same database for the cache and the app's ordinary data storage and retrieval can negatively impact the performance of both. We recommend using a dedicated SQL Server instance for the distributed cache backing store.

Additional resources

- Redis Cache on Azure
- SQL Database on Azure
- ASP.NET Core IDistributedCache Provider for NCache in Web Farms (NCache on GitHub)
- Cache in-memory in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core
- Host ASP.NET Core in a web farm

Response Caching Middleware in ASP.NET Core

9/22/2020 • 14 minutes to read • Edit Online

By John Luo

This article explains how to configure Response Caching Middleware in an ASP.NET Core app. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the `[ResponseCache]` attribute, see Response Caching.

View or download sample code (how to download)

Configuration

Response Caching Middleware is implicitly available for ASP.NET Core apps via the shared framework.

In `Startup.ConfigureServices`, add the Response Caching Middleware to the service collection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCaching();
    services.AddRazorPages();
}
```

Configure the app to use the middleware with the UseResponseCaching extension method, which adds the middleware to the request processing pipeline in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseRouting();
    // UseCors must be called before UseResponseCaching
    // app.UseCors("myAllowSpecificOrigins");

    app.UseResponseCaching();

    app.Use(async (context, next) =>
    {
        context.Response.GetTypedHeaders().CacheControl =
            new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
            {
                Public = true,
                MaxAge = TimeSpan.FromSeconds(10)
            };
        context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
            new string[] { "Accept-Encoding" };

        await next();
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

WARNING

UseCors must be called before UseResponseCaching when using CORS middleware.

The sample app adds headers to control caching on subsequent requests:

- Cache-Control: Caches cacheable responses for up to 10 seconds.
- Vary: Configures the middleware to serve a cached response only if the Accept-Encoding header of subsequent requests matches that of the original request.

```
// using Microsoft.AspNetCore.Http;

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next();
});
```

The preceding headers are not written to the response and are overridden when a controller, action, or Razor Page:

- Has a [ResponseCache] attribute. This applies even if a property isn't set. For example, omitting the VaryByHeader property will cause the corresponding header to be removed from the response.

Response Caching Middleware only caches server responses that result in a 200 (OK) status code. Any other responses, including error pages, are ignored by the middleware.

> **WARNING**
> Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See Conditions for caching for details on how the middleware determines if a response is cacheable.

Options

Response caching options are shown in the following table.

OPTION	DESCRIPTION
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is `64 * 1024 * 1024` (64 MB).
SizeLimit	The size limit for the response cache middleware in bytes. The default value is `100 * 1024 * 1024` (100 MB).
UseCaseSensitivePaths	Determines if responses are cached on case-sensitive paths. The default value is `false`.

The following example configures the middleware to:

- Cache responses with a body size smaller than or equal to 1,024 bytes.
- Store the responses by case-sensitive paths. For example, `/page1` and `/Page1` are stored separately.

```
services.AddResponseCaching(options =>
{
    options.MaximumBodySize = 1024;
    options.UseCaseSensitivePaths = true;
});
```

VaryByQueryKeys

When using MVC / web API controllers or Razor Pages page models, the `[ResponseCache]` attribute specifies the parameters necessary for setting the appropriate headers for response caching. The only parameter of the `[ResponseCache]` attribute that strictly requires the middleware is VaryByQueryKeys, which doesn't correspond to an actual HTTP header. For more information, see Response caching in ASP.NET Core.

When not using the `[ResponseCache]` attribute, response caching can be varied with `VaryByQueryKeys`. Use the ResponseCachingFeature directly from the HttpContext.Features:

```
var responseCachingFeature = context.HttpContext.Features.Get<IResponseCachingFeature>();

if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

Using a single value equal to `*` in `VaryByQueryKeys` varies the cache by all request query parameters.

HTTP headers used by Response Caching Middleware

The following table provides information on HTTP headers that affect response caching.

HEADER	DETAILS
`Authorization`	The response isn't cached if the header exists.
`Cache-Control`	The middleware only considers caching responses marked with the `public` cache directive. Control caching with the following parameters: • max-age • max-stale† • min-fresh • must-revalidate • no-cache • no-store • only-if-cached • private • public • s-maxage • proxy-revalidate‡ †If no limit is specified to `max-stale`, the middleware takes no action. ‡ `proxy-revalidate` has the same effect as `must-revalidate`. For more information, see RFC 7231: Request Cache-Control Directives.
`Pragma`	A `Pragma: no-cache` header in the request produces the same effect as `Cache-Control: no-cache`. This header is overridden by the relevant directives in the `Cache-Control` header, if present. Considered for backward compatibility with HTTP/1.0.

HEADER	DETAILS
`Set-Cookie`	The response isn't cached if the header exists. Any middleware in the request processing pipeline that sets one or more cookies prevents the Response Caching Middleware from caching the response (for example, the cookie-based TempData provider).
`Vary`	The `Vary` header is used to vary the cached response by another header. For example, cache responses by encoding by including the `Vary: Accept-Encoding` header, which caches responses for requests with headers `Accept-Encoding: gzip` and `Accept-Encoding: text/plain` separately. A response with a header value of `*` is never stored.
`Expires`	A response deemed stale by this header isn't stored or retrieved unless overridden by other `Cache-Control` headers.
`If-None-Match`	The full response is served from cache if the value isn't `*` and the `ETag` of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.
`If-Modified-Since`	If the `If-None-Match` header isn't present, a full response is served from cache if the cached response date is newer than the value provided. Otherwise, a *304 - Not Modified* response is served.
`Date`	When serving from cache, the `Date` header is set by the middleware if it wasn't provided on the original response.
`Content-Length`	When serving from cache, the `Content-Length` header is set by the middleware if it wasn't provided on the original response.
`Age`	The `Age` header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

Caching respects request Cache-Control directives

The middleware respects the rules of the HTTP 1.1 Caching specification. The rules require a cache to honor a valid `Cache-Control` header sent by the client. Under the specification, a client can make requests with a `no-cache` header value and force the server to generate a new response for every request. Currently, there's no developer control over this caching behavior when using the middleware because the middleware adheres to the official caching specification.

For more control over caching behavior, explore other caching features of ASP.NET Core. See the following topics:

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Troubleshooting

If caching behavior isn't as expected, confirm that responses are cacheable and capable of being served from the cache. Examine the request's incoming headers and the response's outgoing headers. Enable logging to help with debugging.

When testing and troubleshooting caching behavior, a browser may set request headers that affect caching in undesirable ways. For example, a browser may set the `Cache-Control` header to `no-cache` or `max-age=0` when refreshing a page. The following tools can explicitly set request headers and are preferred for testing caching:

- Fiddler
- Postman

Conditions for caching

- The request must result in a server response with a 200 (OK) status code.
- The request method must be GET or HEAD.
- In `Startup.Configure`, Response Caching Middleware must be placed before middleware that require caching. For more information, see ASP.NET Core Middleware.
- The `Authorization` header must not be present.
- `Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
- The `Pragma: no-cache` header must not be present if the `Cache-Control` header isn't present, as the `Cache-Control` header overrides the `Pragma` header when present.
- The `Set-Cookie` header must not be present.
- `Vary` header parameters must be valid and not equal to `*`.
- The `Content-Length` header value (if set) must match the size of the response body.
- The IHttpSendFileFeature isn't used.
- The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
- Response buffering must be successful. The size of the response must be smaller than the configured or default SizeLimit. The body size of the response must be smaller than the configured or default MaximumBodySize.
- The response must be cacheable according to the RFC 7234 specifications. For example, the `no-store` directive must not exist in request or response header fields. See *Section 3: Storing Responses in Caches* of RFC 7234 for details.

> **NOTE**
>
> The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks sets the `Cache-Control` and `Pragma` headers to `no-cache` so that responses aren't cached. For information on how to disable antiforgery tokens for HTML form elements, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Additional resources

- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core

- Response caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

This article explains how to configure Response Caching Middleware in an ASP.NET Core app. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the `[ResponseCache]` attribute, see Response Caching.

View or download sample code (how to download)

Configuration

Use the Microsoft.AspNetCore.App metapackage or add a package reference to the Microsoft.AspNetCore.ResponseCaching package.

In `Startup.ConfigureServices`, add the Response Caching Middleware to the service collection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCaching();
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Configure the app to use the middleware with the UseResponseCaching extension method, which adds the middleware to the request processing pipeline in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();

    app.UseResponseCaching();

    app.Use(async (context, next) =>
    {
        context.Response.GetTypedHeaders().CacheControl =
            new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
            {
                Public = true,
                MaxAge = TimeSpan.FromSeconds(10)
            };
        context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
            new string[] { "Accept-Encoding" };

        await next();
    });

    app.UseMvc();
}
```

The sample app adds headers to control caching on subsequent requests:

- Cache-Control: Caches cacheable responses for up to 10 seconds.
- Vary: Configures the middleware to serve a cached response only if the Accept-Encoding header of subsequent requests matches that of the original request.

```
// using Microsoft.AspNetCore.Http;

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next();
});
```

The preceding headers are not written to the response and are overridden when a controller, action, or Razor Page:

- Has a [ResponseCache] attribute. This applies even if a property isn't set. For example, omitting the VaryByHeader property will cause the corresponding header to be removed from the response.

Response Caching Middleware only caches server responses that result in a 200 (OK) status code. Any other responses, including error pages, are ignored by the middleware.

> **WARNING**
>
> Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See Conditions for caching for details on how the middleware determines if a response is cacheable.

Options

Response caching options are shown in the following table.

OPTION	DESCRIPTION
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is `64 * 1024 * 1024` (64 MB).
SizeLimit	The size limit for the response cache middleware in bytes. The default value is `100 * 1024 * 1024` (100 MB).
UseCaseSensitivePaths	Determines if responses are cached on case-sensitive paths. The default value is `false`.

The following example configures the middleware to:

- Cache responses with a body size smaller than or equal to 1,024 bytes.
- Store the responses by case-sensitive paths. For example, `/page1` and `/Page1` are stored separately.

```
services.AddResponseCaching(options =>
{
    options.MaximumBodySize = 1024;
    options.UseCaseSensitivePaths = true;
});
```

VaryByQueryKeys

When using MVC / web API controllers or Razor Pages page models, the `[ResponseCache]` attribute specifies the parameters necessary for setting the appropriate headers for response caching. The only parameter of the `[ResponseCache]` attribute that strictly requires the middleware is VaryByQueryKeys, which doesn't correspond to an actual HTTP header. For more information, see Response caching in ASP.NET Core.

When not using the `[ResponseCache]` attribute, response caching can be varied with `VaryByQueryKeys`. Use the ResponseCachingFeature directly from the HttpContext.Features:

```
var responseCachingFeature = context.HttpContext.Features.Get<IResponseCachingFeature>();

if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

Using a single value equal to `*` in `VaryByQueryKeys` varies the cache by all request query parameters.

HTTP headers used by Response Caching Middleware

The following table provides information on HTTP headers that affect response caching.

HEADER	DETAILS
`Authorization`	The response isn't cached if the header exists.
`Cache-Control`	The middleware only considers caching responses marked with the `public` cache directive. Control caching with the following parameters: • max-age • max-stale† • min-fresh • must-revalidate • no-cache • no-store • only-if-cached • private • public • s-maxage • proxy-revalidate‡ †If no limit is specified to `max-stale`, the middleware takes no action. ‡ `proxy-revalidate` has the same effect as `must-revalidate`. For more information, see RFC 7231: Request Cache-Control Directives.

HEADER	DETAILS
`Pragma`	A `Pragma: no-cache` header in the request produces the same effect as `Cache-Control: no-cache`. This header is overridden by the relevant directives in the `Cache-Control` header, if present. Considered for backward compatibility with HTTP/1.0.
`Set-Cookie`	The response isn't cached if the header exists. Any middleware in the request processing pipeline that sets one or more cookies prevents the Response Caching Middleware from caching the response (for example, the cookie-based TempData provider).
`Vary`	The `Vary` header is used to vary the cached response by another header. For example, cache responses by encoding by including the `Vary: Accept-Encoding` header, which caches responses for requests with headers `Accept-Encoding: gzip` and `Accept-Encoding: text/plain` separately. A response with a header value of `*` is never stored.
`Expires`	A response deemed stale by this header isn't stored or retrieved unless overridden by other `Cache-Control` headers.
`If-None-Match`	The full response is served from cache if the value isn't `*` and the `ETag` of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.
`If-Modified-Since`	If the `If-None-Match` header isn't present, a full response is served from cache if the cached response date is newer than the value provided. Otherwise, a *304 - Not Modified* response is served.
`Date`	When serving from cache, the `Date` header is set by the middleware if it wasn't provided on the original response.
`Content-Length`	When serving from cache, the `Content-Length` header is set by the middleware if it wasn't provided on the original response.
`Age`	The `Age` header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

Caching respects request Cache-Control directives

The middleware respects the rules of the HTTP 1.1 Caching specification. The rules require a cache to honor a valid `Cache-Control` header sent by the client. Under the specification, a client can make requests with a `no-cache` header value and force the server to generate a new response for every request. Currently, there's no developer control over this caching behavior when using the middleware because the middleware adheres to the official caching specification.

For more control over caching behavior, explore other caching features of ASP.NET Core. See the following

topics:

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Troubleshooting

If caching behavior isn't as expected, confirm that responses are cacheable and capable of being served from the cache. Examine the request's incoming headers and the response's outgoing headers. Enable logging to help with debugging.

When testing and troubleshooting caching behavior, a browser may set request headers that affect caching in undesirable ways. For example, a browser may set the `Cache-Control` header to `no-cache` or `max-age=0` when refreshing a page. The following tools can explicitly set request headers and are preferred for testing caching:

- Fiddler
- Postman

Conditions for caching

- The request must result in a server response with a 200 (OK) status code.
- The request method must be GET or HEAD.
- In `Startup.Configure`, Response Caching Middleware must be placed before middleware that require caching. For more information, see ASP.NET Core Middleware.
- The `Authorization` header must not be present.
- `Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
- The `Pragma: no-cache` header must not be present if the `Cache-Control` header isn't present, as the `Cache-Control` header overrides the `Pragma` header when present.
- The `Set-Cookie` header must not be present.
- `Vary` header parameters must be valid and not equal to `*`.
- The `Content-Length` header value (if set) must match the size of the response body.
- The IHttpSendFileFeature isn't used.
- The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
- Response buffering must be successful. The size of the response must be smaller than the configured or default SizeLimit. The body size of the response must be smaller than the configured or default MaximumBodySize.
- The response must be cacheable according to the RFC 7234 specifications. For example, the `no-store` directive must not exist in request or response header fields. See *Section 3: Storing Responses in Caches* of RFC 7234 for details.

> **NOTE**
>
> The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks sets the `Cache-Control` and `Pragma` headers to `no-cache` so that responses aren't cached. For information on how to disable antiforgery tokens for HTML form elements, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Additional resources

- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Object reuse with ObjectPool in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Steve Gordon, Ryan Nowak, and Günther Foidl

Microsoft.Extensions.ObjectPool is part of the ASP.NET Core infrastructure that supports keeping a group of objects in memory for reuse rather than allowing the objects to be garbage collected.

You might want to use the object pool if the objects that are being managed are:

- Expensive to allocate/initialize.
- Represent some limited resource.
- Used predictably and frequently.

For example, the ASP.NET Core framework uses the object pool in some places to reuse StringBuilder instances. `StringBuilder` allocates and manages its own buffers to hold character data. ASP.NET Core regularly uses `StringBuilder` to implement features, and reusing them provides a performance benefit.

Object pooling doesn't always improve performance:

- Unless the initialization cost of an object is high, it's usually slower to get the object from the pool.
- Objects managed by the pool aren't de-allocated until the pool is de-allocated.

Use object pooling only after collecting performance data using realistic scenarios for your app or library.

WARNING: The `ObjectPool` doesn't implement `IDisposable`. We don't recommend using it with types that need disposal. `ObjectPool` in ASP.NET Core 3.0 and later supports `IDisposable`.

NOTE: The ObjectPool doesn't place a limit on the number of objects that it will allocate, it places a limit on the number of objects it will retain.

Concepts

ObjectPool<T> - the basic object pool abstraction. Used to get and return objects.

PooledObjectPolicy<T> - implement this to customize how an object is created and how it is *reset* when returned to the pool. This can be passed into an object pool that you construct directly.... OR

Create acts as a factory for creating object pools.

The ObjectPool can be used in an app in multiple ways:

- Instantiating a pool.
- Registering a pool in Dependency injection (DI) as an instance.
- Registering the `ObjectPoolProvider<>` in DI and using it as a factory.

How to use ObjectPool

Call Get to get an object and Return to return the object. There's no requirement that you return every object. If you don't return an object, it will be garbage collected.

When DefaultObjectPoolProvider is used and `T` implements `IDisposable`:

- Items that are ***not*** returned to the pool will be disposed.

- When the pool gets disposed by DI, all items in the pool are disposed.

NOTE: After the pool is disposed:

- Calling `Get` throws a `ObjectDisposedException`.
- `return` disposes the given item.

ObjectPool sample

The following code:

- Adds `ObjectPoolProvider` to the Dependency injection (DI) container.
- Adds and configures `ObjectPool<StringBuilder>` to the DI container.
- Adds the `BirthdayMiddleware`.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.TryAddSingleton<ObjectPoolProvider, DefaultObjectPoolProvider>();

        services.TryAddSingleton<ObjectPool<StringBuilder>>(serviceProvider =>
        {
            var provider = serviceProvider.GetRequiredService<ObjectPoolProvider>();
            var policy = new StringBuilderPooledObjectPolicy();
            return provider.Create(policy);
        });

        services.AddWebEncoders();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        // Test using /?firstname=Steve&lastName=Gordon&day=28&month=9
        app.UseMiddleware<BirthdayMiddleware>();
    }
}
```

The following code implements `BirthdayMiddleware`

```
public class BirthdayMiddleware
{
    private readonly RequestDelegate _next;

    public BirthdayMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context,
                                  ObjectPool<StringBuilder> builderPool)
    {
        if (context.Request.Query.TryGetValue("firstName", out var firstName) &&
            context.Request.Query.TryGetValue("lastName", out var lastName) &&
            context.Request.Query.TryGetValue("month", out var month) &&
            context.Request.Query.TryGetValue("day", out var day) &&
            int.TryParse(month, out var monthOfYear) &&
            int.TryParse(day, out var dayOfMonth))
        {
            var now = DateTime.UtcNow; // Ignoring timezones.

            // Request a StringBuilder from the pool.
            var stringBuilder = builderPool.Get();

            try
            {
                stringBuilder.Append("Hi ")
                    .Append(firstName).Append(" ").Append(lastName).Append(". ");

                var encoder = context.RequestServices.GetRequiredService<HtmlEncoder>();

                if (now.Day == dayOfMonth && now.Month == monthOfYear)
                {
                    stringBuilder.Append("Happy birthday!!!");

                    var html = encoder.Encode(stringBuilder.ToString());
                    await context.Response.WriteAsync(html);
                }
                else
                {
                    var thisYearsBirthday = new DateTime(now.Year, monthOfYear,
                                                         dayOfMonth);

                    int daysUntilBirthday = thisYearsBirthday > now
                        ? (thisYearsBirthday - now).Days
                        : (thisYearsBirthday.AddYears(1) - now).Days;

                    stringBuilder.Append("There are ")
                        .Append(daysUntilBirthday).Append(" days until your birthday!");

                    var html = encoder.Encode(stringBuilder.ToString());
                    await context.Response.WriteAsync(html);
                }
            }
            finally // Ensure this runs even if the main code throws.
            {
                // Return the StringBuilder to the pool.
                builderPool.Return(stringBuilder);
            }

            return;
        }

        await _next(context);
    }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

Response compression in ASP.NET Core

9/22/2020 • 27 minutes to read • Edit Online

Network bandwidth is a limited resource. Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses.

View or download sample code (how to download)

When to use Response Compression Middleware

Use server-based response compression technologies in IIS, Apache, or Nginx. The performance of the middleware probably won't match that of the server modules. HTTP.sys server server and Kestrel server don't currently offer built-in compression support.

Use Response Compression Middleware when you're:

- Unable to use the following server-based compression technologies:
  - IIS Dynamic Compression module
  - Apache mod_deflate module
  - Nginx Compression and Decompression
- Hosting directly on:
  - HTTP.sys server (formerly called WebListener)
  - Kestrel server

Response compression

Usually, any response not natively compressed can benefit from response compression. Responses not natively compressed typically include: CSS, JavaScript, HTML, XML, and JSON. You shouldn't compress natively compressed assets, such as PNG files. If you attempt to further compress a natively compressed response, any small additional reduction in size and transmission time will likely be overshadowed by the time it took to process the compression. Don't compress files smaller than about 150-1000 bytes (depending on the file's content and the efficiency of compression). The overhead of compressing small files may produce a compressed file larger than the uncompressed file.

When a client can process compressed content, the client must inform the server of its capabilities by sending the `Accept-Encoding` header with the request. When a server sends compressed content, it must include information in the `Content-Encoding` header on how the compressed response is encoded. Content encoding designations supported by the middleware are shown in the following table.

`ACCEPT-ENCODING` HEADER VALUES	MIDDLEWARE SUPPORTED	DESCRIPTION
`br`	Yes (default)	Brotli compressed data format
`deflate`	No	DEFLATE compressed data format
`exi`	No	W3C Efficient XML Interchange
`gzip`	Yes	Gzip file format

`Accept-Encoding` HEADER VALUES	MIDDLEWARE SUPPORTED	DESCRIPTION
`identity`	Yes	"No encoding" identifier: The response must not be encoded.
`pack200-gzip`	No	Network Transfer Format for Java Archives
`*`	Yes	Any available content encoding not explicitly requested

For more information, see the IANA Official Content Coding List.

The middleware allows you to add additional compression providers for custom `Accept-Encoding` header values. For more information, see Custom Providers below.

The middleware is capable of reacting to quality value (qvalue, `q`) weighting when sent by the client to prioritize compression schemes. For more information, see RFC 7231: Accept-Encoding.

Compression algorithms are subject to a tradeoff between compression speed and the effectiveness of the compression. *Effectiveness* in this context refers to the size of the output after compression. The smallest size is achieved by the most *optimal* compression.

The headers involved in requesting, sending, caching, and receiving compressed content are described in the table below.

HEADER	ROLE
`Accept-Encoding`	Sent from the client to the server to indicate the content encoding schemes acceptable to the client.
`Content-Encoding`	Sent from the server to the client to indicate the encoding of the content in the payload.
`Content-Length`	When compression occurs, the `Content-Length` header is removed, since the body content changes when the response is compressed.
`Content-MD5`	When compression occurs, the `Content-MD5` header is removed, since the body content has changed and the hash is no longer valid.
`Content-Type`	Specifies the MIME type of the content. Every response should specify its `Content-Type`. The middleware checks this value to determine if the response should be compressed. The middleware specifies a set of default MIME types that it can encode, but you can replace or add MIME types.
`Vary`	When sent by the server with a value of `Accept-Encoding` to clients and proxies, the `Vary` header indicates to the client or proxy that it should cache (vary) responses based on the value of the `Accept-Encoding` header of the request. The result of returning content with the `Vary: Accept-Encoding` header is that both compressed and uncompressed responses are cached separately.

Explore the features of the Response Compression Middleware with the sample app. The sample illustrates:

- The compression of app responses using Gzip and custom compression providers.
- How to add a MIME type to the default list of MIME types for compression.

Package

Response Compression Middleware is provided by the Microsoft.AspNetCore.ResponseCompression package, which is implicitly included in ASP.NET Core apps.

Configuration

The following code shows how to enable the Response Compression Middleware for default MIME types and compression providers (Brotli and Gzip):

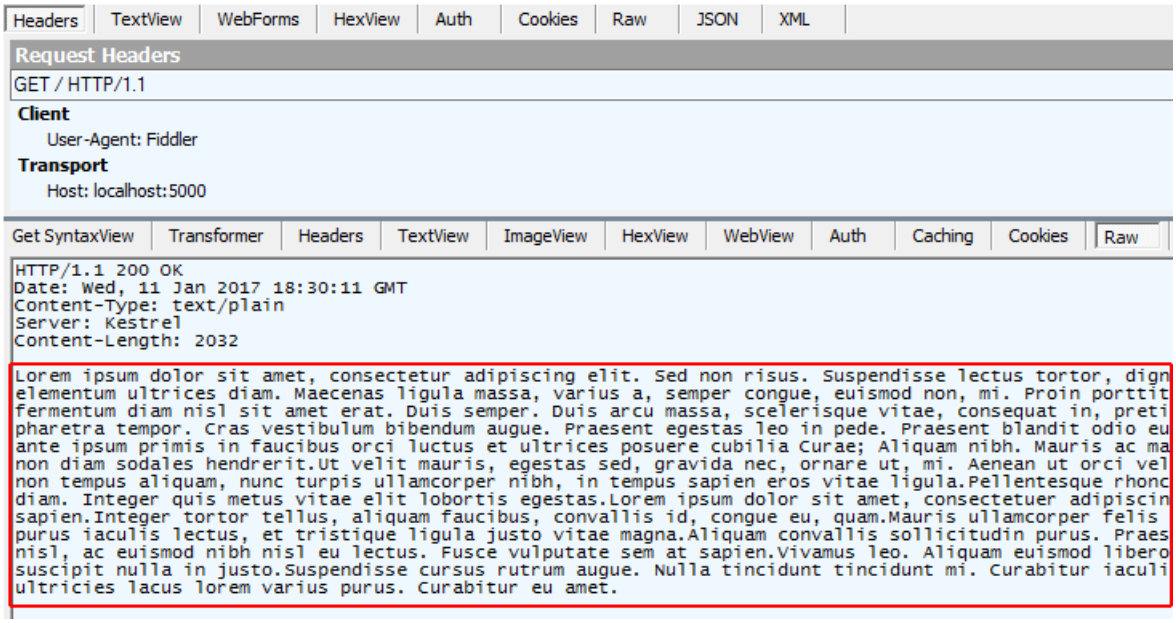
```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddResponseCompression();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseResponseCompression();
    }
}
```

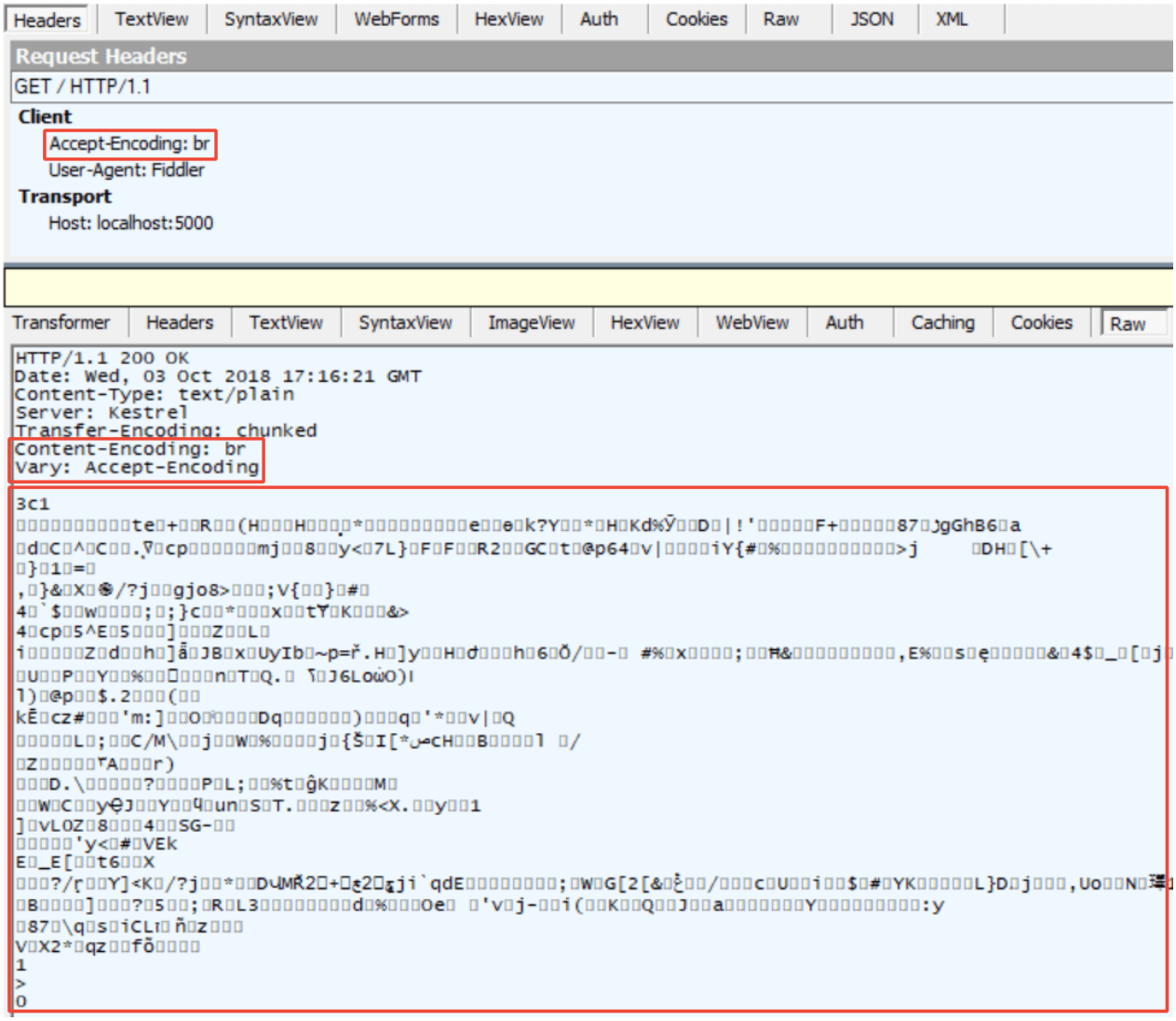
Notes:

- `app.UseResponseCompression` must be called before any middleware that compresses responses. For more information, see ASP.NET Core Middleware.
- Use a tool such as Fiddler, Firebug, or Postman to set the `Accept-Encoding` request header and study the response headers, size, and body.

Submit a request to the sample app without the `Accept-Encoding` header and observe that the response is uncompressed. The `Content-Encoding` and `Vary` headers aren't present on the response.



Submit a request to the sample app with the `Accept-Encoding: br` header (Brotli compression) and observe that the response is compressed. The `Content-Encoding` and `Vary` headers are present on the response.



Providers

Brotli Compression Provider

Use the BrotliCompressionProvider to compress responses with the Brotli compressed data format.

If no compression providers are explicitly added to the CompressionProviderCollection:

- The Brotli Compression Provider is added by default to the array of compression providers along with the Gzip compression provider.
- Compression defaults to Brotli compression when the Brotli compressed data format is supported by the client. If Brotli isn't supported by the client, compression defaults to Gzip when the client supports Gzip compression.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();
}
```

The Brotli Compression Provider must be added when any compression providers are explicitly added:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Set the compression level with BrotliCompressionProviderOptions. The Brotli Compression Provider defaults to the fastest compression level (CompressionLevel.Fastest), which might not produce the most efficient compression. If the most efficient compression is desired, configure the middleware for optimal compression.

COMPRESSION LEVEL	DESCRIPTION
CompressionLevel.Fastest	Compression should complete as quickly as possible, even if the resulting output isn't optimally compressed.
CompressionLevel.NoCompression	No compression should be performed.
CompressionLevel.Optimal	Responses should be optimally compressed, even if the compression takes more time to complete.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();

    services.Configure<BrotliCompressionProviderOptions>(options =>
    {
        options.Level = CompressionLevel.Fastest;
    });
}
```

Gzip Compression Provider

Use the GzipCompressionProvider to compress responses with the Gzip file format.

If no compression providers are explicitly added to the CompressionProviderCollection:

- The Gzip Compression Provider is added by default to the array of compression providers along with the Brotli Compression Provider.
- Compression defaults to Brotli compression when the Brotli compressed data format is supported by the client. If Brotli isn't supported by the client, compression defaults to Gzip when the client supports Gzip compression.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();
}
```

The Gzip Compression Provider must be added when any compression providers are explicitly added:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Set the compression level with GzipCompressionProviderOptions. The Gzip Compression Provider defaults to the fastest compression level (CompressionLevel.Fastest), which might not produce the most efficient compression. If the most efficient compression is desired, configure the middleware for optimal compression.

COMPRESSION LEVEL	DESCRIPTION
CompressionLevel.Fastest	Compression should complete as quickly as possible, even if the resulting output isn't optimally compressed.
CompressionLevel.NoCompression	No compression should be performed.
CompressionLevel.Optimal	Responses should be optimally compressed, even if the compression takes more time to complete.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();

    services.Configure<GzipCompressionProviderOptions>(options =>
    {
        options.Level = CompressionLevel.Fastest;
    });
}
```

Custom providers

Create custom compression implementations with ICompressionProvider. The EncodingName represents the content encoding that this `ICompressionProvider` produces. The middleware uses this information to choose the provider based on the list specified in the `Accept-Encoding` header of the request.

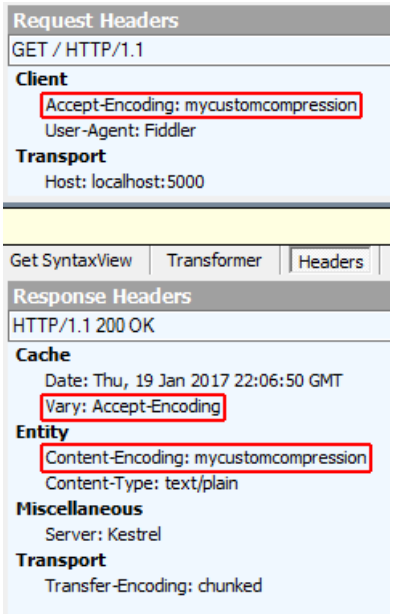
Using the sample app, the client submits a request with the `Accept-Encoding: mycustomcompression` header. The middleware uses the custom compression implementation and returns the response with a `Content-Encoding: mycustomcompression` header. The client must be able to decompress the custom encoding in order for a custom compression implementation to work.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}
```

Submit a request to the sample app with the `Accept-Encoding: mycustomcompression` header and observe the response headers. The `Vary` and `Content-Encoding` headers are present on the response. The response body (not shown) isn't compressed by the sample. There isn't a compression implementation in the `CustomCompressionProvider` class of the sample. However, the sample shows where you would implement such a compression algorithm.



MIME types

The middleware specifies a default set of MIME types for compression:

- `application/javascript`
- `application/json`
- `application/xml`
- `text/css`

- `text/html`
- `text/json`
- `text/plain`
- `text/xml`

Replace or append MIME types with the Response Compression Middleware options. Note that wildcard MIME types, such as `text/*` aren't supported. The sample app adds a MIME type for `image/svg+xml` and compresses and serves the ASP.NET Core banner image (*banner.svg*).

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Compression with secure protocol

Compressed responses over secure connections can be controlled with the `EnableForHttps` option, which is disabled by default. Using compression with dynamically generated pages can lead to security problems such as the CRIME and BREACH attacks.

Adding the Vary header

When compressing responses based on the `Accept-Encoding` header, there are potentially multiple compressed versions of the response and an uncompressed version. In order to instruct client and proxy caches that multiple versions exist and should be stored, the `Vary` header is added with an `Accept-Encoding` value. In ASP.NET Core 2.0 or later, the middleware adds the `Vary` header automatically when the response is compressed.

Middleware issue when behind an Nginx reverse proxy

When a request is proxied by Nginx, the `Accept-Encoding` header is removed. Removal of the `Accept-Encoding` header prevents the middleware from compressing the response. For more information, see NGINX: Compression and Decompression. This issue is tracked by Figure out pass-through compression for Nginx (aspnet/BasicMiddleware #123).

Working with IIS dynamic compression

If you have an active IIS Dynamic Compression Module configured at the server level that you would like to disable for an app, disable the module with an addition to the *web.config* file. For more information, see Disabling IIS modules.

Troubleshooting

Use a tool like Fiddler, Firebug, or Postman, which allow you to set the `Accept-Encoding` request header and study the response headers, size, and body. By default, Response Compression Middleware compresses responses that meet the following conditions:

- The `Accept-Encoding` header is present with a value of `br`, `gzip`, `*`, or custom encoding that matches a

custom compression provider that you've established. The value must not be `identity` or have a quality value (qvalue, `q`) setting of 0 (zero).

- The MIME type (`Content-Type`) must be set and must match a MIME type configured on the ResponseCompressionOptions.
- The request must not include the `Content-Range` header.
- The request must use insecure protocol (http), unless secure protocol (https) is configured in the Response Compression Middleware options. *Note the danger described above when enabling secure content compression.*

Additional resources

- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Mozilla Developer Network: Accept-Encoding
- RFC 7231 Section 3.1.2.1: Content Codings
- RFC 7230 Section 4.2.3: Gzip Coding
- GZIP file format specification version 4.3

Network bandwidth is a limited resource. Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses.

View or download sample code (how to download)

When to use Response Compression Middleware

Use server-based response compression technologies in IIS, Apache, or Nginx. The performance of the middleware probably won't match that of the server modules. HTTP.sys server server and Kestrel server don't currently offer built-in compression support.

Use Response Compression Middleware when you're:

- Unable to use the following server-based compression technologies:
  - IIS Dynamic Compression module
  - Apache mod_deflate module
  - Nginx Compression and Decompression
- Hosting directly on:
  - HTTP.sys server (formerly called WebListener)
  - Kestrel server

Response compression

Usually, any response not natively compressed can benefit from response compression. Responses not natively compressed typically include: CSS, JavaScript, HTML, XML, and JSON. You shouldn't compress natively compressed assets, such as PNG files. If you attempt to further compress a natively compressed response, any small additional reduction in size and transmission time will likely be overshadowed by the time it took to process the compression. Don't compress files smaller than about 150-1000 bytes (depending on the file's content and the efficiency of compression). The overhead of compressing small files may produce a compressed file larger than the uncompressed file.

When a client can process compressed content, the client must inform the server of its capabilities by sending the `Accept-Encoding` header with the request. When a server sends compressed content, it must include information in the `Content-Encoding` header on how the compressed response is encoded. Content encoding designations supported by the middleware are shown in the following table.

`Accept-Encoding` header values	Middleware supported	Description
`br`	Yes (default)	Brotli compressed data format
`deflate`	No	DEFLATE compressed data format
`exi`	No	W3C Efficient XML Interchange
`gzip`	Yes	Gzip file format
`identity`	Yes	"No encoding" identifier: The response must not be encoded.
`pack200-gzip`	No	Network Transfer Format for Java Archives
`*`	Yes	Any available content encoding not explicitly requested

For more information, see the IANA Official Content Coding List.

The middleware allows you to add additional compression providers for custom `Accept-Encoding` header values. For more information, see Custom Providers below.

The middleware is capable of reacting to quality value (qvalue, `q`) weighting when sent by the client to prioritize compression schemes. For more information, see RFC 7231: Accept-Encoding.

Compression algorithms are subject to a tradeoff between compression speed and the effectiveness of the compression. *Effectiveness* in this context refers to the size of the output after compression. The smallest size is achieved by the most *optimal* compression.

The headers involved in requesting, sending, caching, and receiving compressed content are described in the table below.

Header	Role
`Accept-Encoding`	Sent from the client to the server to indicate the content encoding schemes acceptable to the client.
`Content-Encoding`	Sent from the server to the client to indicate the encoding of the content in the payload.
`Content-Length`	When compression occurs, the `Content-Length` header is removed, since the body content changes when the response is compressed.
`Content-MD5`	When compression occurs, the `Content-MD5` header is removed, since the body content has changed and the hash is no longer valid.
`Content-Type`	Specifies the MIME type of the content. Every response should specify its `Content-Type`. The middleware checks this value to determine if the response should be compressed. The middleware specifies a set of default MIME types that it can encode, but you can replace or add MIME types.

HEADER	ROLE
`Vary`	When sent by the server with a value of `Accept-Encoding` to clients and proxies, the `Vary` header indicates to the client or proxy that it should cache (vary) responses based on the value of the `Accept-Encoding` header of the request. The result of returning content with the `Vary: Accept-Encoding` header is that both compressed and uncompressed responses are cached separately.

Explore the features of the Response Compression Middleware with the sample app. The sample illustrates:

- The compression of app responses using Gzip and custom compression providers.
- How to add a MIME type to the default list of MIME types for compression.

Package

To include the middleware in a project, add a reference to the Microsoft.AspNetCore.App metapackage, which includes the Microsoft.AspNetCore.ResponseCompression package.

Configuration

The following code shows how to enable the Response Compression Middleware for default MIME types and compression providers (Brotli and Gzip):

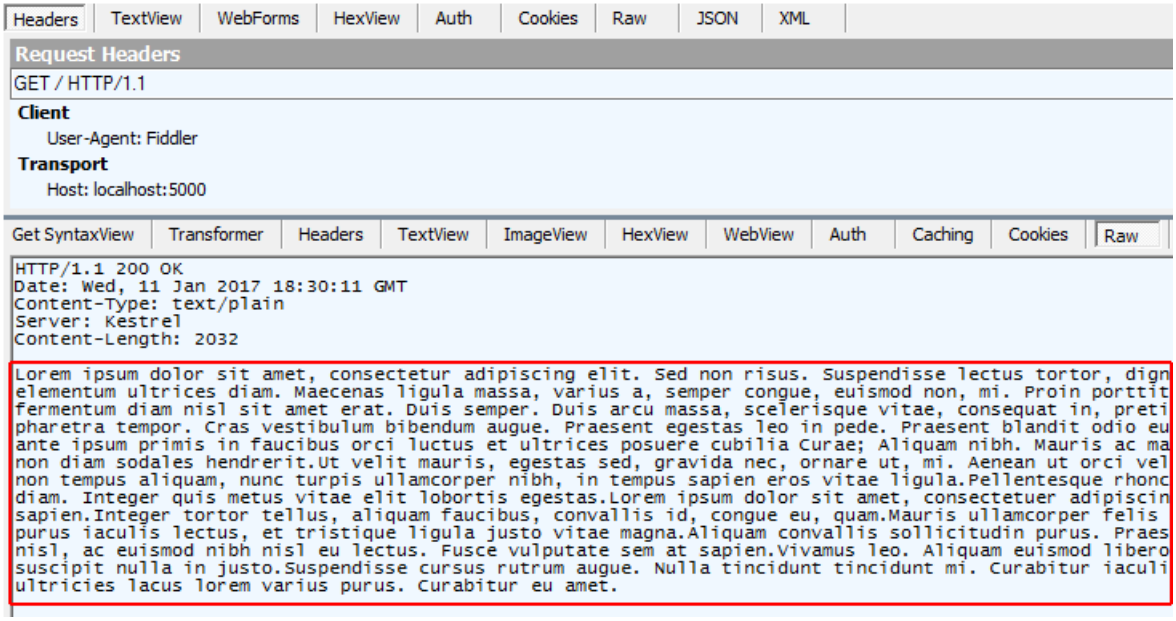
```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddResponseCompression();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseResponseCompression();
    }
}
```

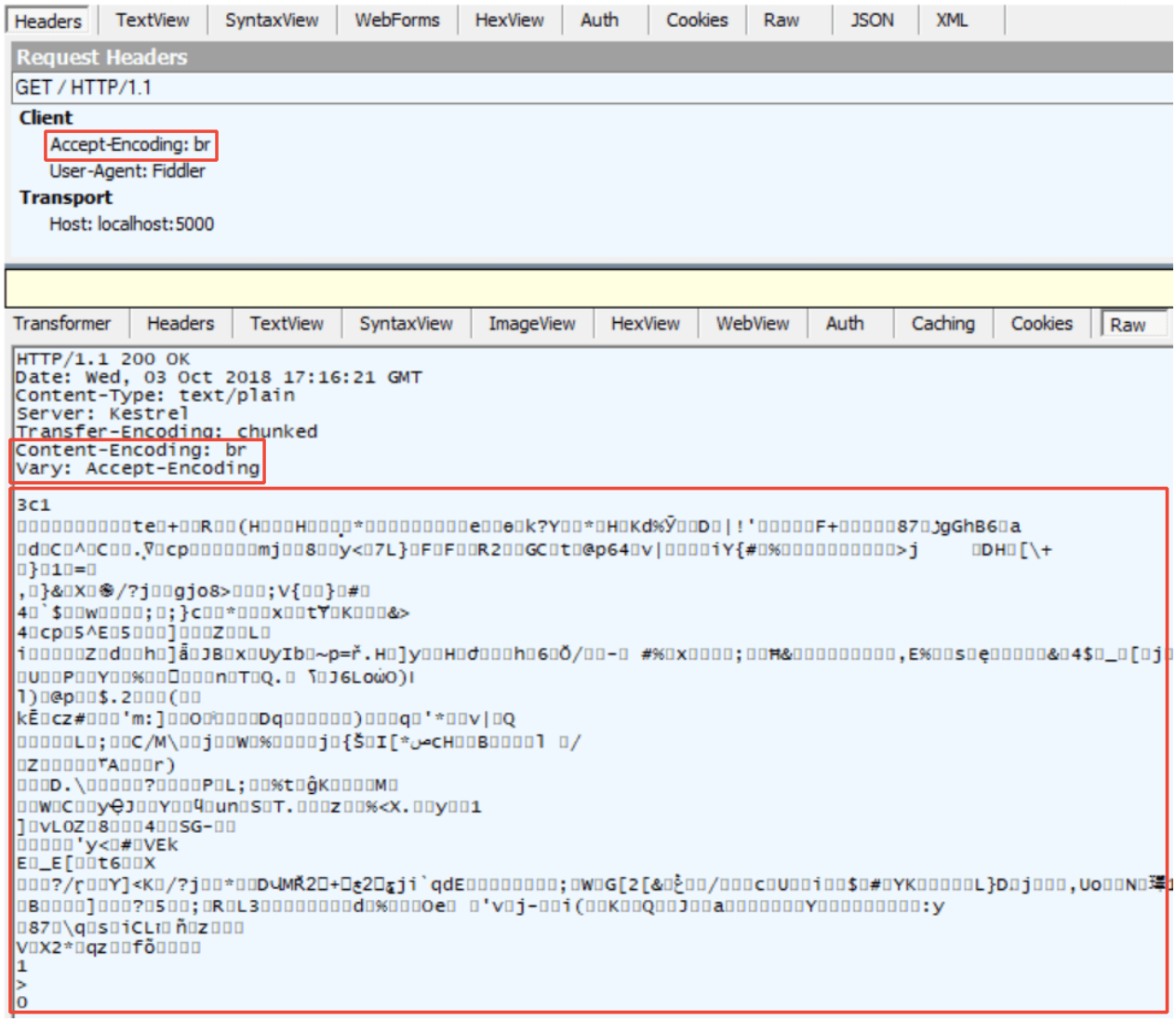
Notes:

- `app.UseResponseCompression` must be called before any middleware that compresses responses. For more information, see ASP.NET Core Middleware.
- Use a tool such as Fiddler, Firebug, or Postman to set the `Accept-Encoding` request header and study the response headers, size, and body.

Submit a request to the sample app without the `Accept-Encoding` header and observe that the response is uncompressed. The `Content-Encoding` and `Vary` headers aren't present on the response.



Submit a request to the sample app with the `Accept-Encoding: br` header (Brotli compression) and observe that the response is compressed. The `Content-Encoding` and `Vary` headers are present on the response.



Providers

Brotli Compression Provider

Use the BrotliCompressionProvider to compress responses with the Brotli compressed data format.

If no compression providers are explicitly added to the CompressionProviderCollection:

- The Brotli Compression Provider is added by default to the array of compression providers along with the Gzip compression provider.
- Compression defaults to Brotli compression when the Brotli compressed data format is supported by the client. If Brotli isn't supported by the client, compression defaults to Gzip when the client supports Gzip compression.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();
}
```

The Brotli Compression Provider must be added when any compression providers are explicitly added:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Set the compression level with BrotliCompressionProviderOptions. The Brotli Compression Provider defaults to the fastest compression level (CompressionLevel.Fastest), which might not produce the most efficient compression. If the most efficient compression is desired, configure the middleware for optimal compression.

COMPRESSION LEVEL	DESCRIPTION
CompressionLevel.Fastest	Compression should complete as quickly as possible, even if the resulting output isn't optimally compressed.
CompressionLevel.NoCompression	No compression should be performed.
CompressionLevel.Optimal	Responses should be optimally compressed, even if the compression takes more time to complete.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();

    services.Configure<BrotliCompressionProviderOptions>(options =>
    {
        options.Level = CompressionLevel.Fastest;
    });
}
```

Gzip Compression Provider

Use the GzipCompressionProvider to compress responses with the Gzip file format.

If no compression providers are explicitly added to the CompressionProviderCollection:

- The Gzip Compression Provider is added by default to the array of compression providers along with the Brotli Compression Provider.
- Compression defaults to Brotli compression when the Brotli compressed data format is supported by the client. If Brotli isn't supported by the client, compression defaults to Gzip when the client supports Gzip compression.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();
}
```

The Gzip Compression Provider must be added when any compression providers are explicitly added:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Set the compression level with GzipCompressionProviderOptions. The Gzip Compression Provider defaults to the fastest compression level (CompressionLevel.Fastest), which might not produce the most efficient compression. If the most efficient compression is desired, configure the middleware for optimal compression.

COMPRESSION LEVEL	DESCRIPTION
CompressionLevel.Fastest	Compression should complete as quickly as possible, even if the resulting output isn't optimally compressed.
CompressionLevel.NoCompression	No compression should be performed.
CompressionLevel.Optimal	Responses should be optimally compressed, even if the compression takes more time to complete.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();

    services.Configure<GzipCompressionProviderOptions>(options =>
    {
        options.Level = CompressionLevel.Fastest;
    });
}
```

Custom providers

Create custom compression implementations with ICompressionProvider. The EncodingName represents the content encoding that this `ICompressionProvider` produces. The middleware uses this information to choose the provider based on the list specified in the `Accept-Encoding` header of the request.

Using the sample app, the client submits a request with the `Accept-Encoding: mycustomcompression` header. The

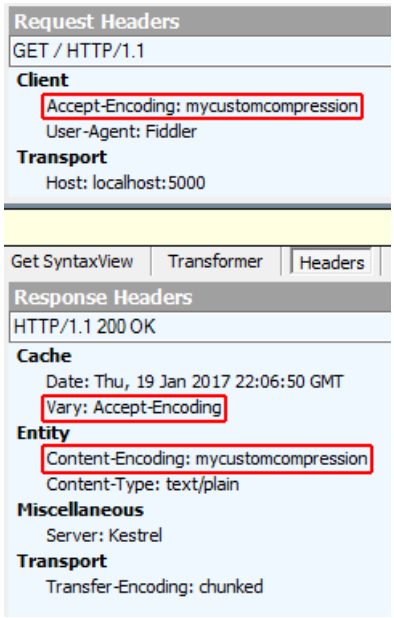
middleware uses the custom compression implementation and returns the response with a `Content-Encoding: mycustomcompression` header. The client must be able to decompress the custom encoding in order for a custom compression implementation to work.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}
```

Submit a request to the sample app with the `Accept-Encoding: mycustomcompression` header and observe the response headers. The `Vary` and `Content-Encoding` headers are present on the response. The response body (not shown) isn't compressed by the sample. There isn't a compression implementation in the `CustomCompressionProvider` class of the sample. However, the sample shows where you would implement such a compression algorithm.



MIME types

The middleware specifies a default set of MIME types for compression:

- `application/javascript`

- `application/json`
- `application/xml`
- `text/css`
- `text/html`
- `text/json`
- `text/plain`
- `text/xml`

Replace or append MIME types with the Response Compression Middleware options. Note that wildcard MIME types, such as `text/*` aren't supported. The sample app adds a MIME type for `image/svg+xml` and compresses and serves the ASP.NET Core banner image (*banner.svg*).

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Compression with secure protocol

Compressed responses over secure connections can be controlled with the `EnableForHttps` option, which is disabled by default. Using compression with dynamically generated pages can lead to security problems such as the CRIME and BREACH attacks.

Adding the Vary header

When compressing responses based on the `Accept-Encoding` header, there are potentially multiple compressed versions of the response and an uncompressed version. In order to instruct client and proxy caches that multiple versions exist and should be stored, the `Vary` header is added with an `Accept-Encoding` value. In ASP.NET Core 2.0 or later, the middleware adds the `Vary` header automatically when the response is compressed.

Middleware issue when behind an Nginx reverse proxy

When a request is proxied by Nginx, the `Accept-Encoding` header is removed. Removal of the `Accept-Encoding` header prevents the middleware from compressing the response. For more information, see NGINX: Compression and Decompression. This issue is tracked by Figure out pass-through compression for Nginx (aspnet/BasicMiddleware #123).

Working with IIS dynamic compression

If you have an active IIS Dynamic Compression Module configured at the server level that you would like to disable for an app, disable the module with an addition to the *web.config* file. For more information, see Disabling IIS modules.

Troubleshooting

Use a tool like Fiddler, Firebug, or Postman, which allow you to set the `Accept-Encoding` request header and study

the response headers, size, and body. By default, Response Compression Middleware compresses responses that meet the following conditions:

- The `Accept-Encoding` header is present with a value of `br`, `gzip`, `*`, or custom encoding that matches a custom compression provider that you've established. The value must not be `identity` or have a quality value (qvalue, `q`) setting of 0 (zero).
- The MIME type (`Content-Type`) must be set and must match a MIME type configured on the ResponseCompressionOptions.
- The request must not include the `Content-Range` header.
- The request must use insecure protocol (http), unless secure protocol (https) is configured in the Response Compression Middleware options. *Note the danger described above when enabling secure content compression.*

Additional resources

- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Mozilla Developer Network: Accept-Encoding
- RFC 7231 Section 3.1.2.1: Content Codings
- RFC 7230 Section 4.2.3: Gzip Coding
- GZIP file format specification version 4.3

Network bandwidth is a limited resource. Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses.

View or download sample code (how to download)

When to use Response Compression Middleware

Use server-based response compression technologies in IIS, Apache, or Nginx. The performance of the middleware probably won't match that of the server modules. HTTP.sys server server and Kestrel server don't currently offer built-in compression support.

Use Response Compression Middleware when you're:

- Unable to use the following server-based compression technologies:
  - IIS Dynamic Compression module
  - Apache mod_deflate module
  - Nginx Compression and Decompression
- Hosting directly on:
  - HTTP.sys server (formerly called WebListener)
  - Kestrel server

Response compression

Usually, any response not natively compressed can benefit from response compression. Responses not natively compressed typically include: CSS, JavaScript, HTML, XML, and JSON. You shouldn't compress natively compressed assets, such as PNG files. If you attempt to further compress a natively compressed response, any small additional reduction in size and transmission time will likely be overshadowed by the time it took to process the compression. Don't compress files smaller than about 150-1000 bytes (depending on the file's content and the efficiency of compression). The overhead of compressing small files may produce a compressed file larger than the uncompressed file.

When a client can process compressed content, the client must inform the server of its capabilities by sending the `Accept-Encoding` header with the request. When a server sends compressed content, it must include information in the `Content-Encoding` header on how the compressed response is encoded. Content encoding designations supported by the middleware are shown in the following table.

`ACCEPT-ENCODING` HEADER VALUES	MIDDLEWARE SUPPORTED	DESCRIPTION
`br`	No	Brotli compressed data format
`deflate`	No	DEFLATE compressed data format
`exi`	No	W3C Efficient XML Interchange
`gzip`	Yes (default)	Gzip file format
`identity`	Yes	"No encoding" identifier: The response must not be encoded.
`pack200-gzip`	No	Network Transfer Format for Java Archives
`*`	Yes	Any available content encoding not explicitly requested

For more information, see the IANA Official Content Coding List.

The middleware allows you to add additional compression providers for custom `Accept-Encoding` header values. For more information, see Custom Providers below.

The middleware is capable of reacting to quality value (qvalue, `q`) weighting when sent by the client to prioritize compression schemes. For more information, see RFC 7231: Accept-Encoding.

Compression algorithms are subject to a tradeoff between compression speed and the effectiveness of the compression. *Effectiveness* in this context refers to the size of the output after compression. The smallest size is achieved by the most *optimal* compression.

The headers involved in requesting, sending, caching, and receiving compressed content are described in the table below.

HEADER	ROLE
`Accept-Encoding`	Sent from the client to the server to indicate the content encoding schemes acceptable to the client.
`Content-Encoding`	Sent from the server to the client to indicate the encoding of the content in the payload.
`Content-Length`	When compression occurs, the `Content-Length` header is removed, since the body content changes when the response is compressed.
`Content-MD5`	When compression occurs, the `Content-MD5` header is removed, since the body content has changed and the hash is no longer valid.

HEADER	ROLE
`Content-Type`	Specifies the MIME type of the content. Every response should specify its `Content-Type`. The middleware checks this value to determine if the response should be compressed. The middleware specifies a set of default MIME types that it can encode, but you can replace or add MIME types.
`Vary`	When sent by the server with a value of `Accept-Encoding` to clients and proxies, the `Vary` header indicates to the client or proxy that it should cache (vary) responses based on the value of the `Accept-Encoding` header of the request. The result of returning content with the `Vary: Accept-Encoding` header is that both compressed and uncompressed responses are cached separately.

Explore the features of the Response Compression Middleware with the sample app. The sample illustrates:

- The compression of app responses using Gzip and custom compression providers.
- How to add a MIME type to the default list of MIME types for compression.

Package

To include the middleware in a project, add a reference to the Microsoft.AspNetCore.App metapackage, which includes the Microsoft.AspNetCore.ResponseCompression package.

Configuration

The following code shows how to enable the Response Compression Middleware for default MIME types and the Gzip Compression Provider:

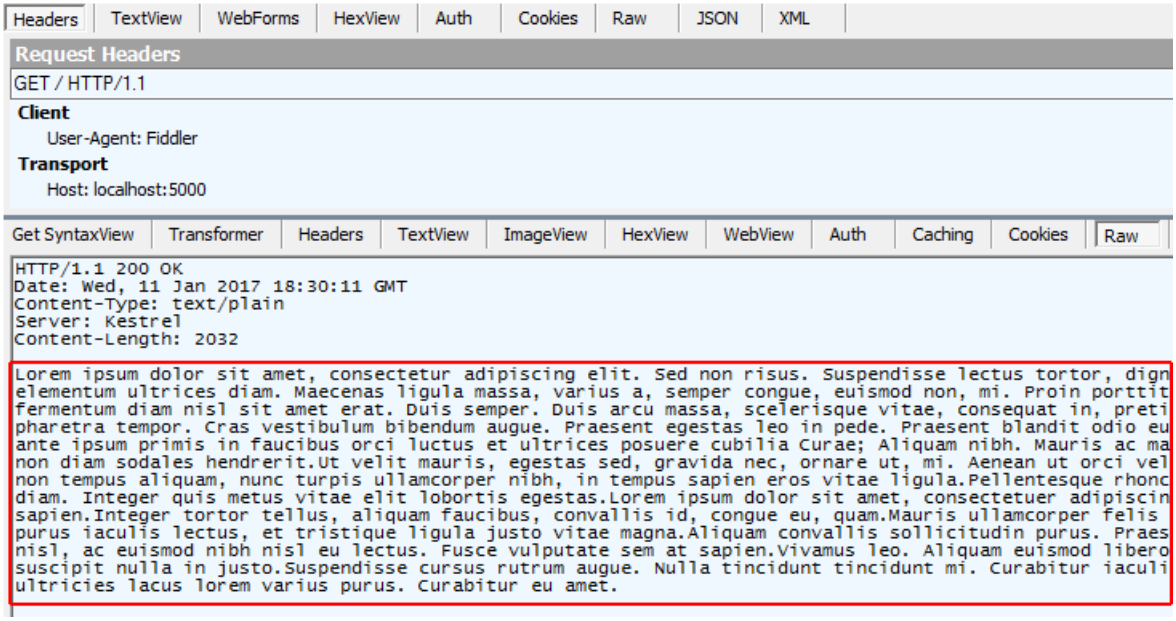
```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddResponseCompression();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseResponseCompression();
    }
}
```

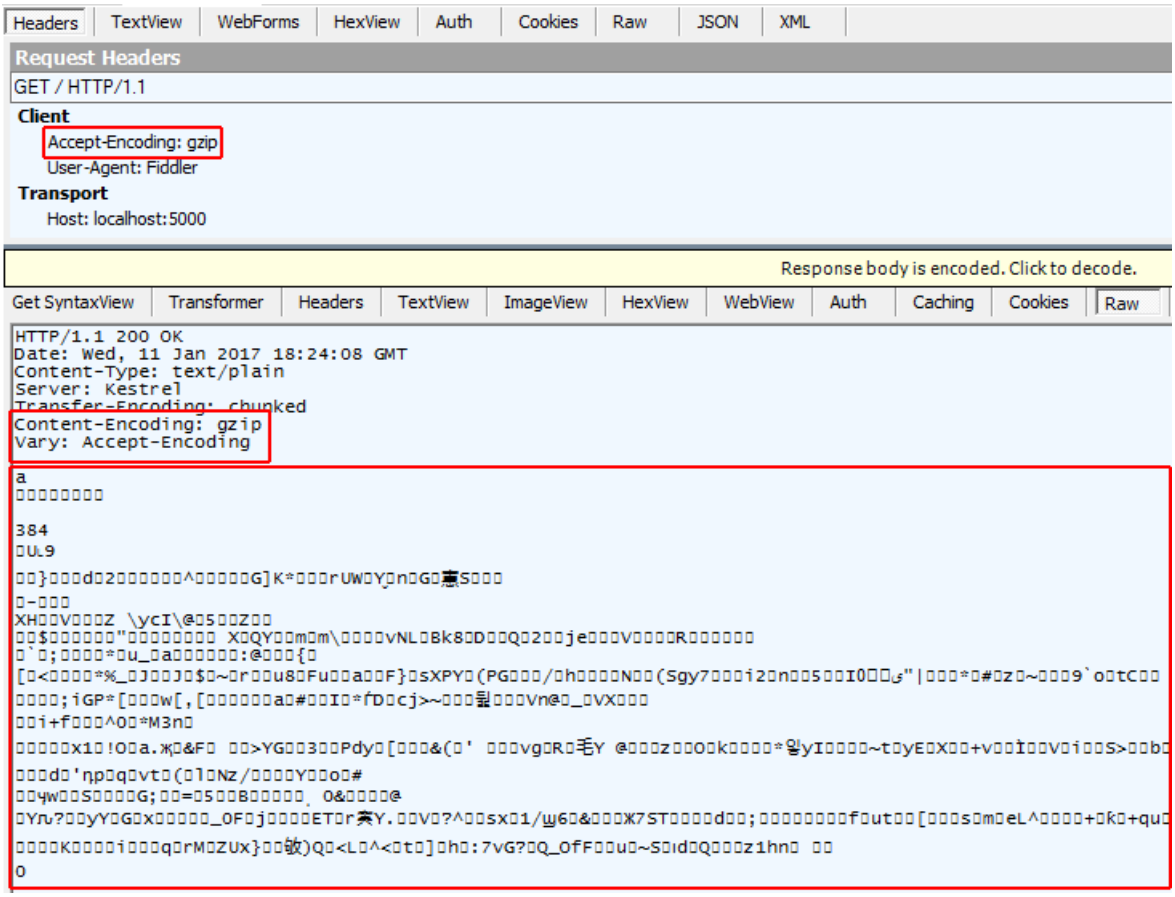
Notes:

- `app.UseResponseCompression` must be called before any middleware that compresses responses. For more information, see ASP.NET Core Middleware.
- Use a tool such as Fiddler, Firebug, or Postman to set the `Accept-Encoding` request header and study the response headers, size, and body.

Submit a request to the sample app without the `Accept-Encoding` header and observe that the response is uncompressed. The `Content-Encoding` and `Vary` headers aren't present on the response.



Submit a request to the sample app with the `Accept-Encoding: gzip` header and observe that the response is compressed. The `Content-Encoding` and `Vary` headers are present on the response.



Providers

Gzip Compression Provider

Use the GzipCompressionProvider to compress responses with the Gzip file format.

If no compression providers are explicitly added to the CompressionProviderCollection:

- The Gzip Compression Provider is added by default to the array of compression providers.
- Compression defaults to Gzip when the client supports Gzip compression.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();
}
```

The Gzip Compression Provider must be added when any compression providers are explicitly added:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Set the compression level with GzipCompressionProviderOptions. The Gzip Compression Provider defaults to the fastest compression level (CompressionLevel.Fastest), which might not produce the most efficient compression. If the most efficient compression is desired, configure the middleware for optimal compression.

COMPRESSION LEVEL	DESCRIPTION
CompressionLevel.Fastest	Compression should complete as quickly as possible, even if the resulting output isn't optimally compressed.
CompressionLevel.NoCompression	No compression should be performed.
CompressionLevel.Optimal	Responses should be optimally compressed, even if the compression takes more time to complete.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression();

    services.Configure<GzipCompressionProviderOptions>(options =>
    {
        options.Level = CompressionLevel.Fastest;
    });
}
```

Custom providers

Create custom compression implementations with ICompressionProvider. The EncodingName represents the content encoding that this `ICompressionProvider` produces. The middleware uses this information to choose the provider based on the list specified in the `Accept-Encoding` header of the request.

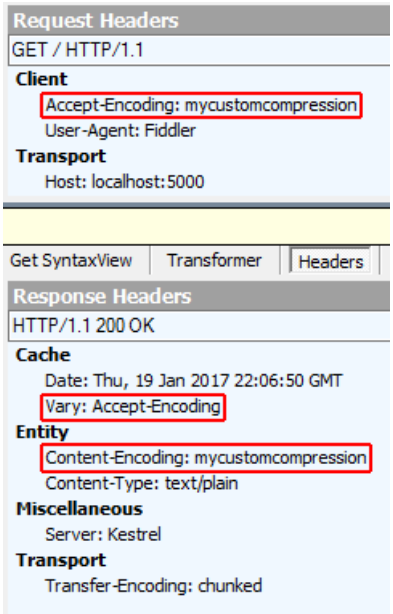
Using the sample app, the client submits a request with the `Accept-Encoding: mycustomcompression` header. The middleware uses the custom compression implementation and returns the response with a `Content-Encoding: mycustomcompression` header. The client must be able to decompress the custom encoding in order for a custom compression implementation to work.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

```
public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;

    public Stream CreateStream(Stream outputStream)
    {
        // Create a custom compression stream wrapper here
        return outputStream;
    }
}
```

Submit a request to the sample app with the `Accept-Encoding: mycustomcompression` header and observe the response headers. The `Vary` and `Content-Encoding` headers are present on the response. The response body (not shown) isn't compressed by the sample. There isn't a compression implementation in the `CustomCompressionProvider` class of the sample. However, the sample shows where you would implement such a compression algorithm.



MIME types

The middleware specifies a default set of MIME types for compression:

- `application/javascript`
- `application/json`
- `application/xml`
- `text/css`

- `text/html`
- `text/json`
- `text/plain`
- `text/xml`

Replace or append MIME types with the Response Compression Middleware options. Note that wildcard MIME types, such as `text/*` aren't supported. The sample app adds a MIME type for `image/svg+xml` and compresses and serves the ASP.NET Core banner image (*banner.svg*).

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCompression(options =>
    {
        options.Providers.Add<BrotliCompressionProvider>();
        options.Providers.Add<GzipCompressionProvider>();
        options.Providers.Add<CustomCompressionProvider>();
        options.MimeTypes =
            ResponseCompressionDefaults.MimeTypes.Concat(
                new[] { "image/svg+xml" });
    });
}
```

Compression with secure protocol

Compressed responses over secure connections can be controlled with the `EnableForHttps` option, which is disabled by default. Using compression with dynamically generated pages can lead to security problems such as the CRIME and BREACH attacks.

Adding the Vary header

When compressing responses based on the `Accept-Encoding` header, there are potentially multiple compressed versions of the response and an uncompressed version. In order to instruct client and proxy caches that multiple versions exist and should be stored, the `Vary` header is added with an `Accept-Encoding` value. In ASP.NET Core 2.0 or later, the middleware adds the `Vary` header automatically when the response is compressed.

Middleware issue when behind an Nginx reverse proxy

When a request is proxied by Nginx, the `Accept-Encoding` header is removed. Removal of the `Accept-Encoding` header prevents the middleware from compressing the response. For more information, see NGINX: Compression and Decompression. This issue is tracked by Figure out pass-through compression for Nginx (aspnet/BasicMiddleware #123).

Working with IIS dynamic compression

If you have an active IIS Dynamic Compression Module configured at the server level that you would like to disable for an app, disable the module with an addition to the *web.config* file. For more information, see Disabling IIS modules.

Troubleshooting

Use a tool like Fiddler, Firebug, or Postman, which allow you to set the `Accept-Encoding` request header and study the response headers, size, and body. By default, Response Compression Middleware compresses responses that meet the following conditions:

- The `Accept-Encoding` header is present with a value of `gzip`, `*`, or custom encoding that matches a custom

compression provider that you've established. The value must not be `identity` or have a quality value (qvalue, `q`) setting of 0 (zero).

- The MIME type (`Content-Type`) must be set and must match a MIME type configured on the ResponseCompressionOptions.
- The request must not include the `Content-Range` header.
- The request must use insecure protocol (http), unless secure protocol (https) is configured in the Response Compression Middleware options. *Note the danger described above when enabling secure content compression.*

Additional resources

- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Mozilla Developer Network: Accept-Encoding
- RFC 7231 Section 3.1.2.1: Content Codings
- RFC 7230 Section 4.2.3: Gzip Coding
- GZIP file format specification version 4.3

Performance Diagnostic Tools

9/22/2020 • 2 minutes to read • Edit Online

By Mike Rousos

This article lists tools for diagnosing performance issues in ASP.NET Core.

Visual Studio Diagnostic Tools

The profiling and diagnostic tools built into Visual Studio are a good place to start investigating performance issues. These tools are powerful and convenient to use from the Visual Studio development environment. The tooling allows analysis of CPU usage, memory usage, and performance events in ASP.NET Core apps. Being built-in makes profiling easy at development time.

More information is available in Visual Studio documentation.

Application Insights

Application Insights provides in-depth performance data for your app. Application Insights automatically collects data on response rates, failure rates, dependency response times, and more. Application Insights supports logging custom events and metrics specific to your app.

Azure Application Insights provides multiple ways to give insights on monitored apps:

- Application Map – helps spot performance bottlenecks or failure hot-spots across all components of distributed apps.
- Azure Metrics Explorer is a component of the Microsoft Azure portal that allows plotting charts, visually correlating trends, and investigating spikes and dips in metrics' values.
- Performance blade in Application Insights portal:
  - Shows performance details for different operations in the monitored app.
  - Allows drilling into a single operation to check all parts/dependencies that contribute to a long duration.
  - Profiler can be invoked from here to collect performance traces on-demand.
- Azure Application Insights Profiler allows regular and on-demand profiling of .NET apps. Azure portal shows captured performance traces with call stacks and hot paths. The trace files can also be downloaded for deeper analysis using PerfView.

Application Insights can be used in a variety environments:

- Optimized to work in Azure.
- Works in production, development, and staging.
- Works locally from Visual Studio or in other hosting environments.

For more information, see Application Insights for ASP.NET Core.

PerfView

PerfView is a performance analysis tool created by the .NET team specifically for diagnosing .NET performance issues. PerfView allows analysis of CPU usage, memory and GC behavior, performance events, and wall clock time.

You can learn more about PerfView and how to get started with PerfView video tutorials or by reading the user's

guide available in the tool or on GitHub.

Windows Performance Toolkit

Windows Performance Toolkit (WPT) consists of two components: Windows Performance Recorder (WPR) and Windows Performance Analyzer (WPA). The tools produce in-depth performance profiles of Windows operating systems and apps. WPT has richer ways of visualizing data, but its data collecting is less powerful than PerfView's.

PerfCollect

While PerfView is a useful performance analysis tool for .NET scenarios, it only runs on Windows so you can't use it to collect traces from ASP.NET Core apps running in Linux environments.

PerfCollect is a bash script that uses native Linux profiling tools (Perf and LTTng) to collect traces on Linux that can be analyzed by PerfView. PerfCollect is useful when performance problems show up in Linux environments where PerfView can't be used directly. Instead, PerfCollect can collect traces from .NET Core apps that are then analyzed on a Windows computer using PerfView.

More information about how to install and get started with PerfCollect is available on GitHub.

Other Third-party Performance Tools

The following lists some third-party performance tools that are useful in performance investigation of .NET Core applications.

- MiniProfiler
- dotTrace and dotMemory from JetBrains
- VTune from Intel

ASP.NET Core load/stress testing

9/22/2020 • 2 minutes to read • Edit Online

Load testing and stress testing are important to ensure a web app is performant and scalable. Their goals are different even though they often share similar tests.

Load tests: Test whether the app can handle a specified load of users for a certain scenario while still satisfying the response goal. The app is run under normal conditions.

Stress tests: Test app stability when running under extreme conditions, often for a long period of time. The tests place high user load, either spikes or gradually increasing load, on the app, or they limit the app's computing resources.

Stress tests determine if an app under stress can recover from failure and gracefully return to expected behavior. Under stress, the app isn't run under normal conditions.

Visual Studio 2019 announced plans to deprecate the load testing. The corresponding Azure DevOps cloud-based load testing service has been closed.

Third-party tools

The following list contains third-party web performance tools with various feature sets:

- Apache JMeter
- ApacheBench (ab)
- Gatling
- k6
- Locust
- West Wind WebSurge
- Netling
- Vegeta

Globalization and localization in ASP.NET Core

9/22/2020 • 52 minutes to read • Edit Online

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

A multilingual website allows the site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

Internationalization involves Globalization and Localization. Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas.

Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

1. Make the app's content localizable
2. Provide localized resources for the languages and cultures you support
3. Implement a strategy to select the language/culture for each request

View or download sample code (how to download)

Make the app's content localizable

IStringLocalizer and IStringLocalizer<T> were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require storing the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development. The code below shows how to wrap the string "About Title" for localization.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.Controllers
{
    [Route("api/[controller]")]
    public class AboutController : Controller
    {
        private readonly IStringLocalizer<AboutController> _localizer;

        public AboutController(IStringLocalizer<AboutController> localizer)
        {
            _localizer = localizer;
        }

        [HttpGet]
        public string Get()
        {
            return _localizer["About Title"];
        }
    }
}
```

In the preceding code, the `IStringLocalizer<T>` implementation comes from Dependency Injection. If the localized value of "About Title" isn't found, then the indexer key is returned, that is, the string "About Title". You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop your app with your default language and prepare it for the localization step without first creating a default resource file. Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers the new workflow of not having a default language *.resx* file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers will prefer the traditional work flow as it can make it easier to work with longer string literals and make it easier to update localized strings.

Use the `IHtmlLocalizer<T>` implementation for resources that contain HTML. `IHtmlLocalizer` HTML encodes arguments that are formatted in the resource string, but doesn't HTML encode the resource string itself. In the sample highlighted below, only the value of `name` parameter is HTML encoded.

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.Controllers
{
    public class BookController : Controller
    {
        private readonly IHtmlLocalizer<BookController> _localizer;

        public BookController(IHtmlLocalizer<BookController> localizer)
        {
            _localizer = localizer;
        }

        public IActionResult Hello(string name)
        {
            ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

            return View();
        }
```

NOTE

Generally, only localize text, not HTML.

At the lowest level, you can get `IStringLocalizerFactory` out of Dependency Injection:

```
{
    public class TestController : Controller
    {
        private readonly IStringLocalizer _localizer;
        private readonly IStringLocalizer _localizer2;

        public TestController(IStringLocalizerFactory factory)
        {
            var type = typeof(SharedResource);
            var assemblyName = new AssemblyName(type.GetTypeInfo().Assembly.FullName);
            _localizer = factory.Create(type);
            _localizer2 = factory.Create("SharedResource", assemblyName.Name);
        }

        public IActionResult About()
        {
            ViewData["Message"] = _localizer["Your application description page."]
                + " loc 2: " + _localizer2["Your application description page."];
```

The code above demonstrates each of the two factory create methods.

You can partition your localized strings by controller, area, or have just one container. In the sample app, a dummy class named `SharedResource` is used for shared resources.

```
// Dummy class to group shared resources

namespace Localization
{
    public class SharedResource
    {
    }
}
```

Some developers use the `Startup` class to contain global or shared strings. In the sample below, the `InfoController` and the `SharedResource` localizers are used:

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                          IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
```

View localization

The `IViewLocalizer` service provides localized strings for a view. The `ViewLocalizer` class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

```
@using Microsoft.AspNetCore.Mvc.Localization

@inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There's no option to use a global shared resource file. `ViewLocalizer` implements the localizer using `IHtmlLocalizer`, so Razor doesn't HTML encode the localized string. You can parameterize resource strings and `IViewLocalizer` will HTML encode the parameters, but not the resource string. Consider the following Razor markup:

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following:

KEY	VALUE
`<i>Hello</i> <b>{0}!</b>`	`<i>Bonjour</i> <b>{0} !</b>`

The rendered view would contain the HTML markup from the resource file.

> **NOTE**
> Generally, only localize text, not HTML.

To use a shared resource file in a view, inject `IHtmlLocalizer<T>`:

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

DataAnnotations error messages are localized with `IStringLocalizer<T>`. Using the option `ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

- *Resources/ViewModels.Account.RegisterViewModel.fr.resx*
- *Resources/ViewModels/Account/RegisterViewModel.fr.resx*

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid email address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

In ASP.NET Core MVC 1.1.0 and higher, non-validation attributes are localized. ASP.NET Core MVC 1.0 does **not** look up localized strings for non-validation attributes.

Using one resource string for multiple classes

The following code shows how to use one resource string for validation attributes with multiple classes:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```

In the preceding code, `SharedResource` is the class corresponding to the resx where your validation messages are stored. With this approach, DataAnnotations will only use `SharedResource`, rather than the resource for each class.

Provide localized resources for the languages and cultures you support

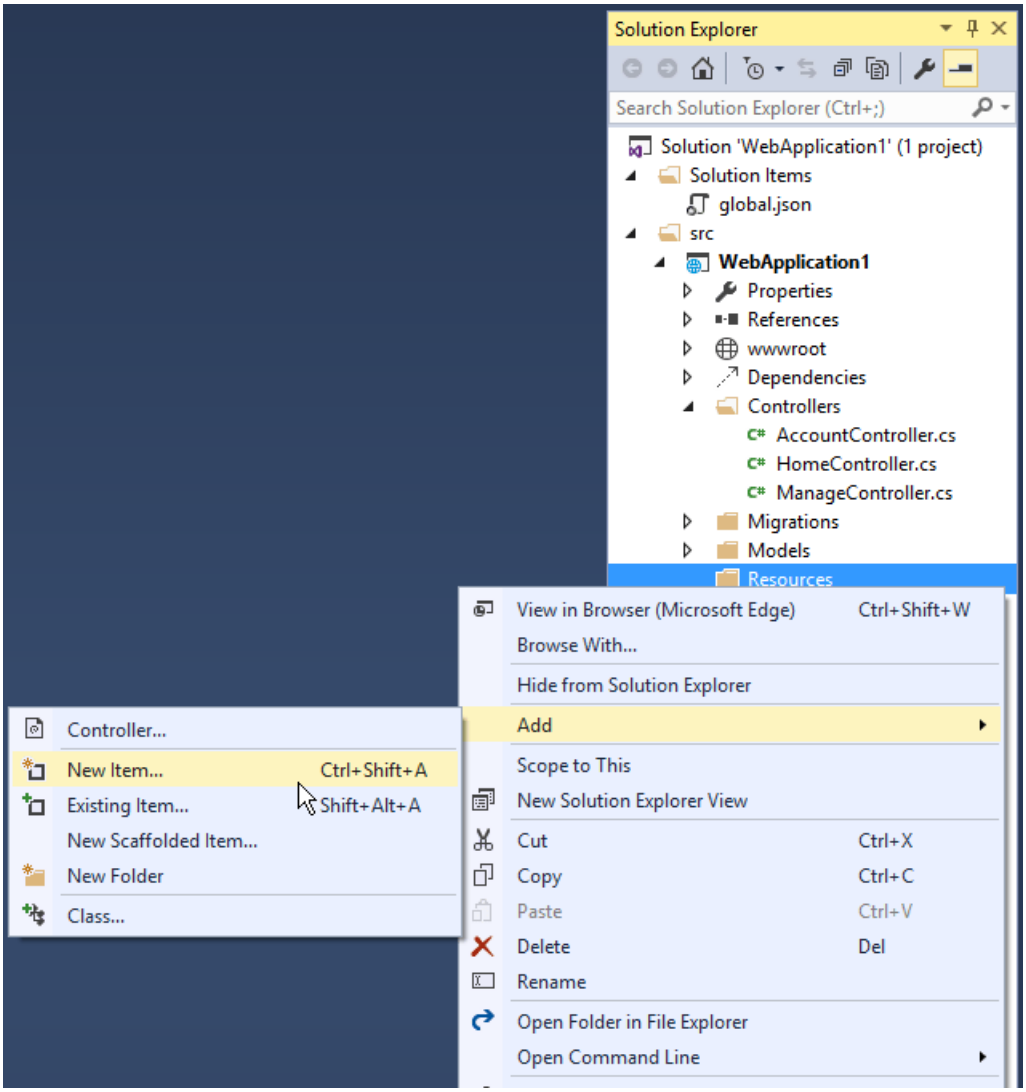
SupportedCultures and SupportedUICultures

ASP.NET Core allows you to specify two culture values, `SupportedCultures` and `SupportedUICultures`. The CultureInfo object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See CultureInfo.CurrentCulture for more info on how the server gets the Culture. The `SupportedUICultures` determines which translated strings (from *.resx* files) are looked up by the ResourceManager. The `ResourceManager` simply looks up culture-specific strings that's determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18, 2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

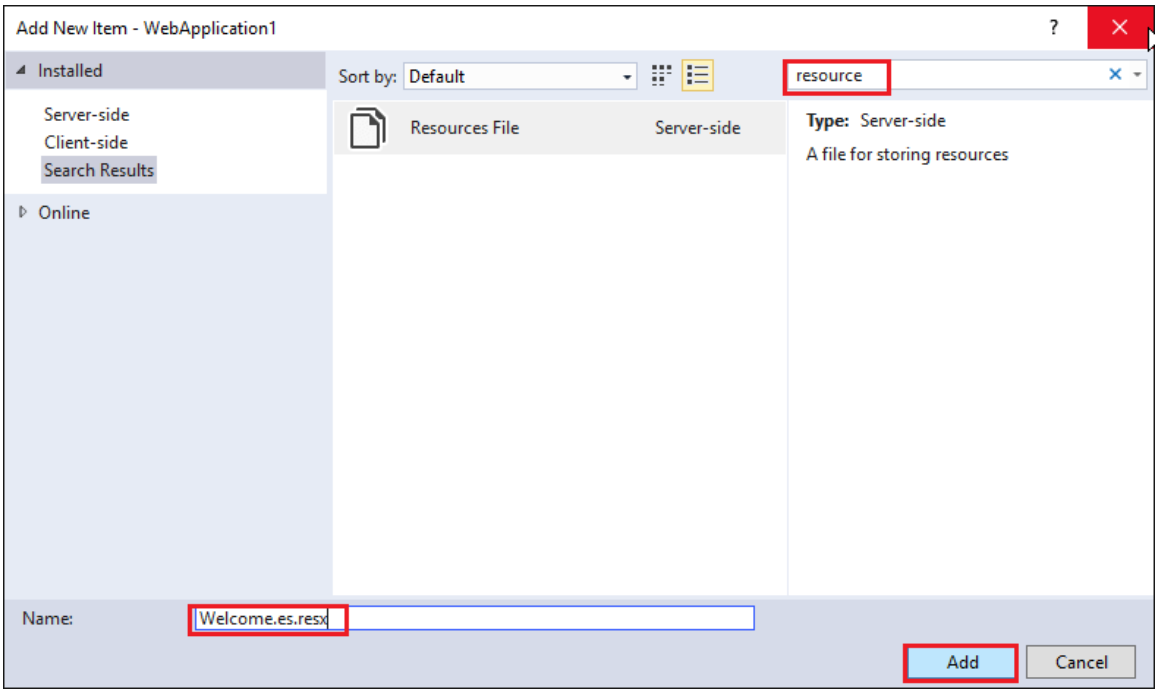
Resource files

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated in *.resx* resource files. For example, you might want to create Spanish resource file named *Welcome.es.resx* containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

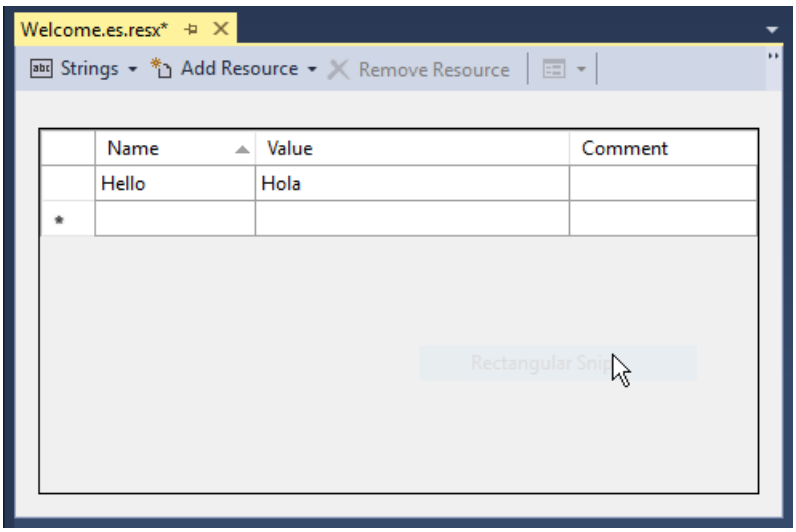
1. In **Solution Explorer**, right click on the folder which will contain the resource file > **Add** > **New Item**.



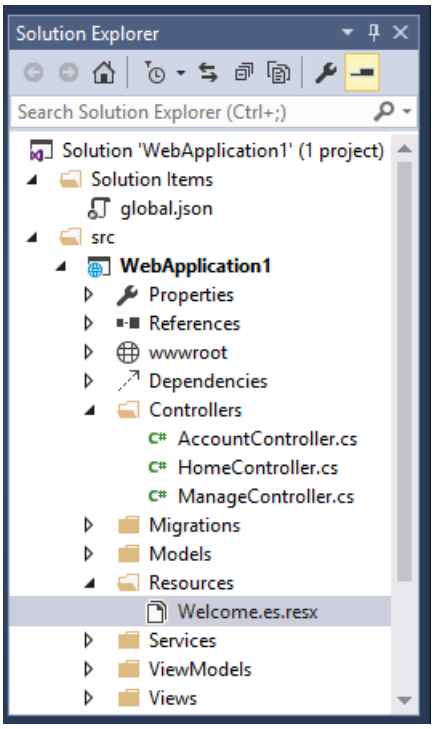
2. In the **Search installed templates** box, enter "resource" and name the file.



3. Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.



Visual Studio shows the *Welcome.es.resx* file.



Resource file naming

Resources are named for the full type name of their class minus the assembly name. For example, a French resource in a project whose main assembly is `LocalizationWebsite.Web.dll` for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. A resource for the class `LocalizationWebsite.Web.Controllers.HomeController` would be named *Controllers.HomeController.fr.resx*. If your targeted class's namespace isn't the same as the assembly name you will need the full type name. For example, in the sample project a resource for the type `ExtraNamespace.Tools` would be named *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

RESOURCE NAME	DOT OR PATH NAMING
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the *Views/Home/About.cshtml* view could be either of the following:

- Resources/Views/Home/About.fr.resx
- Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the

view.

RootNamespaceAttribute

The RootNamespace attribute provides the root namespace of an assembly when the root namespace of an assembly is different than the assembly name.

> **WARNING**
> This can occur when a project's name is not a valid .NET identifier. For instance `my-project-name.csproj` will use the root namespace `my_project_name` and the assembly name `my-project-name` leading to this error.

If the root namespace of an assembly is different than the assembly name:

- Localization does not work by default.
- Localization fails due to the way resources are searched for within the assembly. `RootNamespace` is a build-time value which is not available to the executing process.

If the `RootNamespace` is different from the `AssemblyName`, include the following in *AssemblyInfo.cs* (with parameter values replaced with the actual values):

```
using System.Reflection;
using Microsoft.Extensions.Localization;

[assembly: ResourceLocation("Resource Folder Name")]
[assembly: RootNamespace("App Root Namespace")]
```

The preceding code enables the successful resolution of resx files.

Culture fallback behavior

When searching for a resource, localization engages in "culture fallback". Starting from the requested culture, if not found, it reverts to the parent culture of that culture. As an aside, the CultureInfo.Parent property represents the parent culture. This usually (but not always) means removing the national signifier from the ISO. For example, the dialect of Spanish spoken in Mexico is "es-MX". It has the parent "es"—Spanish non-specific to any country.

Imagine your site receives a request for a "Welcome" resource using culture "fr-CA". The localization system looks for the following resources, in order, and selects the first match:

- *Welcome.fr-CA.resx*
- *Welcome.fr.resx*
- *Welcome.resx* (if the `NeutralResourcesLanguage` is "fr-CA")

As an example, if you remove the ".fr" culture designator and you have the culture set to French, the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource for when nothing meets your requested culture. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Generate resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core. You typically don't have a default *.resx* resource file (a *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio won't generate the class file.

Add other cultures

Each language and culture combination (other than the default language) requires a unique resource file. You create resource files for different cultures and locales by creating new resource files in which the ISO language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These ISO codes are placed between the file name and the *.resx* file extension, as in *Welcome.es-MX.resx* (Spanish/Mexico).

Implement a strategy to select the language/culture for each request

Configure localization

Localization is configured in the `Startup.ConfigureServices` method:

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- `AddLocalization` adds the localization services to the services container. The code above also sets the resources path to "Resources".
- `AddViewLocalization` adds support for localized view files. In this sample view localization is based on the view file suffix. For example "fr" in the *Index.fr.cshtml* file.
- `AddDataAnnotationsLocalization` adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

Localization middleware

The current culture on a request is set in the localization Middleware. The localization middleware is enabled in the `Startup.Configure` method. The localization middleware must be configured before any middleware which might check the request culture (for example, `app.UseMvcWithDefaultRoute()`).

```
var supportedCultures = new[] { "en-US", "fr" };
var localizationOptions = new RequestLocalizationOptions().SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);

app.UseRouting();
app.UseStaticFiles();

app.UseAuthentication();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

`UseRequestLocalization` initializes a `RequestLocalizationOptions` object. On every request the list of `RequestCultureProvider` in the `RequestLocalizationOptions` is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

1. `QueryStringRequestCultureProvider`

2. `CookieRequestCultureProvider`
3. `AcceptLanguageHeaderRequestCultureProvider`

The default list goes from most specific to least specific. Later in the article we'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the `DefaultRequestCulture` is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the /dotnet/api/system.globalization.cultureinfo?view=netcore-3.1. For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the `QueryStringRequestCultureProvider` is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

`http://localhost:5000/?culture=es-MX&ui-culture=es-MX`

If you only pass in one of the two (`culture` or `ui-culture`), the query string provider will set both values using the one you passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

```
http://localhost:5000/?culture=es-MX
```

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the `MakeCookieValue` method to create a cookie.

The `CookieRequestCultureProvider` `DefaultCookieName` returns the default cookie name used to track the user's preferred culture information. The default cookie name is `.AspNetCore.Culture`.

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

```
c=en-UK|uic=en-US
```

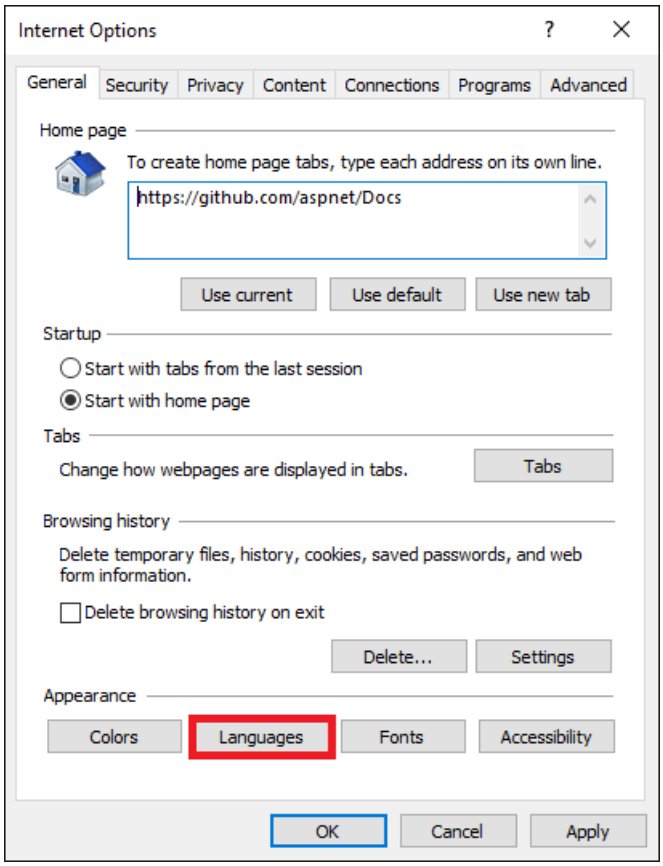
If you only specify one of culture info and UI culture, the specified culture will be used for both culture info and UI culture.

The Accept-Language HTTP header

The Accept-Language header is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request isn't an infallible way to detect the user's preferred language (see Setting language preferences in a browser). A production app should include a way for a user to customize their choice of culture.

Set the Accept-Language HTTP header in IE

1. From the gear icon, tap **Internet Options**.
2. Tap **Languages**.



3. Tap **Set Language Preferences**.
4. Tap **Add a language**.
5. Add the language.
6. Tap the language, then tap **Move Up**.

Use a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

```
private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture, uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.AddInitialRequestCultureProvider(new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
});
```

Use `RequestLocalizationOptions` to add or remove localization providers.

Set the culture programmatically

This sample **Localization.StarterWeb** project on GitHub contains UI to set the `Culture`. The *Views/Shared/_SelectLanguagePartial.cshtml* file allows you to select the culture from the list of supported cultures:

```
@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~/" : $"~{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
          method="post" class="form-horizontal" role="form">
        <label asp-for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]</label>
<select name="culture"
          onchange="this.form.submit();"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
        </select>
    </form>
</div>
```

The *Views/Shared/_SelectLanguagePartial.cshtml* file is added to the `footer` section of the layout file so it will be available to all views:

```
<div class="container body-content" style="margin-top:60px">
    @RenderBody()
    <hr>
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; @System.DateTime.Now.Year - Localization</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

The `SetLanguage` method sets the culture cookie.

```
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

You can't plug in the *_SelectLanguagePartial.cshtml* to sample code for this project. The **Localization.StarterWeb** project on GitHub has code to flow the `RequestLocalizationOptions` to a Razor partial through the Dependency Injection container.

Model binding route data and query strings

See Globalization behavior of model binding route data and query strings.

Globalization and localization terms

The process of localizing your app also requires a basic understanding of relevant character sets commonly used in modern software development and an understanding of the issues associated with them. Although all computers store text as numbers (codes), different systems store the same text using different numbers. The localization process refers to translating the app user interface (UI) for a specific culture/locale.

Localizability is an intermediate process for verifying that a globalized app is ready for localization.

The RFC 4646 format for the culture name is `<languagecode2>-<country/regioncode2>`, where `<languagecode2>` is the language code and `<country/regioncode2>` is the subculture code. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). RFC 4646 is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. See /previous-versions/commerce-server/ee825488(v=cs.20).

Internationalization is often abbreviated to "I18N". The abbreviation takes the first and last letters and the number of letters between them, so 18 stands for the number of letters between the first "I" and the last "N". The same applies to Globalization (G11N), and Localization (L10N).

Terms:

- Globalization (G11N): The process of making an app support different languages and regions.
- Localization (L10N): The process of customizing an app for a given language and region.
- Internationalization (I18N): Describes both globalization and localization.
- Culture: It's a language and, optionally, a region.
- Neutral culture: A culture that has a specified language, but not a region. (for example "en", "es")
- Specific culture: A culture that has a specified language and region. (for example "en-US", "en-GB", "es-CL")
- Parent culture: The neutral culture that contains a specific culture. (for example, "en" is the parent culture of "en-US" and "en-GB")
- Locale: A locale is the same as a culture.

NOTE

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

NOTE

Prior to ASP.NET Core 3.0 web apps write one log of type `LogLevel.Warning` per request if the requested culture is unsupported. Logging one `LogLevel.Warning` per request is can make large log files with redundant information. This behavior has been changed in ASP.NET 3.0. The `RequestLocalizationMiddleware` writes a log of type `LogLevel.Debug`, which reduces the size of production logs.

Additional resources

- Troubleshoot ASP.NET Core Localization
- Localization.StarterWeb project used in the article.
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

A multilingual website allows the site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

Internationalization involves Globalization and Localization. Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas.

Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

1. Make the app's content localizable
2. Provide localized resources for the languages and cultures you support
3. Implement a strategy to select the language/culture for each request

View or download sample code (how to download)

Make the app's content localizable

IStringLocalizer and IStringLocalizer<T> were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require storing the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development. The code below shows how to wrap the string "About Title" for localization.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.Controllers
{
    [Route("api/[controller]")]
    public class AboutController : Controller
    {
        private readonly IStringLocalizer<AboutController> _localizer;

        public AboutController(IStringLocalizer<AboutController> localizer)
        {
            _localizer = localizer;
        }

        [HttpGet]
        public string Get()
        {
            return _localizer["About Title"];
        }
    }
}
```

In the preceding code, the `IStringLocalizer<T>` implementation comes from Dependency Injection. If the localized value of "About Title" isn't found, then the indexer key is returned, that is, the string "About Title". You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop your app with your default language and prepare it for the localization step without first creating a default resource file. Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers the new workflow of not having a default language *.resx* file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers will prefer the traditional work flow as it can make it easier to work with longer string literals and make it easier to update localized strings.

Use the `IHtmlLocalizer<T>` implementation for resources that contain HTML. `IHtmlLocalizer` HTML encodes arguments that are formatted in the resource string, but doesn't HTML encode the resource string itself. In the sample highlighted below, only the value of `name` parameter is HTML encoded.

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.Controllers
{
    public class BookController : Controller
    {
        private readonly IHtmlLocalizer<BookController> _localizer;

        public BookController(IHtmlLocalizer<BookController> localizer)
        {
            _localizer = localizer;
        }

        public IActionResult Hello(string name)
        {
            ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

            return View();
        }
```

> **NOTE**
> Generally, only localize text, not HTML.

At the lowest level, you can get `IStringLocalizerFactory` out of Dependency Injection:

```
{
    public class TestController : Controller
    {
        private readonly IStringLocalizer _localizer;
        private readonly IStringLocalizer _localizer2;

        public TestController(IStringLocalizerFactory factory)
        {
            var type = typeof(SharedResource);
            var assemblyName = new AssemblyName(type.GetTypeInfo().Assembly.FullName);
            _localizer = factory.Create(type);
            _localizer2 = factory.Create("SharedResource", assemblyName.Name);
        }

        public IActionResult About()
        {
            ViewData["Message"] = _localizer["Your application description page."]
                + " loc 2: " + _localizer2["Your application description page."];
```

The code above demonstrates each of the two factory create methods.

You can partition your localized strings by controller, area, or have just one container. In the sample app, a dummy class named `SharedResource` is used for shared resources.

```
// Dummy class to group shared resources

namespace Localization
{
    public class SharedResource
    {
    }
}
```

Some developers use the `Startup` class to contain global or shared strings. In the sample below, the `InfoController` and the `SharedResource` localizers are used:

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                          IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
```

View localization

The `IViewLocalizer` service provides localized strings for a view. The `ViewLocalizer` class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

```
@using Microsoft.AspNetCore.Mvc.Localization

@inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There's no option to use a global shared resource file. `ViewLocalizer` implements the localizer using `IHtmlLocalizer`, so Razor doesn't HTML encode the localized string. You can parameterize resource strings and `IViewLocalizer` will HTML encode the parameters, but not the resource string. Consider the following Razor markup:

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following:

KEY	VALUE
`<i>Hello</i> <b>{0}!</b>`	`<i>Bonjour</i> <b>{0} !</b>`

The rendered view would contain the HTML markup from the resource file.

> **NOTE**
> Generally, only localize text, not HTML.

To use a shared resource file in a view, inject `IHtmlLocalizer<T>`:

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

DataAnnotations error messages are localized with `IStringLocalizer<T>`. Using the option

`ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

- *Resources/ViewModels.Account.RegisterViewModel.fr.resx*
- *Resources/ViewModels/Account/RegisterViewModel.fr.resx*

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid email address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

In ASP.NET Core MVC 1.1.0 and higher, non-validation attributes are localized. ASP.NET Core MVC 1.0 does **not** look up localized strings for non-validation attributes.

Using one resource string for multiple classes

The following code shows how to use one resource string for validation attributes with multiple classes:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```

In the preceding code, `SharedResource` is the class corresponding to the resx where your validation messages are stored. With this approach, DataAnnotations will only use `SharedResource`, rather than the resource for each class.

Provide localized resources for the languages and cultures you support

SupportedCultures and SupportedUICultures

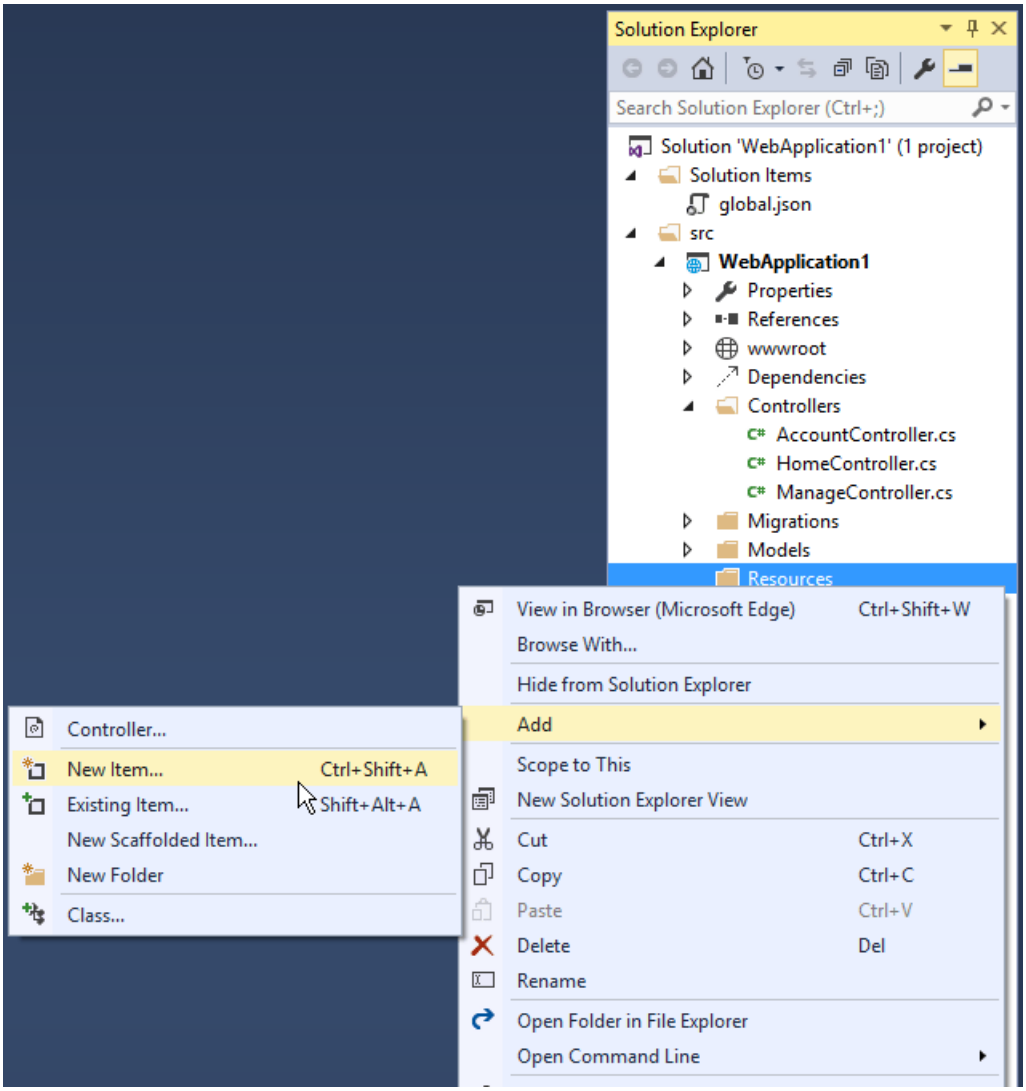
ASP.NET Core allows you to specify two culture values, `SupportedCultures` and `SupportedUICultures`. The CultureInfo object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See CultureInfo.CurrentCulture for more info on how the server gets the Culture. The `SupportedUICultures` determines which translated strings (from *.resx* files) are looked up by the ResourceManager. The `ResourceManager` simply looks up culture-specific strings that's determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18,

2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

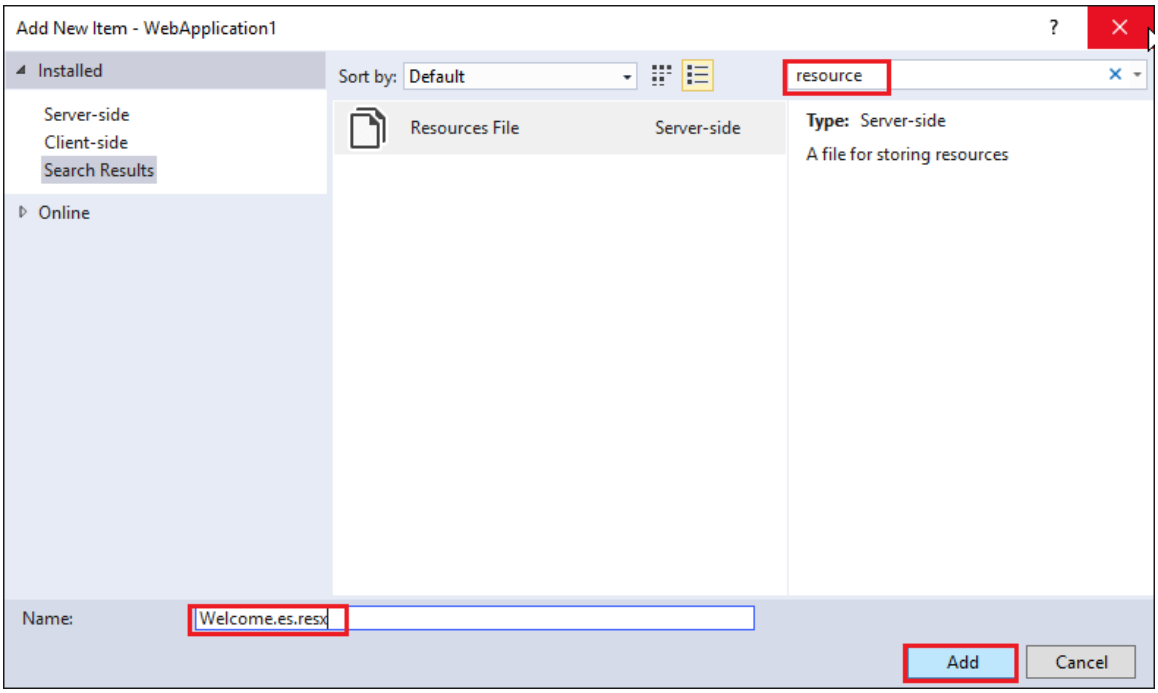
Resource files

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated in *.resx* resource files. For example, you might want to create Spanish resource file named *Welcome.es.resx* containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

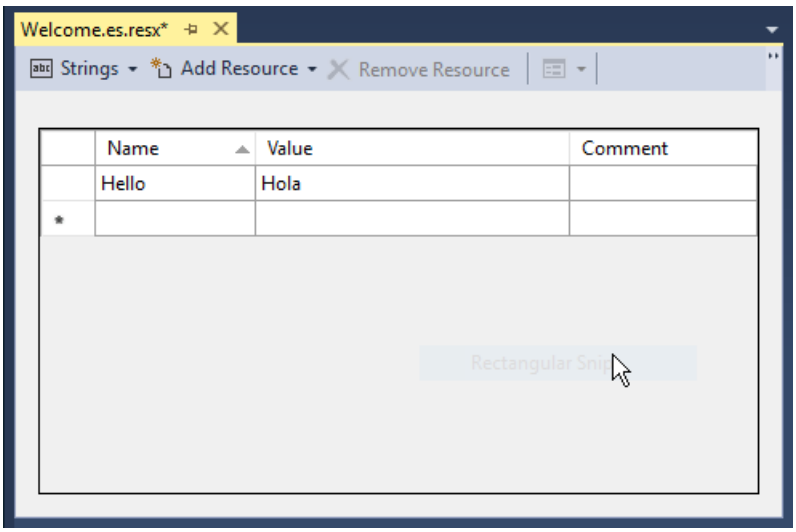
1. In **Solution Explorer**, right click on the folder which will contain the resource file > **Add** > **New Item**.



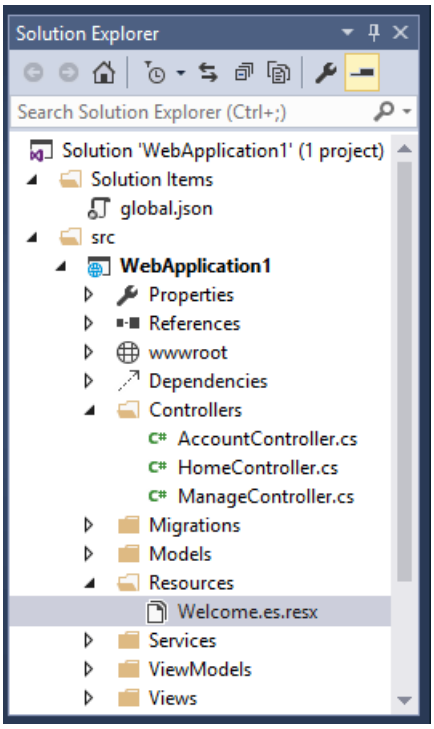
2. In the **Search installed templates** box, enter "resource" and name the file.



3. Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.



Visual Studio shows the *Welcome.es.resx* file.



Resource file naming

Resources are named for the full type name of their class minus the assembly name. For example, a French resource in a project whose main assembly is `LocalizationWebsite.Web.dll` for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. A resource for the class `LocalizationWebsite.Web.Controllers.HomeController` would be named *Controllers.HomeController.fr.resx*. If your targeted class's namespace isn't the same as the assembly name you will need the full type name. For example, in the sample project a resource for the type `ExtraNamespace.Tools` would be named *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

RESOURCE NAME	DOT OR PATH NAMING
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the *Views/Home/About.cshtml* view could be either of the following:

- Resources/Views/Home/About.fr.resx
- Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the

view.

RootNamespaceAttribute

The RootNamespace attribute provides the root namespace of an assembly when the root namespace of an assembly is different than the assembly name.

> **WARNING**
> This can occur when a project's name is not a valid .NET identifier. For instance `my-project-name.csproj` will use the root namespace `my_project_name` and the assembly name `my-project-name` leading to this error.

If the root namespace of an assembly is different than the assembly name:

- Localization does not work by default.
- Localization fails due to the way resources are searched for within the assembly. `RootNamespace` is a build-time value which is not available to the executing process.

If the `RootNamespace` is different from the `AssemblyName`, include the following in *AssemblyInfo.cs* (with parameter values replaced with the actual values):

```
using System.Reflection;
using Microsoft.Extensions.Localization;

[assembly: ResourceLocation("Resource Folder Name")]
[assembly: RootNamespace("App Root Namespace")]
```

The preceding code enables the successful resolution of resx files.

Culture fallback behavior

When searching for a resource, localization engages in "culture fallback". Starting from the requested culture, if not found, it reverts to the parent culture of that culture. As an aside, the CultureInfo.Parent property represents the parent culture. This usually (but not always) means removing the national signifier from the ISO. For example, the dialect of Spanish spoken in Mexico is "es-MX". It has the parent "es"—Spanish non-specific to any country.

Imagine your site receives a request for a "Welcome" resource using culture "fr-CA". The localization system looks for the following resources, in order, and selects the first match:

- *Welcome.fr-CA.resx*
- *Welcome.fr.resx*
- *Welcome.resx* (if the `NeutralResourcesLanguage` is "fr-CA")

As an example, if you remove the ".fr" culture designator and you have the culture set to French, the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource for when nothing meets your requested culture. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Generate resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core. You typically don't have a default *.resx* resource file (a *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio won't generate the class file.

Add other cultures

Each language and culture combination (other than the default language) requires a unique resource file. You create resource files for different cultures and locales by creating new resource files in which the ISO language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These ISO codes are placed between the file name and the *.resx* file extension, as in *Welcome.es-MX.resx* (Spanish/Mexico).

Implement a strategy to select the language/culture for each request

Configure localization

Localization is configured in the `Startup.ConfigureServices` method:

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- `AddLocalization` adds the localization services to the services container. The code above also sets the resources path to "Resources".
- `AddViewLocalization` adds support for localized view files. In this sample view localization is based on the view file suffix. For example "fr" in the *Index.fr.cshtml* file.
- `AddDataAnnotationsLocalization` adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

Localization middleware

The current culture on a request is set in the localization Middleware. The localization middleware is enabled in the `Startup.Configure` method. The localization middleware must be configured before any middleware which might check the request culture (for example, `app.UseMvcWithDefaultRoute()`).

```
var supportedCultures = new[] { "en-US", "fr" };
var localizationOptions = new RequestLocalizationOptions().SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);

app.UseRouting();
app.UseStaticFiles();

app.UseAuthentication();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

`UseRequestLocalization` initializes a `RequestLocalizationOptions` object. On every request the list of `RequestCultureProvider` in the `RequestLocalizationOptions` is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

1. `QueryStringRequestCultureProvider`

2. `CookieRequestCultureProvider`
3. `AcceptLanguageHeaderRequestCultureProvider`

The default list goes from most specific to least specific. Later in the article we'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the `DefaultRequestCulture` is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the /dotnet/api/system.globalization.cultureinfo?view=netcore-3.1. For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the `QueryStringRequestCultureProvider` is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

If you only pass in one of the two (`culture` or `ui-culture`), the query string provider will set both values using the one you passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

```
http://localhost:5000/?culture=es-MX
```

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the `MakeCookieValue` method to create a cookie.

The `CookieRequestCultureProvider` `DefaultCookieName` returns the default cookie name used to track the user's preferred culture information. The default cookie name is `.AspNetCore.Culture`.

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

```
c=en-UK|uic=en-US
```

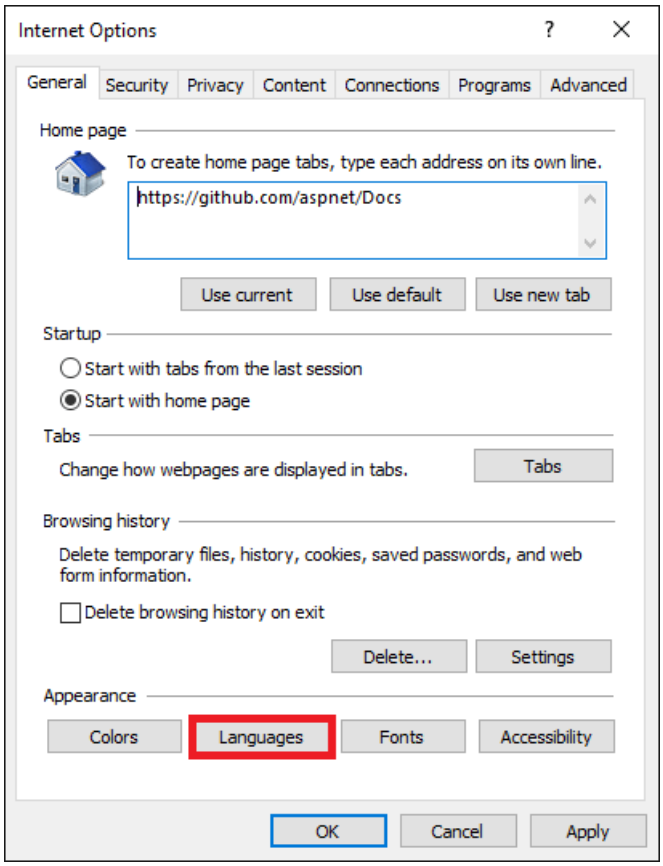
If you only specify one of culture info and UI culture, the specified culture will be used for both culture info and UI culture.

The Accept-Language HTTP header

The Accept-Language header is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request isn't an infallible way to detect the user's preferred language (see Setting language preferences in a browser). A production app should include a way for a user to customize their choice of culture.

Set the Accept-Language HTTP header in IE

1. From the gear icon, tap **Internet Options**.
2. Tap **Languages**.



3. Tap **Set Language Preferences**.
4. Tap **Add a language**.
5. Add the language.
6. Tap the language, then tap **Move Up**.

Use a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

```
private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture, uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
});
```

Use `RequestLocalizationOptions` to add or remove localization providers.

Set the culture programmatically

This sample **Localization.StarterWeb** project on GitHub contains UI to set the `Culture`. The *Views/Shared/_SelectLanguagePartial.cshtml* file allows you to select the culture from the list of supported cultures:

```
@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~/" : $"~{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
          method="post" class="form-horizontal" role="form">
        <label asp-for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]</label>
<select name="culture"
          onchange="this.form.submit();"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
        </select>
    </form>
</div>
```

The *Views/Shared/_SelectLanguagePartial.cshtml* file is added to the `footer` section of the layout file so it will be available to all views:

```
<div class="container body-content" style="margin-top:60px">
    @RenderBody()
    <hr>
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; @System.DateTime.Now.Year - Localization</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

The `SetLanguage` method sets the culture cookie.

```
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

You can't plug in the *_SelectLanguagePartial.cshtml* to sample code for this project. The **Localization.StarterWeb** project on GitHub has code to flow the `RequestLocalizationOptions` to a Razor partial through the Dependency Injection container.

Model binding route data and query strings

See Globalization behavior of model binding route data and query strings.

Globalization and localization terms

The process of localizing your app also requires a basic understanding of relevant character sets commonly used in modern software development and an understanding of the issues associated with them. Although all computers store text as numbers (codes), different systems store the same text using different numbers. The localization process refers to translating the app user interface (UI) for a specific culture/locale.

Localizability is an intermediate process for verifying that a globalized app is ready for localization.

The RFC 4646 format for the culture name is `<languagecode2>-<country/regioncode2>`, where `<languagecode2>` is the language code and `<country/regioncode2>` is the subculture code. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). RFC 4646 is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. See /previous-versions/commerce-server/ee825488(v=cs.20).

Internationalization is often abbreviated to "I18N". The abbreviation takes the first and last letters and the number of letters between them, so 18 stands for the number of letters between the first "I" and the last "N". The same applies to Globalization (G11N), and Localization (L10N).

Terms:

- Globalization (G11N): The process of making an app support different languages and regions.
- Localization (L10N): The process of customizing an app for a given language and region.
- Internationalization (I18N): Describes both globalization and localization.
- Culture: It's a language and, optionally, a region.
- Neutral culture: A culture that has a specified language, but not a region. (for example "en", "es")
- Specific culture: A culture that has a specified language and region. (for example "en-US", "en-GB", "es-CL")
- Parent culture: The neutral culture that contains a specific culture. (for example, "en" is the parent culture of "en-US" and "en-GB")
- Locale: A locale is the same as a culture.

NOTE

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

Additional resources

- Troubleshoot ASP.NET Core Localization
- Localization.StarterWeb project used in the article.
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

A multilingual website allows the site to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

Internationalization involves Globalization and Localization. Globalization is the process of designing apps that support different cultures. Globalization adds support for input, display, and output of a defined set of language scripts that relate to specific geographic areas.

Localization is the process of adapting a globalized app, which you have already processed for localizability, to a particular culture/locale. For more information see **Globalization and localization terms** near the end of this document.

App localization involves the following:

1. Make the app's content localizable
2. Provide localized resources for the languages and cultures you support
3. Implement a strategy to select the language/culture for each request

View or download sample code (how to download)

Make the app's content localizable

IStringLocalizer and IStringLocalizer<T> were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require storing the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development. The code below shows how to wrap the string "About Title" for localization.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.Controllers
{
    [Route("api/[controller]")]
    public class AboutController : Controller
    {
        private readonly IStringLocalizer<AboutController> _localizer;

        public AboutController(IStringLocalizer<AboutController> localizer)
        {
            _localizer = localizer;
        }

        [HttpGet]
        public string Get()
        {
            return _localizer["About Title"];
        }
    }
}
```

In the preceding code, the `IStringLocalizer<T>` implementation comes from Dependency Injection. If the localized value of "About Title" isn't found, then the indexer key is returned, that is, the string "About Title". You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop your app with your default language and prepare it for the localization step without first creating a default resource file. Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers the new workflow of not having a default language *.resx* file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers will prefer the traditional work flow as it can make it easier to work with longer string literals and make it easier to update localized strings.

Use the `IHtmlLocalizer<T>` implementation for resources that contain HTML. `IHtmlLocalizer` HTML encodes arguments that are formatted in the resource string, but doesn't HTML encode the resource string itself. In the sample highlighted below, only the value of `name` parameter is HTML encoded.

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.Controllers
{
    public class BookController : Controller
    {
        private readonly IHtmlLocalizer<BookController> _localizer;

        public BookController(IHtmlLocalizer<BookController> localizer)
        {
            _localizer = localizer;
        }

        public IActionResult Hello(string name)
        {
            ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

            return View();
        }
```

> **NOTE**
> Generally, only localize text, not HTML.

At the lowest level, you can get `IStringLocalizerFactory` out of Dependency Injection:

```
{
    public class TestController : Controller
    {
        private readonly IStringLocalizer _localizer;
        private readonly IStringLocalizer _localizer2;

        public TestController(IStringLocalizerFactory factory)
        {
            var type = typeof(SharedResource);
            var assemblyName = new AssemblyName(type.GetTypeInfo().Assembly.FullName);
            _localizer = factory.Create(type);
            _localizer2 = factory.Create("SharedResource", assemblyName.Name);
        }

        public IActionResult About()
        {
            ViewData["Message"] = _localizer["Your application description page."]
                + " loc 2: " + _localizer2["Your application description page."];
```

The code above demonstrates each of the two factory create methods.

You can partition your localized strings by controller, area, or have just one container. In the sample app, a dummy class named `SharedResource` is used for shared resources.

```
// Dummy class to group shared resources

namespace Localization
{
    public class SharedResource
    {
    }
}
```

Some developers use the `Startup` class to contain global or shared strings. In the sample below, the `InfoController` and the `SharedResource` localizers are used:

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                          IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
```

View localization

The `IViewLocalizer` service provides localized strings for a view. The `ViewLocalizer` class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

```
@using Microsoft.AspNetCore.Mvc.Localization

@inject IViewLocalizer Localizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There's no option to use a global shared resource file. `ViewLocalizer` implements the localizer using `IHtmlLocalizer`, so Razor doesn't HTML encode the localized string. You can parameterize resource strings and `IViewLocalizer` will HTML encode the parameters, but not the resource string. Consider the following Razor markup:

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following:

KEY	VALUE
`<i>Hello</i> <b>{0}!</b>`	`<i>Bonjour</i> <b>{0} !</b>`

The rendered view would contain the HTML markup from the resource file.

> **NOTE**
> Generally, only localize text, not HTML.

To use a shared resource file in a view, inject `IHtmlLocalizer<T>`:

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

DataAnnotations error messages are localized with `IStringLocalizer<T>`. Using the option

`ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

- *Resources/ViewModels.Account.RegisterViewModel.fr.resx*
- *Resources/ViewModels/Account/RegisterViewModel.fr.resx*

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid email address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters long.", MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

In ASP.NET Core MVC 1.1.0 and higher, non-validation attributes are localized. ASP.NET Core MVC 1.0 does **not** look up localized strings for non-validation attributes.

Using one resource string for multiple classes

The following code shows how to use one resource string for validation attributes with multiple classes:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
}
```

In the preceding code, `SharedResource` is the class corresponding to the resx where your validation messages are stored. With this approach, DataAnnotations will only use `SharedResource`, rather than the resource for each class.

Provide localized resources for the languages and cultures you support

SupportedCultures and SupportedUICultures

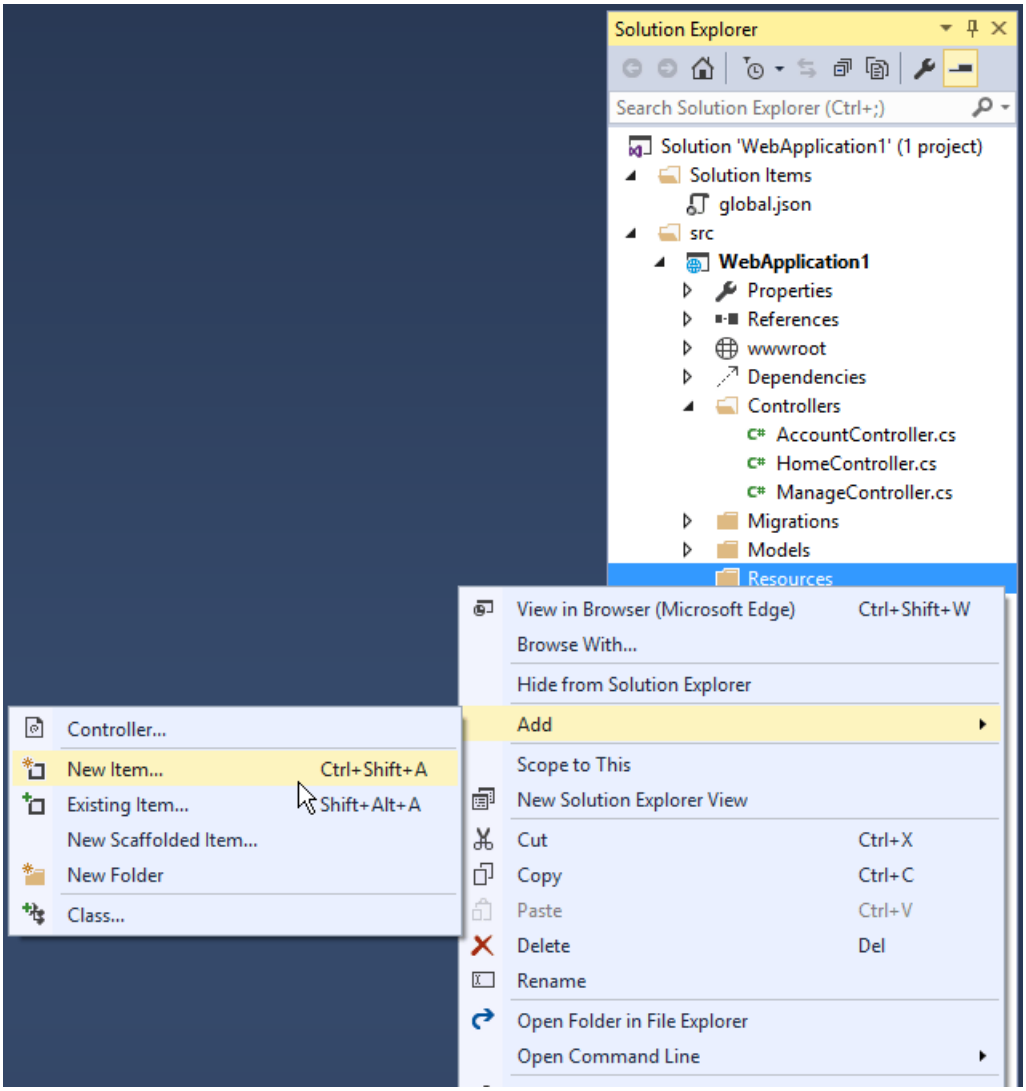
ASP.NET Core allows you to specify two culture values, `SupportedCultures` and `SupportedUICultures`. The CultureInfo object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See CultureInfo.CurrentCulture for more info on how the server gets the Culture. The `SupportedUICultures` determines which translated strings (from *.resx* files) are looked up by the ResourceManager. The `ResourceManager` simply looks up culture-specific strings that's determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18,

2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

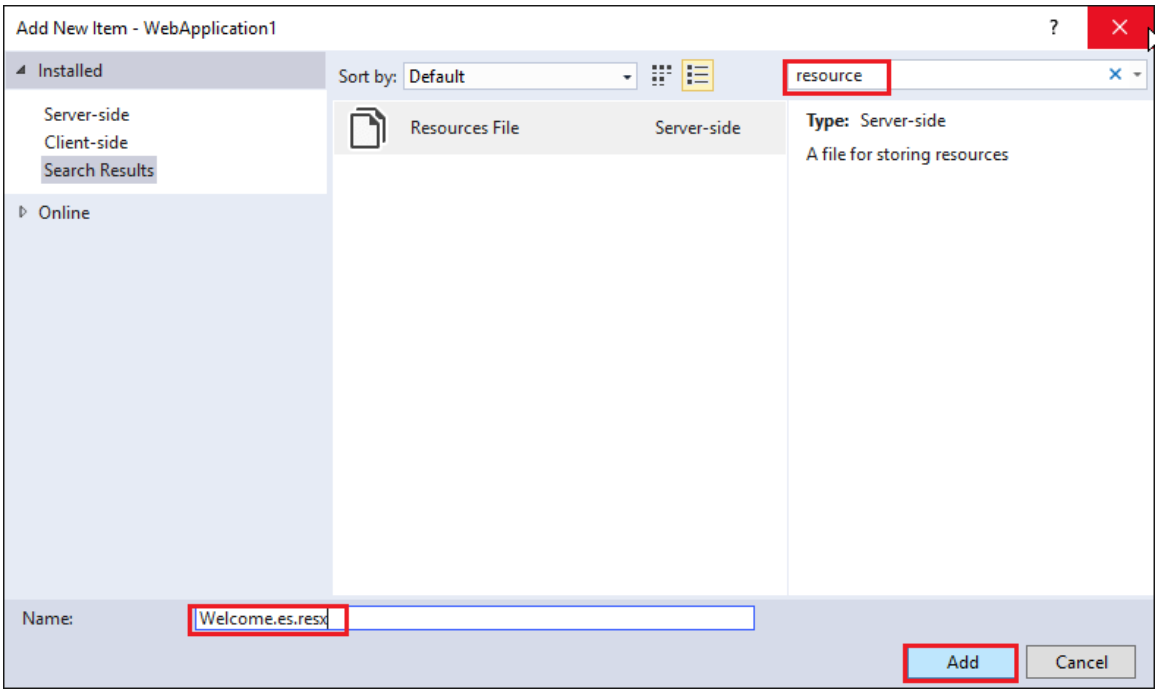
Resource files

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated in *.resx* resource files. For example, you might want to create Spanish resource file named *Welcome.es.resx* containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

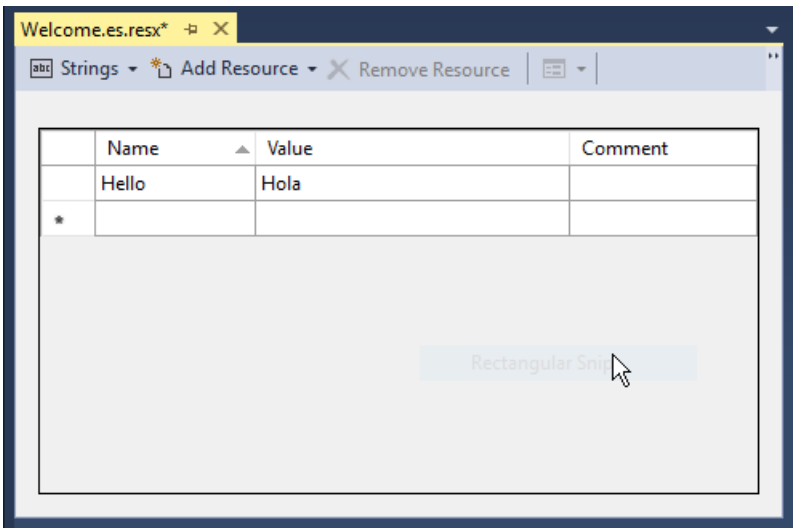
1. In **Solution Explorer**, right click on the folder which will contain the resource file > **Add** > **New Item**.



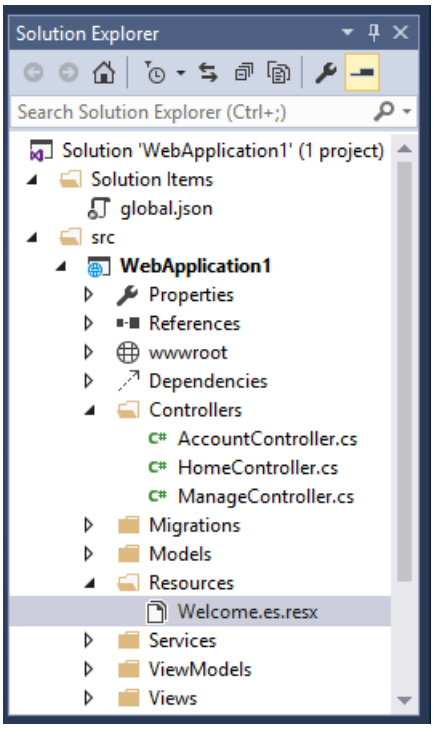
2. In the **Search installed templates** box, enter "resource" and name the file.



3. Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.



Visual Studio shows the *Welcome.es.resx* file.



Resource file naming

Resources are named for the full type name of their class minus the assembly name. For example, a French resource in a project whose main assembly is `LocalizationWebsite.Web.dll` for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. A resource for the class `LocalizationWebsite.Web.Controllers.HomeController` would be named *Controllers.HomeController.fr.resx*. If your targeted class's namespace isn't the same as the assembly name you will need the full type name. For example, in the sample project a resource for the type `ExtraNamespace.Tools` would be named *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

RESOURCE NAME	DOT OR PATH NAMING
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the *Views/Home/About.cshtml* view could be either of the following:

- Resources/Views/Home/About.fr.resx
- Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the

view.

RootNamespaceAttribute

The RootNamespace attribute provides the root namespace of an assembly when the root namespace of an assembly is different than the assembly name.

> **WARNING**
> This can occur when a project's name is not a valid .NET identifier. For instance `my-project-name.csproj` will use the root namespace `my_project_name` and the assembly name `my-project-name` leading to this error.

If the root namespace of an assembly is different than the assembly name:

- Localization does not work by default.
- Localization fails due to the way resources are searched for within the assembly. `RootNamespace` is a build-time value which is not available to the executing process.

If the `RootNamespace` is different from the `AssemblyName`, include the following in *AssemblyInfo.cs* (with parameter values replaced with the actual values):

```
using System.Reflection;
using Microsoft.Extensions.Localization;

[assembly: ResourceLocation("Resource Folder Name")]
[assembly: RootNamespace("App Root Namespace")]
```

The preceding code enables the successful resolution of resx files.

Culture fallback behavior

When searching for a resource, localization engages in "culture fallback". Starting from the requested culture, if not found, it reverts to the parent culture of that culture. As an aside, the CultureInfo.Parent property represents the parent culture. This usually (but not always) means removing the national signifier from the ISO. For example, the dialect of Spanish spoken in Mexico is "es-MX". It has the parent "es"—Spanish non-specific to any country.

Imagine your site receives a request for a "Welcome" resource using culture "fr-CA". The localization system looks for the following resources, in order, and selects the first match:

- *Welcome.fr-CA.resx*
- *Welcome.fr.resx*
- *Welcome.resx* (if the `NeutralResourcesLanguage` is "fr-CA")

As an example, if you remove the ".fr" culture designator and you have the culture set to French, the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource for when nothing meets your requested culture. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Generate resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core. You typically don't have a default *.resx* resource file (a *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio won't generate the class file.

Add other cultures

Each language and culture combination (other than the default language) requires a unique resource file. You create resource files for different cultures and locales by creating new resource files in which the ISO language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These ISO codes are placed between the file name and the *.resx* file extension, as in *Welcome.es-MX.resx* (Spanish/Mexico).

Implement a strategy to select the language/culture for each request

Configure localization

Localization is configured in the `Startup.ConfigureServices` method:

```
services.AddLocalization(options => options.ResourcesPath = "Resources");

services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- `AddLocalization` adds the localization services to the services container. The code above also sets the resources path to "Resources".
- `AddViewLocalization` adds support for localized view files. In this sample view localization is based on the view file suffix. For example "fr" in the *Index.fr.cshtml* file.
- `AddDataAnnotationsLocalization` adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

Localization middleware

The current culture on a request is set in the localization Middleware. The localization middleware is enabled in the `Startup.Configure` method. The localization middleware must be configured before any middleware which might check the request culture (for example, `app.UseMvcWithDefaultRoute()`).

```
var supportedCultures = new[] { "en-US", "fr" };
var localizationOptions = new RequestLocalizationOptions().SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);

app.UseRouting();
app.UseStaticFiles();

app.UseAuthentication();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

`UseRequestLocalization` initializes a `RequestLocalizationOptions` object. On every request the list of `RequestCultureProvider` in the `RequestLocalizationOptions` is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

1. `QueryStringRequestCultureProvider`

2. `CookieRequestCultureProvider`
3. `AcceptLanguageHeaderRequestCultureProvider`

The default list goes from most specific to least specific. Later in the article we'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the `DefaultRequestCulture` is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the /dotnet/api/system.globalization.cultureinfo?view=netcore-3.1. For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the `QueryStringRequestCultureProvider` is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

If you only pass in one of the two (`culture` or `ui-culture`), the query string provider will set both values using the one you passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

```
http://localhost:5000/?culture=es-MX
```

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the `MakeCookieValue` method to create a cookie.

The `CookieRequestCultureProvider` `DefaultCookieName` returns the default cookie name used to track the user's preferred culture information. The default cookie name is `.AspNetCore.Culture`.

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

```
c=en-UK|uic=en-US
```

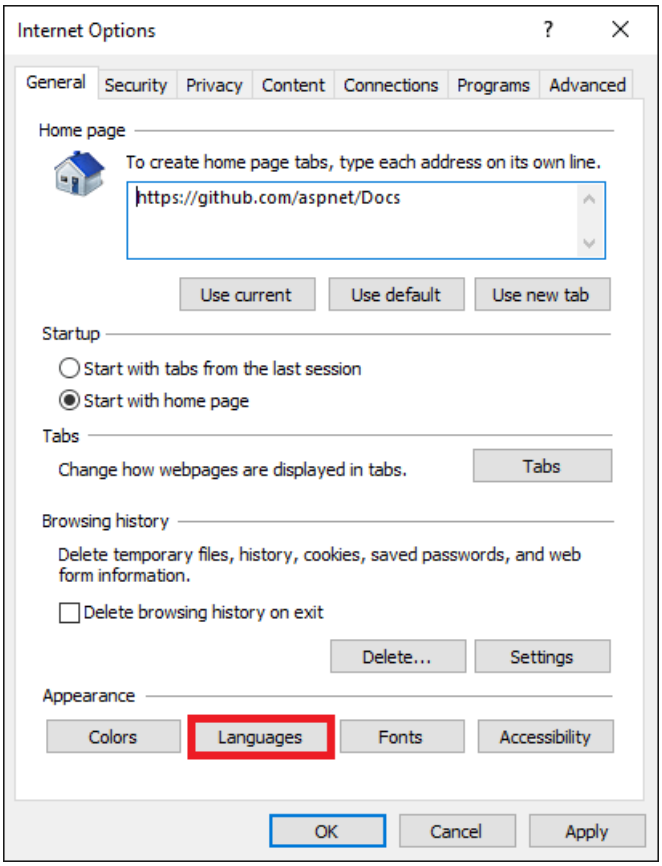
If you only specify one of culture info and UI culture, the specified culture will be used for both culture info and UI culture.

The Accept-Language HTTP header

The Accept-Language header is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request isn't an infallible way to detect the user's preferred language (see Setting language preferences in a browser). A production app should include a way for a user to customize their choice of culture.

Set the Accept-Language HTTP header in IE

1. From the gear icon, tap **Internet Options**.
2. Tap **Languages**.



3. Tap **Set Language Preferences**.
4. Tap **Add a language**.
5. Add the language.
6. Tap the language, then tap **Move Up**.

The Content-Language HTTP header

The Content-Language entity header:

- Is used to describe the language(s) intended for the audience.
- Allows a user to differentiate according to the users' own preferred language.

Entity headers are used in both HTTP requests and responses.

The `Content-Language` header can be added by setting the property `ApplyCurrentCultureToResponseHeaders`.

Adding the `Content-Language` header:

- Allows the RequestLocalizationMiddleware to set the `Content-Language` header with the `CurrentUICulture`.
- Eliminates the need to set the response header `Content-Language` explicitly.

```
app.UseRequestLocalization(new RequestLocalizationOptions
{
    ApplyCurrentCultureToResponseHeaders = true
});
```

Use a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

```
private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture, uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.AddInitialRequestCultureProvider(new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
});
```

Use `RequestLocalizationOptions` to add or remove localization providers.

Set the culture programmatically

This sample **Localization.StarterWeb** project on GitHub contains UI to set the `Culture`. The *Views/Shared/_SelectLanguagePartial.cshtml* file allows you to select the culture from the list of supported cultures:

```
@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text = c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~/" : $"~{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"] @requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
          method="post" class="form-horizontal" role="form">
        <label asp-for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]</label>
<select name="culture"
          onchange="this.form.submit();"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-items="cultureItems">
        </select>
    </form>
</div>
```

The *Views/Shared/_SelectLanguagePartial.cshtml* file is added to the `footer` section of the layout file so it will be available to all views:

```
<div class="container body-content" style="margin-top:60px">
    @RenderBody()
    <hr>
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; @System.DateTime.Now.Year - Localization</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

The `SetLanguage` method sets the culture cookie.

```
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

You can't plug in the *_SelectLanguagePartial.cshtml* to sample code for this project. The **Localization.StarterWeb** project on GitHub has code to flow the `RequestLocalizationOptions` to a Razor partial through the Dependency Injection container.

Model binding route data and query strings

See Globalization behavior of model binding route data and query strings.

Globalization and localization terms

The process of localizing your app also requires a basic understanding of relevant character sets commonly used in modern software development and an understanding of the issues associated with them. Although all computers store text as numbers (codes), different systems store the same text using different numbers. The localization process refers to translating the app user interface (UI) for a specific culture/locale.

Localizability is an intermediate process for verifying that a globalized app is ready for localization.

The RFC 4646 format for the culture name is `<languagecode2>-<country/regioncode2>`, where `<languagecode2>` is the language code and `<country/regioncode2>` is the subculture code. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). RFC 4646 is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. See /previous-versions/commerce-server/ee825488(v=cs.20).

Internationalization is often abbreviated to "I18N". The abbreviation takes the first and last letters and the number of letters between them, so 18 stands for the number of letters between the first "I" and the last "N". The same applies to Globalization (G11N), and Localization (L10N).

Terms:

- Globalization (G11N): The process of making an app support different languages and regions.
- Localization (L10N): The process of customizing an app for a given language and region.
- Internationalization (I18N): Describes both globalization and localization.
- Culture: It's a language and, optionally, a region.
- Neutral culture: A culture that has a specified language, but not a region. (for example "en", "es")
- Specific culture: A culture that has a specified language and region. (for example "en-US", "en-GB", "es-CL")
- Parent culture: The neutral culture that contains a specific culture. (for example, "en" is the parent culture of "en-US" and "en-GB")
- Locale: A locale is the same as a culture.

> **NOTE**
>
> You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub issue 4076 for instructions on adding decimal comma.

> **NOTE**
>
> Prior to ASP.NET Core 3.0 web apps write one log of type `LogLevel.Warning` per request if the requested culture is unsupported. Logging one `LogLevel.Warning` per request is can make large log files with redundant information. This behavior has been changed in ASP.NET 3.0. The `RequestLocalizationMiddleware` writes a log of type `LogLevel.Debug`, which reduces the size of production logs.

Additional resources

- Troubleshoot ASP.NET Core Localization
- Localization.StarterWeb project used in the article.
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Configure portable object localization in ASP.NET Core

9/22/2020 • 13 minutes to read • Edit Online

By Sébastien Ros, Scott Addie and Hisham Bin Ateya

This article walks through the steps for using Portable Object (PO) files in an ASP.NET Core application with the Orchard Core framework.

Note: Orchard Core isn't a Microsoft product. Consequently, Microsoft provides no support for this feature.

View or download sample code (how to download)

What is a PO file?

PO files are distributed as text files containing the translated strings for a given language. Some advantages of using PO files instead *.resx* files include:

- PO files support pluralization; *.resx* files don't support pluralization.
- PO files aren't compiled like *.resx* files. As such, specialized tooling and build steps aren't required.
- PO files work well with collaborative online editing tools.

Example

Here is a sample PO file containing the translation for two strings in French, including one with its plural form:

fr.po

```
#: Services/EmailService.cs:29
msgid "Enter a comma separated list of email addresses."
msgstr "Entrez une liste d'emails séparés par une virgule."

#: Views/Email.cshtml:112
msgid "The email address is \"{0}\"."
msgid_plural "The email addresses are \"{0}\"."
msgstr[0] "L'adresse email est \"{0}\"."
msgstr[1] "Les adresses email sont \"{0}\""
```

This example uses the following syntax:

- `#:` : A comment indicating the context of the string to be translated. The same string might be translated differently depending on where it's being used.
- `msgid` : The untranslated string.
- `msgstr` : The translated string.

In the case of pluralization support, more entries can be defined.

- `msgid_plural` : The untranslated plural string.
- `msgstr[0]` : The translated string for the case 0.
- `msgstr[N]` : The translated string for the case N.

The PO file specification can be found here.

Configuring PO file support in ASP.NET Core

This example is based on an ASP.NET Core MVC application generated from a Visual Studio 2017 project template.

Referencing the package

Add a reference to the `OrchardCore.Localization.Core` NuGet package. It's available on MyGet at the following package source: https://www.myget.org/F/orchardcore-preview/api/v3/index.json

The *.csproj* file now contains a line similar to the following (version number may vary):

```
<PackageReference Include="OrchardCore.Localization.Core" Version="1.0.0-rc2-13450" />
```

Registering the service

Add the required services to the `ConfigureServices` method of *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix);

    services.AddPortableObjectLocalization();

    services.Configure<RequestLocalizationOptions>(options =>
        {
            var supportedCultures = new List<CultureInfo>
            {
                new CultureInfo("en-US"),
                new CultureInfo("en"),
                new CultureInfo("fr-FR"),
                new CultureInfo("fr")
            };

            options.DefaultRequestCulture = new RequestCulture("en-US");
            options.SupportedCultures = supportedCultures;
            options.SupportedUICultures = supportedCultures;
        });
}
```

Add the required middleware to the `Configure` method of *Startup.cs*:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseRouting();
    app.UseStaticFiles();

    app.UseRequestLocalization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Add the following code to your Razor view of choice. *About.cshtml* is used in this example.

```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer

<p>@Localizer["Hello world!"]</p>
```

An `IViewLocalizer` instance is injected and used to translate the text "Hello world!".

Creating a PO file

Create a file named *<culture code>.po* in your application root folder. In this example, the file name is *fr.po* because the French language is used:

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

This file stores both the string to translate and the French-translated string. Translations revert to their parent culture, if necessary. In this example, the *fr.po* file is used if the requested culture is `fr-FR` or `fr-CA`.

Testing the application

Run your application, and navigate to the URL `/Home/About`. The text **Hello world!** is displayed.

Navigate to the URL `/Home/About?culture=fr-FR`. The text **Bonjour le monde!** is displayed.

Pluralization

PO files support pluralization forms, which is useful when the same string needs to be translated differently based on a cardinality. This task is made complicated by the fact that each language defines custom rules to select which string to use based on the cardinality.

The Orchard Localization package provides an API to invoke these different plural forms automatically.

Creating pluralization PO files

Add the following content to the previously mentioned *fr.po* file:

```
msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

See What is a PO file? for an explanation of what each entry in this example represents.

Adding a language using different pluralization forms

English and French strings were used in the previous example. English and French have only two pluralization forms and share the same form rules, which is that a cardinality of one is mapped to the first plural form. Any other cardinality is mapped to the second plural form.

Not all languages share the same rules. This is illustrated with the Czech language, which has three plural forms.

Create the `cs.po` file as follows, and note how the pluralization needs three different translations:

```
msgid "Hello world!"
msgstr "Ahoj světe!!"

msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Existuje jedna položka."
msgstr[1] "Existují {0} položky."
msgstr[2] "Existuje {0} položek."
```

To accept Czech localizations, add `"cs"` to the list of supported cultures in the `ConfigureServices` method:

```
var supportedCultures = new List<CultureInfo>
{
    new CultureInfo("en-US"),
    new CultureInfo("en"),
    new CultureInfo("fr-FR"),
    new CultureInfo("fr"),
    new CultureInfo("cs")
};
```

Edit the *Views/Home/About.cshtml* file to render localized, plural strings for several cardinalities:

```
<p>@Localizer.Plural(1, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(2, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(5, "There is one item.", "There are {0} items.")</p>
```

Note: In a real world scenario, a variable would be used to represent the count. Here, we repeat the same code with three different values to expose a very specific case.

Upon switching cultures, you see the following:

For `/Home/About`:

```
There is one item.
There are 2 items.
There are 5 items.
```

For `/Home/About?culture=fr`:

```
Il y a un élément.
Il y a 2 éléments.
Il y a 5 éléments.
```

For `/Home/About?culture=cs`:

```
Existuje jedna položka.
Existují 2 položky.
Existuje 5 položek.
```

Note that for the Czech culture, the three translations are different. The French and English cultures share the same construction for the two last translated strings.

Advanced tasks

Contextualizing strings

Applications often contain the strings to be translated in several places. The same string may have a different translation in certain locations within an app (Razor views or class files). A PO file supports the notion of a file context, which can be used to categorize the string being represented. Using a file context, a string can be translated differently, depending on the file context (or lack of a file context).

The PO localization services use the name of the full class or the view that's used when translating a string. This is accomplished by setting the value on the `msgctxt` entry.

Consider a minor addition to the previous *fr.po* example. A Razor view located at *Views/Home/About.cshtml* can be defined as the file context by setting the reserved `msgctxt` entry's value:

```
msgctxt "Views.Home.About"
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

With the `msgctxt` set as such, text translation occurs when navigating to `/Home/About?culture=fr-FR`. The translation won't occur when navigating to `/Home/Contact?culture=fr-FR`.

When no specific entry is matched with a given file context, Orchard Core's fallback mechanism looks for an appropriate PO file without a context. Assuming there's no specific file context defined for *Views/Home/Contact.cshtml*, navigating to `/Home/Contact?culture=fr-FR` loads a PO file such as:

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

Changing the location of PO files

The default location of PO files can be changed in `ConfigureServices`:

```
services.AddPortableObjectLocalization(options => options.ResourcesPath = "Localization");
```

In this example, the PO files are loaded from the *Localization* folder.

Implementing a custom logic for finding localization files

When more complex logic is needed to locate PO files, the `OrchardCore.Localization.PortableObject.ILocalizationFileLocationProvider` interface can be implemented and registered as a service. This is useful when PO files can be stored in varying locations or when the files have to be

found within a hierarchy of folders.

Using a different default pluralized language

The package includes a `Plural` extension method that's specific to two plural forms. For languages requiring more plural forms, create an extension method. With an extension method, you won't need to provide any localization file for the default language — the original strings are already available directly in the code.

You can use the more generic `Plural(int count, string[] pluralForms, params object[] arguments)` overload which accepts a string array of translations.

By Sébastien Ros and Scott Addie

This article walks through the steps for using Portable Object (PO) files in an ASP.NET Core application with the Orchard Core framework.

Note: Orchard Core isn't a Microsoft product. Consequently, Microsoft provides no support for this feature.

View or download sample code (how to download)

What is a PO file?

PO files are distributed as text files containing the translated strings for a given language. Some advantages of using PO files instead *.resx* files include:

- PO files support pluralization; *.resx* files don't support pluralization.
- PO files aren't compiled like *.resx* files. As such, specialized tooling and build steps aren't required.
- PO files work well with collaborative online editing tools.

Example

Here is a sample PO file containing the translation for two strings in French, including one with its plural form:

fr.po

```
#: Services/EmailService.cs:29
msgid "Enter a comma separated list of email addresses."
msgstr "Entrez une liste d'emails séparés par une virgule."

#: Views/Email.cshtml:112
msgid "The email address is \"{0}\"."
msgid_plural "The email addresses are \"{0}\"."
msgstr[0] "L'adresse email est \"{0}\"."
msgstr[1] "Les adresses email sont \"{0}\""
```

This example uses the following syntax:

- `#:` : A comment indicating the context of the string to be translated. The same string might be translated differently depending on where it's being used.
- `msgid` : The untranslated string.
- `msgstr` : The translated string.

In the case of pluralization support, more entries can be defined.

- `msgid_plural` : The untranslated plural string.
- `msgstr[0]` : The translated string for the case 0.
- `msgstr[N]` : The translated string for the case N.

The PO file specification can be found here.

Configuring PO file support in ASP.NET Core

This example is based on an ASP.NET Core MVC application generated from a Visual Studio 2017 project template.

Referencing the package

Add a reference to the `OrchardCore.Localization.Core` NuGet package. It's available on MyGet at the following package source: https://www.myget.org/F/orchardcore-preview/api/v3/index.json

The *.csproj* file now contains a line similar to the following (version number may vary):

```
<PackageReference Include="OrchardCore.Localization.Core" Version="1.0.0-beta1-3187" />
```

Registering the service

Add the required services to the `ConfigureServices` method of *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix);

    services.AddPortableObjectLocalization();

    services.Configure<RequestLocalizationOptions>(options =>
        {
            var supportedCultures = new List<CultureInfo>
            {
                new CultureInfo("en-US"),
                new CultureInfo("en"),
                new CultureInfo("fr-FR"),
                new CultureInfo("fr")
            };

            options.DefaultRequestCulture = new RequestCulture("en-US");
            options.SupportedCultures = supportedCultures;
            options.SupportedUICultures = supportedCultures;
        });
}
```

Add the required middleware to the `Configure` method of *Startup.cs*:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseRequestLocalization();

    app.UseMvcWithDefaultRoute();
}
```

Add the following code to your Razor view of choice. *About.cshtml* is used in this example.

```
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer

<p>@Localizer["Hello world!"]</p>
```

An `IViewLocalizer` instance is injected and used to translate the text "Hello world!".

Creating a PO file

Create a file named *<culture code>.po* in your application root folder. In this example, the file name is *fr.po* because the French language is used:

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

This file stores both the string to translate and the French-translated string. Translations revert to their parent culture, if necessary. In this example, the *fr.po* file is used if the requested culture is `fr-FR` or `fr-CA`.

Testing the application

Run your application, and navigate to the URL `/Home/About`. The text **Hello world!** is displayed.

Navigate to the URL `/Home/About?culture=fr-FR`. The text **Bonjour le monde!** is displayed.

Pluralization

PO files support pluralization forms, which is useful when the same string needs to be translated differently based on a cardinality. This task is made complicated by the fact that each language defines custom rules to select which string to use based on the cardinality.

The Orchard Localization package provides an API to invoke these different plural forms automatically.

Creating pluralization PO files

Add the following content to the previously mentioned *fr.po* file:

```
msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

See What is a PO file? for an explanation of what each entry in this example represents.

Adding a language using different pluralization forms

English and French strings were used in the previous example. English and French have only two pluralization forms and share the same form rules, which is that a cardinality of one is mapped to the first plural form. Any other cardinality is mapped to the second plural form.

Not all languages share the same rules. This is illustrated with the Czech language, which has three plural forms.

Create the `cs.po` file as follows, and note how the pluralization needs three different translations:

```
msgid "Hello world!"
msgstr "Ahoj světe!!"

msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Existuje jedna položka."
msgstr[1] "Existují {0} položky."
msgstr[2] "Existuje {0} položek."
```

To accept Czech localizations, add `"cs"` to the list of supported cultures in the `ConfigureServices` method:

```
var supportedCultures = new List<CultureInfo>
{
    new CultureInfo("en-US"),
    new CultureInfo("en"),
    new CultureInfo("fr-FR"),
    new CultureInfo("fr"),
    new CultureInfo("cs")
};
```

Edit the *Views/Home/About.cshtml* file to render localized, plural strings for several cardinalities:

```
<p>@Localizer.Plural(1, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(2, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(5, "There is one item.", "There are {0} items.")</p>
```

Note: In a real world scenario, a variable would be used to represent the count. Here, we repeat the same code with three different values to expose a very specific case.

Upon switching cultures, you see the following:

For `/Home/About`:

```
There is one item.
There are 2 items.
There are 5 items.
```

For `/Home/About?culture=fr`:

```
Il y a un élément.
Il y a 2 éléments.
Il y a 5 éléments.
```

For `/Home/About?culture=cs`:

```
Existuje jedna položka.
Existují 2 položky.
Existuje 5 položek.
```

Note that for the Czech culture, the three translations are different. The French and English cultures share the same construction for the two last translated strings.

Advanced tasks

Contextualizing strings

Applications often contain the strings to be translated in several places. The same string may have a different translation in certain locations within an app (Razor views or class files). A PO file supports the notion of a file context, which can be used to categorize the string being represented. Using a file context, a string can be translated differently, depending on the file context (or lack of a file context).

The PO localization services use the name of the full class or the view that's used when translating a string. This is accomplished by setting the value on the `msgctxt` entry.

Consider a minor addition to the previous *fr.po* example. A Razor view located at *Views/Home/About.cshtml* can be defined as the file context by setting the reserved `msgctxt` entry's value:

```
msgctxt "Views.Home.About"
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

With the `msgctxt` set as such, text translation occurs when navigating to `/Home/About?culture=fr-FR`. The translation won't occur when navigating to `/Home/Contact?culture=fr-FR`.

When no specific entry is matched with a given file context, Orchard Core's fallback mechanism looks for an appropriate PO file without a context. Assuming there's no specific file context defined for *Views/Home/Contact.cshtml*, navigating to `/Home/Contact?culture=fr-FR` loads a PO file such as:

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

Changing the location of PO files

The default location of PO files can be changed in `ConfigureServices`:

```
services.AddPortableObjectLocalization(options => options.ResourcesPath = "Localization");
```

In this example, the PO files are loaded from the *Localization* folder.

Implementing a custom logic for finding localization files

When more complex logic is needed to locate PO files, the `OrchardCore.Localization.PortableObject.ILocalizationFileLocationProvider` interface can be implemented and registered as a service. This is useful when PO files can be stored in varying locations or when the files have to be found within a hierarchy of folders.

Using a different default pluralized language

The package includes a `Plural` extension method that's specific to two plural forms. For languages requiring more plural forms, create an extension method. With an extension method, you won't need to provide any localization file for the default language — the original strings are already available directly in the code.

You can use the more generic `Plural(int count, string[] pluralForms, params object[] arguments)` overload which accepts a string array of translations.

Localization Extensibility

9/22/2020 • 2 minutes to read • Edit Online

By Hisham Bin Ateya

This article:

- Lists the extensibility points on the localization APIs.
- Provides instructions on how to extend ASP.NET Core app localization.

Extensible Points in Localization APIs

ASP.NET Core localization APIs are built to be extensible. Extensibility allows developers to customize the localization according to their needs. For instance, OrchardCore has a `POStringLocalizer`. `POStringLocalizer` describes in detail using Portable Object localization to use `PO` files to store localization resources.

This article lists the two main extensibility points that localization APIs provide:

- RequestCultureProvider
- IStringLocalizer

Localization Culture Providers

ASP.NET Core localization APIs have four default providers that can determine the current culture of an executing request:

- QueryStringRequestCultureProvider
- CookieRequestCultureProvider
- AcceptLanguageHeaderRequestCultureProvider
- CustomRequestCultureProvider

The preceding providers are described in detail in the Localization Middleware documentation. If the default providers don't meet your needs, build a custom provider using one of the following approaches:

Use CustomRequestCultureProvider

CustomRequestCultureProvider provides a custom RequestCultureProvider that uses a simple delegate to determine the current localization culture:

```
options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
{
    var currentCulture = "en";
    var segments = context.Request.Path.Value.Split(new char[] { '/' },
        StringSplitOptions.RemoveEmptyEntries);

    if (segments.Length > 1 && segments[0].Length == 2)
    {
        currentCulture = segments[0];
    }

    var requestCulture = new ProviderCultureResult(currentCulture);

    return Task.FromResult(requestCulture);
}));
```

```
options.AddInitialRequestCultureProvider(new CustomRequestCultureProvider(async context =>
{
    var currentCulture = "en";
    var segments = context.Request.Path.Value.Split(new char[] { '/' },
        StringSplitOptions.RemoveEmptyEntries);

    if (segments.Length > 1 && segments[0].Length == 2)
    {
        currentCulture = segments[0];
    }

    var requestCulture = new ProviderCultureResult(currentCulture);

    return Task.FromResult(requestCulture);
}));
```

Use a new implemetation of RequestCultureProvider

A new implementation of RequestCultureProvider can be created that determines the request culture information from a custom source. For example, the custom source can be a configuration file or database.

The following example shows `AppSettingsRequestCultureProvider`, which extends the RequestCultureProvider to determine the request culture information from *appsettings.json*:

```
public class AppSettingsRequestCultureProvider : RequestCultureProvider
{
    public string CultureKey { get; set; } = "culture";

    public string UICultureKey { get; set; } = "ui-culture";

    public override Task<ProviderCultureResult> DetermineProviderCultureResult(HttpContext httpContext)
    {
        if (httpContext == null)
        {
            throw new ArgumentNullException();
        }

        var configuration = httpContext.RequestServices.GetService<IConfigurationRoot>();
        var culture = configuration[CultureKey];
        var uiCulture = configuration[UICultureKey];

        if (culture == null && uiCulture == null)
        {
            return Task.FromResult((ProviderCultureResult)null);
        }

        if (culture != null && uiCulture == null)
        {
            uiCulture = culture;
        }

        if (culture == null && uiCulture != null)
        {
            culture = uiCulture;
        }

        var providerResultCulture = new ProviderCultureResult(culture, uiCulture);

        return Task.FromResult(providerResultCulture);
    }
}
```

Localization resources

ASP.NET Core localization provides ResourceManagerStringLocalizer. ResourceManagerStringLocalizer is an implementation of IStringLocalizer that is uses `resx` to store localization resources.

You aren't limited to using `resx` files. By implementing `IStringLocalized`, any data source can be used.

The following example projects implement IStringLocalizer:

- EFStringLocalizer
- JsonStringLocalizer
- SqlLocalizer

Troubleshoot ASP.NET Core Localization

9/22/2020 • 2 minutes to read • Edit Online

By Hisham Bin Ateya

This article provides instructions on how to diagnose ASP.NET Core app localization issues.

Localization configuration issues

Localization middleware order

The app may not localize because the localization middleware isn't ordered as expected.

To resolve this issue, ensure that localization middleware is registered before MVC middleware. Otherwise, the localization middleware isn't applied.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddLocalization(options => options.ResourcesPath = "Resources");

    services.AddMvc();
}
```

Localization resources path not found

Supported Cultures in RequestCultureProvider don't match with registered once

Resource file naming issues

ASP.NET Core has predefined rules and guidelines for localization resources file naming, which are described in detail here.

Missing resources

Common causes of resources not being found include:

- Resource names are misspelled in either the `resx` file or the localizer request.
- The resource is missing from the `resx` for some languages, but exists in others.
- If you're still having trouble, check the localization log messages (which are at `Debug` log level) for more details about the missing resources.

Hint: *When using* `CookieRequestCultureProvider`*, verify single quotes are not used with the cultures inside the localization cookie value. For example,* `c='en-UK'|uic='en-US'` *is an invalid cookie value, while* `c=en-UK|uic=en-US` *is a valid.*

Resources & Class Libraries issues

ASP.NET Core by default provides a way to allow the class libraries to find their resource files via ResourceLocationAttribute.

Common issues with class libraries include:

- Missing the `ResourceLocationAttribute` in a class library will prevent `ResourceManagerStringLocalizerFactory` from discovering the resources.

- Resource file naming. For more information, see Resource file naming issues section.
- Changing the root namespace of the class library. For more information, see Root Namespace issues section.

CustomRequestCultureProvider doesn't work as expected

The `RequestLocalizationOptions` class has three default providers:

1. `QueryStringRequestCultureProvider`
2. `CookieRequestCultureProvider`
3. `AcceptLanguageHeaderRequestCultureProvider`

The CustomRequestCultureProvider allows you to customize how the localization culture is provided in your app. The `CustomRequestCultureProvider` is used when the default providers don't meet your requirements.

- A common reason custom provider don't work properly is that it isn't the first provider in the `RequestCultureProviders` list. To resolve this issue:
- Insert the custom provider at the position 0 in the `RequestCultureProviders` list as the following:

```
options.RequestCultureProviders.Insert(0, new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
```

```
options.AddInitialRequestCultureProvider(new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
```

- Use `AddInitialRequestCultureProvider` extension method to set the custom provider as initial provider.

Root Namespace issues

When the root namespace of an assembly is different than the assembly name, localization doesn't work by default. To avoid this issue use RootNamespace, which is described in detail here

> **WARNING**
> This can occur when a project's name is not a valid .NET identifier. For instance `my-project-name.csproj` will use the root namespace `my_project_name` and the assembly name `my-project-name` leading to this error.

Resources & Build Action

If you use resource files for localization, it's important that they have an appropriate build action. They should be **Embedded Resource**, otherwise the `ResourceStringLocalizer` is not able to find these resources.

Model Binding in ASP.NET Core

9/22/2020 • 31 minutes to read • Edit Online

This article explains what model binding is, how it works, and how to customize its behavior.

View or download sample code (how to download).

What is Model binding

Controllers and Razor pages work with data that comes from HTTP requests. For example, route data may provide a record key, and posted form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone. Model binding automates this process. The model binding system:

- Retrieves data from various sources such as route data, form fields, and query strings.
- Provides the data to controllers and Razor pages in method parameters and public properties.
- Converts string data to .NET types.
- Updates properties of complex types.

Example

Suppose you have the following action method:

```
[HttpGet("{id}")]
public ActionResult<Pet> GetById(int id, bool dogsOnly)
```

And the app receives a request with this URL:

```
http://contoso.com/api/pets/2?DogsOnly=true
```

Model binding goes through the following steps after the routing system selects the action method:

- Finds the first parameter of `GetByID`, an integer named `id`.
- Looks through the available sources in the HTTP request and finds `id` = "2" in route data.
- Converts the string "2" into integer 2.
- Finds the next parameter of `GetByID`, a boolean named `dogsOnly`.
- Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
- Converts the string "true" into boolean `true`.

The framework then calls the `GetById` method, passing in 2 for the `id` parameter, and `true` for the `dogsOnly` parameter.

In the preceding example, the model binding targets are method parameters that are simple types. Targets may also be the properties of a complex type. After each property is successfully bound, model validation occurs for that property. The record of what data is bound to the

model, and any binding or validation errors, is stored in ControllerBase.ModelState or PageModel.ModelState. To find out if this process was successful, the app checks the ModelState.IsValid flag.

Targets

Model binding tries to find values for the following kinds of targets:

- Parameters of the controller action method that a request is routed to.
- Parameters of the Razor Pages handler method that a request is routed to.
- Public properties of a controller or `PageModel` class, if specified by attributes.

[BindProperty] attribute

Can be applied to a public property of a controller or `PageModel` class to cause model binding to target that property:

```
public class EditModel : InstructorsPageModel
{
    [BindProperty]
    public Instructor Instructor { get; set; }
```

[BindProperties] attribute

Available in ASP.NET Core 2.1 and later. Can be applied to a controller or `PageModel` class to tell model binding to target all public properties of the class:

```
[BindProperties(SupportsGet = true)]
public class CreateModel : InstructorsPageModel
{
    public Instructor Instructor { get; set; }
```

Model binding for HTTP GET requests

By default, properties are not bound for HTTP GET requests. Typically, all you need for a GET request is a record ID parameter. The record ID is used to look up the item in the database. Therefore, there is no need to bind a property that holds an instance of the model. In scenarios where you do want properties bound to data from GET requests, set the `SupportsGet` property to `true`:

```
[BindProperty(Name = "ai_user", SupportsGet = true)]
public string ApplicationInsightsCookie { get; set; }
```

Sources

By default, model binding gets data in the form of key-value pairs from the following sources in an HTTP request:

1. Form fields
2. The request body (For controllers that have the [ApiController] attribute.)
3. Route data
4. Query string parameters
5. Uploaded files

For each target parameter or property, the sources are scanned in the order indicated in the

preceding list. There are a few exceptions:

- Route data and query string values are used only for simple types.
- Uploaded files are bound only to target types that implement `IFormFile` or `IEnumerable<IFormFile>`.

If the default source is not correct, use one of the following attributes to specify the source:

- `[FromQuery]` - Gets values from the query string.
- `[FromRoute]` - Gets values from route data.
- `[FromForm]` - Gets values from posted form fields.
- `[FromBody]` - Gets values from the request body.
- `[FromHeader]` - Gets values from HTTP headers.

These attributes:

- Are added to model properties individually (not to the model class), as in the following example:

```
public class Instructor
{
    public int ID { get; set; }

    [FromQuery(Name = "Note")]
    public string NoteFromQueryString { get; set; }
```

- Optionally accept a model name value in the constructor. This option is provided in case the property name doesn't match the value in the request. For instance, the value in the request might be a header with a hyphen in its name, as in the following example:

```
public void OnGet([FromHeader(Name = "Accept-Language")] string language)
```

[FromBody] attribute

Apply the `[FromBody]` attribute to a parameter to populate its properties from the body of an HTTP request. The ASP.NET Core runtime delegates the responsibility of reading the body to an input formatter. Input formatters are explained later in this article.

When `[FromBody]` is applied to a complex type parameter, any binding source attributes applied to its properties are ignored. For example, the following `Create` action specifies that its `pet` parameter is populated from the body:

```
public ActionResult<Pet> Create([FromBody] Pet pet)
```

The `Pet` class specifies that its `Breed` property is populated from a query string parameter:

```
public class Pet
{
    public string Name { get; set; }

    [FromQuery] // Attribute is ignored.
    public string Breed { get; set; }
}
```

In the preceding example:

- The `[FromQuery]` attribute is ignored.
- The `Breed` property is not populated from a query string parameter.

Input formatters read only the body and don't understand binding source attributes. If a suitable value is found in the body, that value is used to populate the `Breed` property.

Don't apply `[FromBody]` to more than one parameter per action method. Once the request stream is read by an input formatter, it's no longer available to be read again for binding other `[FromBody]` parameters.

Additional sources

Source data is provided to the model binding system by *value providers*. You can write and register custom value providers that get data for model binding from other sources. For example, you might want data from cookies or session state. To get data from a new source:

- Create a class that implements `IValueProvider`.
- Create a class that implements `IValueProviderFactory`.
- Register the factory class in `Startup.ConfigureServices`.

The sample app includes a value provider and factory example that gets values from cookies. Here's the registration code in `Startup.ConfigureServices`:

```
services.AddRazorPages()
    .AddMvcOptions(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters();
```

The code shown puts the custom value provider after all the built-in value providers. To make it the first in the list, call `Insert(0, new CookieValueProviderFactory())` instead of `Add`.

No source for a model property

By default, a model state error isn't created if no value is found for a model property. The property is set to null or a default value:

- Nullable simple types are set to `null`.
- Non-nullable value types are set to `default(T)`. For example, a parameter `int id` is set to 0.
- For complex Types, model binding creates an instance by using the default constructor, without setting properties.
- Arrays are set to `Array.Empty<T>()`, except that `byte[]` arrays are set to `null`.

If model state should be invalidated when nothing is found in form fields for a model property, use the `[BindRequired]` attribute.

Note that this `[BindRequired]` behavior applies to model binding from posted form data, not to JSON or XML data in a request body. Request body data is handled by input formatters.

Type conversion errors

If a source is found but can't be converted into the target type, model state is flagged as invalid. The target parameter or property is set to null or a default value, as noted in the previous section.

In an API controller that has the `[ApiController]` attribute, invalid model state results in an automatic HTTP 400 response.

In a Razor page, redisplay the page with an error message:

```
public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _instructorsInMemoryStore.Add(Instructor);
    return RedirectToPage("./Index");
}
```

Client-side validation catches most bad data that would otherwise be submitted to a Razor Pages form. This validation makes it hard to trigger the preceding highlighted code. The sample app includes a **Submit with Invalid Date** button that puts bad data in the **Hire Date** field and submits the form. This button shows how the code for redisplaying the page works when data conversion errors occur.

When the page is redisplayed by the preceding code, the invalid input is not shown in the form field. This is because the model property has been set to null or a default value. The invalid input does appear in an error message. But if you want to redisplay the bad data in the form field, consider making the model property a string and doing the data conversion manually.

The same strategy is recommended if you don't want type conversion errors to result in model state errors. In that case, make the model property a string.

Simple types

The simple types that the model binder can convert source strings into include the following:

- Boolean
- Byte, SByte
- Char
- DateTime
- DateTimeOffset
- Decimal
- Double
- Enum
- Guid
- Int16, Int32, Int64
- Single
- TimeSpan
- UInt16, UInt32, UInt64
- Uri
- Version

Complex types

A complex type must have a public default constructor and public writable properties to bind. When model binding occurs, the class is instantiated using the public default constructor.

For each property of the complex type, model binding looks through the sources for the name pattern *prefix.property_name*. If nothing is found, it looks for just *property_name* without the prefix.

For binding to a parameter, the prefix is the parameter name. For binding to a `PageModel` public property, the prefix is the public property name. Some attributes have a `Prefix` property that lets you override the default usage of parameter or property name.

For example, suppose the complex type is the following `Instructor` class:

```
public class Instructor
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Prefix = parameter name

If the model to be bound is a parameter named `instructorToUpdate`:

```
public IActionResult OnPost(int? id, Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `instructorToUpdate.ID`. If that isn't found, it looks for `ID` without a prefix.

Prefix = property name

If the model to be bound is a property named `Instructor` of the controller or `PageModel` class:

```
[BindProperty]
public Instructor Instructor { get; set; }
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Custom prefix

If the model to be bound is a parameter named `instructorToUpdate` and a `Bind` attribute specifies `Instructor` as the prefix:

```
public IActionResult OnPost(
    int? id, [Bind(Prefix = "Instructor")] Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Attributes for complex type targets

Several built-in attributes are available for controlling model binding of complex types:

- `[Bind]`
- `[BindRequired]`

- `[BindNever]`

> **WARNING**
> These attributes affect model binding when posted form data is the source of values. They do ***not*** affect input formatters, which process posted JSON and XML request bodies. Input formatters are explained later in this article.

[Bind] attribute

Can be applied to a class or a method parameter. Specifies which properties of a model should be included in model binding. `[Bind]` does ***not*** affect input formatters.

In the following example, only the specified properties of the `Instructor` model are bound when any handler or action method is called:

```
[Bind("LastName,FirstMidName,HireDate")]
public class Instructor
```

In the following example, only the specified properties of the `Instructor` model are bound when the `OnPost` method is called:

```
[HttpPost]
public IActionResult OnPost([Bind("LastName,FirstMidName,HireDate")] Instructor instructor)
```

The `[Bind]` attribute can be used to protect against overposting in *create* scenarios. It doesn't work well in edit scenarios because excluded properties are set to null or a default value instead of being left unchanged. For defense against overposting, view models are recommended rather than the `[Bind]` attribute. For more information, see Security note about overposting.

[BindRequired] attribute

Can only be applied to model properties, not to method parameters. Causes model binding to add a model state error if binding cannot occur for a model's property. Here's an example:

```
public class InstructorWithCollection
{
    public int ID { get; set; }

    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    [Display(Name = "Hire Date")]
    [BindRequired]
    public DateTime HireDate { get; set; }
```

See also the discussion of the `[Required]` attribute in Model validation.

[BindNever] attribute

Can only be applied to model properties, not to method parameters. Prevents model binding from setting a model's property. Here's an example:

```
public class InstructorWithDictionary
{
    [BindNever]
    public int ID { get; set; }
```

Collections

For targets that are collections of simple types, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the parameter to be bound is an array named `selectedCourses`:

  ```
  public IActionResult OnPost(int? id, int[] selectedCourses)
  ```

- Form or query string data can be in one of the following formats:

  ```
  selectedCourses=1050&selectedCourses=2000
  ```

  ```
  selectedCourses[0]=1050&selectedCourses[1]=2000
  ```

  ```
  [0]=1050&[1]=2000
  ```

  ```
  selectedCourses[a]=1050&selectedCourses[b]=2000&selectedCourses.index=a&selectedCour
  ses.index=b
  ```

  ```
  [a]=1050&[b]=2000&index=a&index=b
  ```

- The following format is supported only in form data:

  ```
  selectedCourses[]=1050&selectedCourses[]=2000
  ```

- For all of the preceding example formats, model binding passes an array of two items to the `selectedCourses` parameter:

  - selectedCourses[0]=1050
  - selectedCourses[1]=2000

  Data formats that use subscript numbers (... [0] ... [1] ...) must ensure that they are numbered sequentially starting at zero. If there are any gaps in subscript numbering, all items after the gap are ignored. For example, if the subscripts are 0 and 2 instead of 0 and 1, the second item is ignored.

Dictionaries

For `Dictionary` targets, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the target parameter is a `Dictionary<int, string>` named `selectedCourses`:

```
public IActionResult OnPost(int? id, Dictionary<int, string> selectedCourses)
```

- The posted form or query string data can look like one of the following examples:

```
selectedCourses[1050]=Chemistry&selectedCourses[2000]=Economics
```

```
[1050]=Chemistry&selectedCourses[2000]=Economics
```

```
selectedCourses[0].Key=1050&selectedCourses[0].Value=Chemistry&
selectedCourses[1].Key=2000&selectedCourses[1].Value=Economics
```

```
[0].Key=1050&[0].Value=Chemistry&[1].Key=2000&[1].Value=Economics
```

- For all of the preceding example formats, model binding passes a dictionary of two items to the `selectedCourses` parameter:
  - selectedCourses["1050"]="Chemistry"
  - selectedCourses["2000"]="Economics"

Globalization behavior of model binding route data and query strings

The ASP.NET Core route value provider and query string value provider:

- Treat values as invariant culture.
- Expect that URLs are culture-invariant.

In contrast, values coming from form data undergo a culture-sensitive conversion. This is by design so that URLs are shareable across locales.

To make the ASP.NET Core route value provider and query string value provider undergo a culture-sensitive conversion:

- Inherit from IValueProviderFactory
- Copy the code from QueryStringValueProviderFactory or RouteValueValueProviderFactory
- Replace the culture value passed to the value provider constructor with CultureInfo.CurrentCulture
- Replace the default value provider factory in MVC options with your new one:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
    {
        var index = options.ValueProviderFactories.IndexOf(
            options.ValueProviderFactories.OfType<QueryStringValueProviderFactory>
().Single());
        options.ValueProviderFactories[index] = new
CulturedQueryStringValueProviderFactory();
    });
}
```

```
public class CulturedQueryStringValueProviderFactory : IValueProviderFactory
{
    public Task CreateValueProviderAsync(ValueProviderFactoryContext context)
    {
        if (context == null)
        {
            throw new ArgumentNullException(nameof(context));
        }

        var query = context.ActionContext.HttpContext.Request.Query;
        if (query != null && query.Count > 0)
        {
            var valueProvider = new QueryStringValueProvider(
                BindingSource.Query,
                query,
                CultureInfo.CurrentCulture);

            context.ValueProviders.Add(valueProvider);
        }

        return Task.CompletedTask;
    }
}
```

Special data types

There are some special data types that model binding can handle.

IFormFile and IFormFileCollection

An uploaded file included in the HTTP request. Also supported is `IEnumerable<IFormFile>` for multiple files.

CancellationToken

Used to cancel activity in asynchronous controllers.

FormCollection

Used to retrieve all the values from posted form data.

Input formatters

Data in the request body can be in JSON, XML, or some other format. To parse this data, model binding uses an *input formatter* that is configured to handle a particular content type. By default, ASP.NET Core includes JSON based input formatters for handling JSON data. You can add other formatters for other content types.

ASP.NET Core selects input formatters based on the Consumes attribute. If no attribute is present, it uses the Content-Type header.

To use the built-in XML input formatters:

- Install the `Microsoft.AspNetCore.Mvc.Formatters.Xml` NuGet package.
- In `Startup.ConfigureServices`, call AddXmlSerializerFormatters or AddXmlDataContractSerializerFormatters.

```
services.AddRazorPages()
    .AddMvcOptions(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters();
```

- Apply the `Consumes` attribute to controller classes or action methods that should expect XML in the request body.

```
[HttpPost]
[Consumes("application/xml")]
public ActionResult<Pet> Create(Pet pet)
```

For more information, see Introducing XML Serialization.

Customize model binding with input formatters

An input formatter takes full responsibility for reading data from the request body. To customize this process, configure the APIs used by the input formatter. This section describes how to customize the `System.Text.Json`-based input formatter to understand a custom type named `ObjectId`.

Consider the following model, which contains a custom `ObjectId` property named `Id`:

```
public class ModelWithObjectId
{
    public ObjectId Id { get; set; }
}
```

To customize the model binding process when using `System.Text.Json`, create a class derived from JsonConverter<T>:

```
internal class ObjectIdConverter : JsonConverter<ObjectId>
{
    public override ObjectId Read(
        ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        return new ObjectId(JsonSerializer.Deserialize<int>(ref reader, options));
    }

    public override void Write(
        Utf8JsonWriter writer, ObjectId value, JsonSerializerOptions options)
    {
        writer.WriteNumberValue(value.Id);
    }
}
```

To use a custom converter, apply the JsonConverterAttribute attribute to the type. In the following example, the `ObjectId` type is configured with `ObjectIdConverter` as its custom converter:

```
[JsonConverter(typeof(ObjectIdConverter))]
public struct ObjectId
{
    public ObjectId(int id) =>
        Id = id;

    public int Id { get; }
}
```

For more information, see How to write custom converters.

Exclude specified types from model binding

The model binding and validation systems' behavior is driven by ModelMetadata. You can customize `ModelMetadata` by adding a details provider to MvcOptions.ModelMetadataDetailsProviders. Built-in details providers are available for disabling model binding or validation for specified types.

To disable model binding on all models of a specified type, add an ExcludeBindingMetadataProvider in `Startup.ConfigureServices`. For example, to disable model binding on all models of type `System.Version`:

```
services.AddRazorPages()
    .AddMvcOptions(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters();
```

To disable validation on properties of a specified type, add a SuppressChildValidationMetadataProvider in `Startup.ConfigureServices`. For example, to disable validation on properties of type `System.Guid`:

```
services.AddRazorPages()
    .AddMvcOptions(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters();
```

Custom model binders

You can extend model binding by writing a custom model binder and using the `[ModelBinder]` attribute to select it for a given target. Learn more about custom model binding.

Manual model binding

Model binding can be invoked manually by using the TryUpdateModelAsync method. The method is defined on both `ControllerBase` and `PageModel` classes. Method overloads let you specify the prefix and value provider to use. The method returns `false` if model binding fails. Here's an example:

```
if (await TryUpdateModelAsync<InstructorWithCollection>(
    newInstructor,
    "Instructor",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate))
{
    _instructorsInMemoryStore.Add(newInstructor);
    return RedirectToPage("./Index");
}
PopulateAssignedCourseData(newInstructor);
return Page();
```

TryUpdateModelAsync uses value providers to get data from the form body, query string, and route data. `TryUpdateModelAsync` is typically:

- Used with Razor Pages and MVC apps using controllers and views to prevent over-posting.
- Not used with a web API unless consumed from form data, query strings, and route data. Web API endpoints that consume JSON use Input formatters to deserialize the request body into an object.

For more information, see TryUpdateModelAsync.

[FromServices] attribute

This attribute's name follows the pattern of model binding attributes that specify a data source. But it's not about binding data from a value provider. It gets an instance of a type from the dependency injection container. Its purpose is to provide an alternative to constructor injection for when you need a service only if a particular method is called.

Additional resources

- Model validation in ASP.NET Core MVC
- Custom Model Binding in ASP.NET Core

This article explains what model binding is, how it works, and how to customize its behavior.

View or download sample code (how to download).

What is Model binding

Controllers and Razor pages work with data that comes from HTTP requests. For example, route data may provide a record key, and posted form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone. Model binding automates this process. The model binding system:

- Retrieves data from various sources such as route data, form fields, and query strings.
- Provides the data to controllers and Razor pages in method parameters and public properties.
- Converts string data to .NET types.

- Updates properties of complex types.

Example

Suppose you have the following action method:

```
[HttpGet("{id}")]
public ActionResult<Pet> GetById(int id, bool dogsOnly)
```

And the app receives a request with this URL:

```
http://contoso.com/api/pets/2?DogsOnly=true
```

Model binding goes through the following steps after the routing system selects the action method:

- Finds the first parameter of `GetByID`, an integer named `id`.
- Looks through the available sources in the HTTP request and finds `id` = "2" in route data.
- Converts the string "2" into integer 2.
- Finds the next parameter of `GetByID`, a boolean named `dogsOnly`.
- Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
- Converts the string "true" into boolean `true`.

The framework then calls the `GetById` method, passing in 2 for the `id` parameter, and `true` for the `dogsOnly` parameter.

In the preceding example, the model binding targets are method parameters that are simple types. Targets may also be the properties of a complex type. After each property is successfully bound, model validation occurs for that property. The record of what data is bound to the model, and any binding or validation errors, is stored in ControllerBase.ModelState or PageModel.ModelState. To find out if this process was successful, the app checks the ModelState.IsValid flag.

Targets

Model binding tries to find values for the following kinds of targets:

- Parameters of the controller action method that a request is routed to.
- Parameters of the Razor Pages handler method that a request is routed to.
- Public properties of a controller or `PageModel` class, if specified by attributes.

[BindProperty] attribute

Can be applied to a public property of a controller or `PageModel` class to cause model binding to target that property:

```
public class EditModel : InstructorsPageModel
{
    [BindProperty]
    public Instructor Instructor { get; set; }
```

[BindProperties] attribute

Available in ASP.NET Core 2.1 and later. Can be applied to a controller or `PageModel` class to tell model binding to target all public properties of the class:

```
[BindProperties(SupportsGet = true)]
public class CreateModel : InstructorsPageModel
{
    public Instructor Instructor { get; set; }
```

Model binding for HTTP GET requests

By default, properties are not bound for HTTP GET requests. Typically, all you need for a GET request is a record ID parameter. The record ID is used to look up the item in the database. Therefore, there is no need to bind a property that holds an instance of the model. In scenarios where you do want properties bound to data from GET requests, set the `SupportsGet` property to `true`:

```
[BindProperty(Name = "ai_user", SupportsGet = true)]
public string ApplicationInsightsCookie { get; set; }
```

Sources

By default, model binding gets data in the form of key-value pairs from the following sources in an HTTP request:

1. Form fields
2. The request body (For controllers that have the [ApiController] attribute.)
3. Route data
4. Query string parameters
5. Uploaded files

For each target parameter or property, the sources are scanned in the order indicated in the preceding list. There are a few exceptions:

- Route data and query string values are used only for simple types.
- Uploaded files are bound only to target types that implement `IFormFile` or `IEnumerable<IFormFile>`.

If the default source is not correct, use one of the following attributes to specify the source:

- `[FromQuery]` - Gets values from the query string.
- `[FromRoute]` - Gets values from route data.
- `[FromForm]` - Gets values from posted form fields.
- `[FromBody]` - Gets values from the request body.
- `[FromHeader]` - Gets values from HTTP headers.

These attributes:

- Are added to model properties individually (not to the model class), as in the following example:

```
public class Instructor
{
    public int ID { get; set; }

    [FromQuery(Name = "Note")]
    public string NoteFromQueryString { get; set; }
```

- Optionally accept a model name value in the constructor. This option is provided in case the property name doesn't match the value in the request. For instance, the value in the request might be a header with a hyphen in its name, as in the following example:

```
public void OnGet([FromHeader(Name = "Accept-Language")] string language)
```

[FromBody] attribute

Apply the `[FromBody]` attribute to a parameter to populate its properties from the body of an HTTP request. The ASP.NET Core runtime delegates the responsibility of reading the body to an input formatter. Input formatters are explained later in this article.

When `[FromBody]` is applied to a complex type parameter, any binding source attributes applied to its properties are ignored. For example, the following `Create` action specifies that its `pet` parameter is populated from the body:

```
public ActionResult<Pet> Create([FromBody] Pet pet)
```

The `Pet` class specifies that its `Breed` property is populated from a query string parameter:

```
public class Pet
{
    public string Name { get; set; }

    [FromQuery] // Attribute is ignored.
    public string Breed { get; set; }
}
```

In the preceding example:

- The `[FromQuery]` attribute is ignored.
- The `Breed` property is not populated from a query string parameter.

Input formatters read only the body and don't understand binding source attributes. If a suitable value is found in the body, that value is used to populate the `Breed` property.

Don't apply `[FromBody]` to more than one parameter per action method. Once the request stream is read by an input formatter, it's no longer available to be read again for binding other `[FromBody]` parameters.

Additional sources

Source data is provided to the model binding system by *value providers*. You can write and register custom value providers that get data for model binding from other sources. For example, you might want data from cookies or session state. To get data from a new source:

- Create a class that implements `IValueProvider`.
- Create a class that implements `IValueProviderFactory`.
- Register the factory class in `Startup.ConfigureServices`.

The sample app includes a value provider and factory example that gets values from cookies. Here's the registration code in `Startup.ConfigureServices`:

```
services.AddMvc(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters()
.SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

The code shown puts the custom value provider after all the built-in value providers. To make it the first in the list, call `Insert(0, new CookieValueProviderFactory())` instead of `Add`.

No source for a model property

By default, a model state error isn't created if no value is found for a model property. The property is set to null or a default value:

- Nullable simple types are set to `null`.
- Non-nullable value types are set to `default(T)`. For example, a parameter `int id` is set to 0.
- For complex Types, model binding creates an instance by using the default constructor, without setting properties.
- Arrays are set to `Array.Empty<T>()`, except that `byte[]` arrays are set to `null`.

If model state should be invalidated when nothing is found in form fields for a model property, use the `[BindRequired]` attribute.

Note that this `[BindRequired]` behavior applies to model binding from posted form data, not to JSON or XML data in a request body. Request body data is handled by input formatters.

Type conversion errors

If a source is found but can't be converted into the target type, model state is flagged as invalid. The target parameter or property is set to null or a default value, as noted in the previous section.

In an API controller that has the `[ApiController]` attribute, invalid model state results in an automatic HTTP 400 response.

In a Razor page, redisplay the page with an error message:

```
public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _instructorsInMemoryStore.Add(Instructor);
    return RedirectToPage("./Index");
}
```

Client-side validation catches most bad data that would otherwise be submitted to a Razor Pages form. This validation makes it hard to trigger the preceding highlighted code. The sample app includes a **Submit with Invalid Date** button that puts bad data in the **Hire Date** field and submits the form. This button shows how the code for redisplaying the page works when data conversion errors occur.

When the page is redisplayed by the preceding code, the invalid input is not shown in the form field. This is because the model property has been set to null or a default value. The invalid input does appear in an error message. But if you want to redisplay the bad data in the form field, consider making the model property a string and doing the data conversion manually.

The same strategy is recommended if you don't want type conversion errors to result in model state errors. In that case, make the model property a string.

Simple types

The simple types that the model binder can convert source strings into include the following:

- Boolean
- Byte, SByte
- Char
- DateTime
- DateTimeOffset
- Decimal
- Double
- Enum
- Guid
- Int16, Int32, Int64
- Single
- TimeSpan
- UInt16, UInt32, UInt64
- Uri
- Version

Complex types

A complex type must have a public default constructor and public writable properties to bind. When model binding occurs, the class is instantiated using the public default constructor.

For each property of the complex type, model binding looks through the sources for the name pattern *prefix.property_name*. If nothing is found, it looks for just *property_name* without the prefix.

For binding to a parameter, the prefix is the parameter name. For binding to a `PageModel` public property, the prefix is the public property name. Some attributes have a `Prefix` property that lets you override the default usage of parameter or property name.

For example, suppose the complex type is the following `Instructor` class:

```
public class Instructor
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Prefix = parameter name

If the model to be bound is a parameter named `instructorToUpdate`:

```
public IActionResult OnPost(int? id, Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `instructorToUpdate.ID`. If that isn't found, it looks for `ID` without a prefix.

Prefix = property name

If the model to be bound is a property named `Instructor` of the controller or `PageModel` class:

```
[BindProperty]
public Instructor Instructor { get; set; }
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Custom prefix

If the model to be bound is a parameter named `instructorToUpdate` and a `Bind` attribute specifies `Instructor` as the prefix:

```
public IActionResult OnPost(
    int? id, [Bind(Prefix = "Instructor")] Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Attributes for complex type targets

Several built-in attributes are available for controlling model binding of complex types:

- `[BindRequired]`
- `[BindNever]`
- `[Bind]`

> **NOTE**
>
> These attributes affect model binding when posted form data is the source of values. They do not affect input formatters, which process posted JSON and XML request bodies. Input formatters are explained later in this article.
>
> See also the discussion of the `[Required]` attribute in Model validation.

[BindRequired] attribute

Can only be applied to model properties, not to method parameters. Causes model binding to add a model state error if binding cannot occur for a model's property. Here's an example:

```
public class InstructorWithCollection
{
    public int ID { get; set; }

    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    [Display(Name = "Hire Date")]
    [BindRequired]
    public DateTime HireDate { get; set; }
```

[BindNever] attribute

Can only be applied to model properties, not to method parameters. Prevents model binding from setting a model's property. Here's an example:

```
public class InstructorWithDictionary
{
    [BindNever]
    public int ID { get; set; }
```

[Bind] attribute

Can be applied to a class or a method parameter. Specifies which properties of a model should be included in model binding.

In the following example, only the specified properties of the `Instructor` model are bound when any handler or action method is called:

```
[Bind("LastName,FirstMidName,HireDate")]
public class Instructor
```

In the following example, only the specified properties of the `Instructor` model are bound when the `OnPost` method is called:

```
[HttpPost]
public IActionResult OnPost([Bind("LastName,FirstMidName,HireDate")] Instructor instructor)
```

The `[Bind]` attribute can be used to protect against overposting in *create* scenarios. It doesn't work well in edit scenarios because excluded properties are set to null or a default value instead of being left unchanged. For defense against overposting, view models are recommended rather than the `[Bind]` attribute. For more information, see Security note about overposting.

Collections

For targets that are collections of simple types, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the parameter to be bound is an array named `selectedCourses`:

  ```
  public IActionResult OnPost(int? id, int[] selectedCourses)
  ```

- Form or query string data can be in one of the following formats:

```
selectedCourses=1050&selectedCourses=2000
```

```
selectedCourses[0]=1050&selectedCourses[1]=2000
```

```
[0]=1050&[1]=2000
```

```
selectedCourses[a]=1050&selectedCourses[b]=2000&selectedCourses.index=a&selectedCour
ses.index=b
```

```
[a]=1050&[b]=2000&index=a&index=b
```

- The following format is supported only in form data:

```
selectedCourses[]=1050&selectedCourses[]=2000
```

- For all of the preceding example formats, model binding passes an array of two items to the `selectedCourses` parameter:
  - selectedCourses[0]=1050
  - selectedCourses[1]=2000

  Data formats that use subscript numbers (... [0] ... [1] ...) must ensure that they are numbered sequentially starting at zero. If there are any gaps in subscript numbering, all items after the gap are ignored. For example, if the subscripts are 0 and 2 instead of 0 and 1, the second item is ignored.

Dictionaries

For `Dictionary` targets, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the target parameter is a `Dictionary<int, string>` named `selectedCourses`:

```
public IActionResult OnPost(int? id, Dictionary<int, string> selectedCourses)
```

- The posted form or query string data can look like one of the following examples:

```
selectedCourses[1050]=Chemistry&selectedCourses[2000]=Economics
```

```
[1050]=Chemistry&selectedCourses[2000]=Economics
```

```
selectedCourses[0].Key=1050&selectedCourses[0].Value=Chemistry&
selectedCourses[1].Key=2000&selectedCourses[1].Value=Economics
```

```
[0].Key=1050&[0].Value=Chemistry&[1].Key=2000&[1].Value=Economics
```

- For all of the preceding example formats, model binding passes a dictionary of two items to the `selectedCourses` parameter:
  - selectedCourses["1050"]="Chemistry"
  - selectedCourses["2000"]="Economics"

Globalization behavior of model binding route data and query strings

The ASP.NET Core route value provider and query string value provider:

- Treat values as invariant culture.
- Expect that URLs are culture-invariant.

In contrast, values coming from form data undergo a culture-sensitive conversion. This is by design so that URLs are shareable across locales.

To make the ASP.NET Core route value provider and query string value provider undergo a culture-sensitive conversion:

- Inherit from IValueProviderFactory
- Copy the code from QueryStringValueProviderFactory or RouteValueValueProviderFactory
- Replace the culture value passed to the value provider constructor with CultureInfo.CurrentCulture
- Replace the default value provider factory in MVC options with your new one:

```
services.AddMvc(options =>
{
    var index = options.ValueProviderFactories.IndexOf(
        options.ValueProviderFactories.OfType<QueryStringValueProviderFactory>().Single());
    options.ValueProviderFactories[index] = new CulturedQueryStringValueProviderFactory();
});
```

```
public class CulturedQueryStringValueProviderFactory : IValueProviderFactory
{
    public Task CreateValueProviderAsync(ValueProviderFactoryContext context)
    {
        if (context == null)
        {
            throw new ArgumentNullException(nameof(context));
        }

        var query = context.ActionContext.HttpContext.Request.Query;
        if (query != null && query.Count > 0)
        {
            var valueProvider = new QueryStringValueProvider(
                BindingSource.Query,
                query,
                CultureInfo.CurrentCulture);

            context.ValueProviders.Add(valueProvider);
        }

        return Task.CompletedTask;
    }
}
```

Special data types

There are some special data types that model binding can handle.

IFormFile and IFormFileCollection

An uploaded file included in the HTTP request. Also supported is `IEnumerable<IFormFile>` for multiple files.

CancellationToken

Used to cancel activity in asynchronous controllers.

FormCollection

Used to retrieve all the values from posted form data.

Input formatters

Data in the request body can be in JSON, XML, or some other format. To parse this data, model binding uses an *input formatter* that is configured to handle a particular content type. By default, ASP.NET Core includes JSON based input formatters for handling JSON data. You can add other formatters for other content types.

ASP.NET Core selects input formatters based on the Consumes attribute. If no attribute is present, it uses the Content-Type header.

To use the built-in XML input formatters:

- Install the `Microsoft.AspNetCore.Mvc.Formatters.Xml` NuGet package.
- In `Startup.ConfigureServices`, call AddXmlSerializerFormatters or AddXmlDataContractSerializerFormatters.

  ```
  services.AddMvc(options =>
  {
      options.ValueProviderFactories.Add(new CookieValueProviderFactory());
      options.ModelMetadataDetailsProviders.Add(
          new ExcludeBindingMetadataProvider(typeof(System.Version)));
      options.ModelMetadataDetailsProviders.Add(
          new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
  })
  .AddXmlSerializerFormatters()
  .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
  ```

- Apply the `Consumes` attribute to controller classes or action methods that should expect XML in the request body.

  ```
  [HttpPost]
  [Consumes("application/xml")]
  public ActionResult<Pet> Create(Pet pet)
  ```

  For more information, see Introducing XML Serialization.

Exclude specified types from model binding

The model binding and validation systems' behavior is driven by ModelMetadata. You can customize `ModelMetadata` by adding a details provider to MvcOptions.ModelMetadataDetailsProviders. Built-in details providers are available for disabling model binding or validation for specified types.

To disable model binding on all models of a specified type, add an ExcludeBindingMetadataProvider in `Startup.ConfigureServices`. For example, to disable model binding on all models of type `System.Version`:

```
services.AddMvc(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters()
.SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

To disable validation on properties of a specified type, add a SuppressChildValidationMetadataProvider in `Startup.ConfigureServices`. For example, to disable validation on properties of type `System.Guid`:

```
services.AddMvc(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
    options.ModelMetadataDetailsProviders.Add(
        new ExcludeBindingMetadataProvider(typeof(System.Version)));
    options.ModelMetadataDetailsProviders.Add(
        new SuppressChildValidationMetadataProvider(typeof(System.Guid)));
})
.AddXmlSerializerFormatters()
.SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Custom model binders

You can extend model binding by writing a custom model binder and using the `[ModelBinder]` attribute to select it for a given target. Learn more about custom model binding.

Manual model binding

Model binding can be invoked manually by using the TryUpdateModelAsync method. The method is defined on both `ControllerBase` and `PageModel` classes. Method overloads let you specify the prefix and value provider to use. The method returns `false` if model binding fails. Here's an example:

```
if (await TryUpdateModelAsync<InstructorWithCollection>(
    newInstructor,
    "Instructor",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate))
{
    _instructorsInMemoryStore.Add(newInstructor);
    return RedirectToPage("./Index");
}
PopulateAssignedCourseData(newInstructor);
return Page();
```

[FromServices] attribute

This attribute's name follows the pattern of model binding attributes that specify a data source.

But it's not about binding data from a value provider. It gets an instance of a type from the dependency injection container. Its purpose is to provide an alternative to constructor injection for when you need a service only if a particular method is called.

Additional resources

- Model validation in ASP.NET Core MVC
- Custom Model Binding in ASP.NET Core

Custom Model Binding in ASP.NET Core

9/22/2020 • 16 minutes to read • Edit Online

By Steve Smith and Kirk Larkin

Model binding allows controller actions to work directly with model types (passed in as method arguments), rather than HTTP requests. Mapping between incoming request data and application models is handled by model binders. Developers can extend the built-in model binding functionality by implementing custom model binders (though typically, you don't need to write your own provider).

View or download sample code (how to download)

Default model binder limitations

The default model binders support most of the common .NET Core data types and should meet most developers' needs. They expect to bind text-based input from the request directly to model types. You might need to transform the input prior to binding it. For example, when you have a key that can be used to look up model data. You can use a custom model binder to fetch data based on the key.

Model binding review

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string in the input. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter`. We recommended you create a type converter if you have a simple `string` -> `SomeType` mapping that doesn't require external resources.

Before creating your own custom model binder, it's worth reviewing how existing model binders are implemented. Consider the ByteArrayModelBinder which can be used to convert base64-encoded strings into byte arrays. The byte arrays are often stored as files or database BLOB fields.

Working with the ByteArrayModelBinder

Base64-encoded strings can be used to represent binary data. For example, an image can be encoded as a string. The sample includes an image as a base64-encoded string in Base64String.txt.

ASP.NET Core MVC can take a base64-encoded string and use a `ByteArrayModelBinder` to convert it into a byte array. The ByteArrayModelBinderProvider maps `byte[]` arguments to `ByteArrayModelBinder`:

```
public IModelBinder GetBinder(ModelBinderProviderContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (context.Metadata.ModelType == typeof(byte[]))
    {
        var loggerFactory = context.Services.GetRequiredService<ILoggerFactory>();
        return new ByteArrayModelBinder(loggerFactory);
    }

    return null;
}
```

When creating your own custom model binder, you can implement your own `IModelBinderProvider` type, or use the ModelBinderAttribute.

The following example shows how to use `ByteArrayModelBinder` to convert a base64-encoded string to a `byte[]` and save the result to a file:

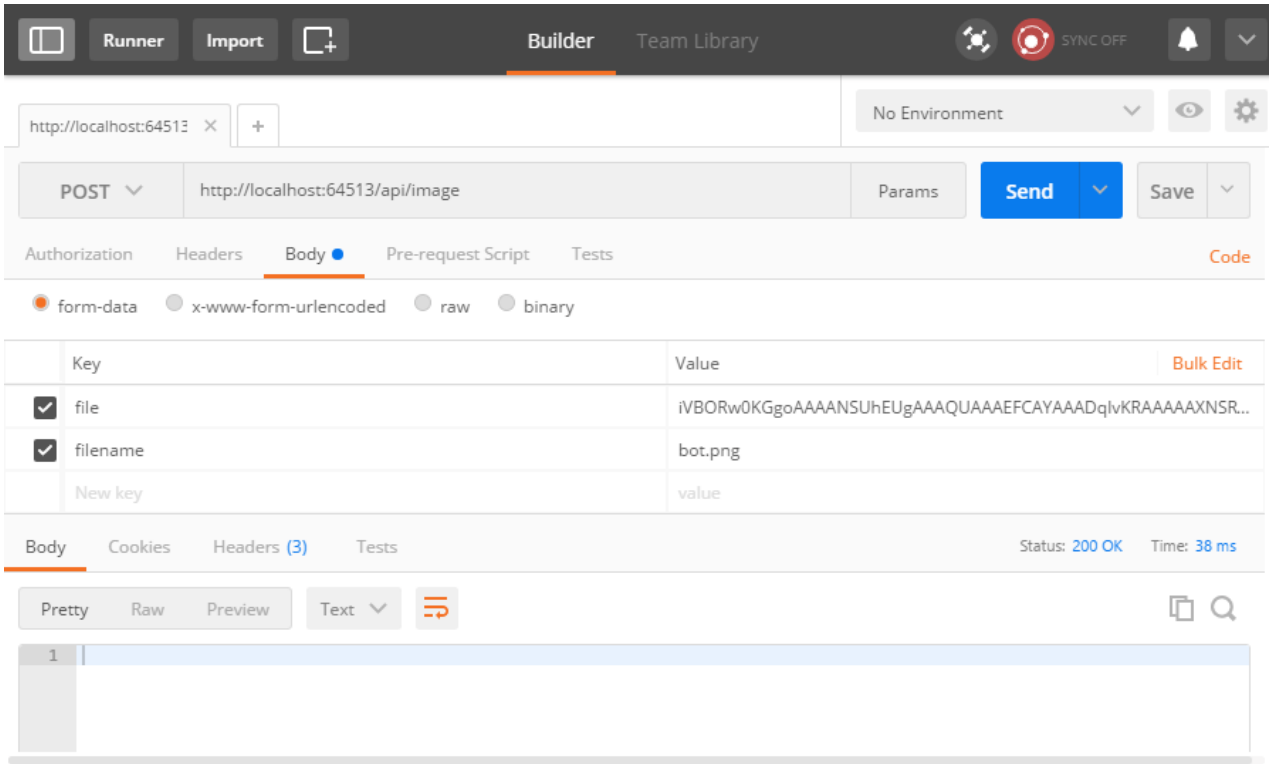
```
[HttpPost]
public void Post([FromForm] byte[] file, string filename)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
    }

    System.IO.File.WriteAllBytes(filePath, file);
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

You can POST a base64-encoded string to this api method using a tool like Postman:



As long as the binder can bind request data to appropriately named properties or arguments, model binding will succeed. The following example shows how to use `ByteArrayModelBinder` with a view model:

```
[HttpPost("Profile")]
public void SaveProfile([FromForm] ProfileViewModel model)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
    }

    System.IO.File.WriteAllBytes(filePath, model.File);
}

public class ProfileViewModel
{
    public byte[] File { get; set; }
    public string FileName { get; set; }
}
```

Custom model binder sample

In this section we'll implement a custom model binder that:

- Converts incoming request data into strongly typed key arguments.
- Uses Entity Framework Core to fetch the associated entity.
- Passes the associated entity as an argument to the action method.

The following sample uses the `ModelBinder` attribute on the `Author` model:

```
using CustomModelBindingSample.Binders;
using Microsoft.AspNetCore.Mvc;

namespace CustomModelBindingSample.Data
{
    [ModelBinder(BinderType = typeof(AuthorEntityBinder))]
    public class Author
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string GitHub { get; set; }
        public string Twitter { get; set; }
        public string BlogUrl { get; set; }
    }
}
```

In the preceding code, the `ModelBinder` attribute specifies the type of `IModelBinder` that should be used to bind `Author` action parameters.

The following `AuthorEntityBinder` class binds an `Author` parameter by fetching the entity from a data source using Entity Framework Core and an `authorId`:

```
public class AuthorEntityBinder : IModelBinder
{
    private readonly AuthorContext _context;

    public AuthorEntityBinder(AuthorContext context)
    {
        _context = context;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException(nameof(bindingContext));
        }

        var modelName = bindingContext.ModelName;

        // Try to fetch the value of the argument by name
        var valueProviderResult = bindingContext.ValueProvider.GetValue(modelName);

        if (valueProviderResult == ValueProviderResult.None)
        {
            return Task.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName, valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return Task.CompletedTask;
        }

        if (!int.TryParse(value, out var id))
        {
            // Non-integer arguments result in model state errors
            bindingContext.ModelState.TryAddModelError(
                modelName, "Author Id must be an integer.");

            return Task.CompletedTask;
        }

        // Model will be null if not found, including for
        // out of range id values (0, -3, etc.)
        var model = _context.Authors.Find(id);
        bindingContext.Result = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}
```

NOTE

The preceding `AuthorEntityBinder` class is intended to illustrate a custom model binder. The class isn't intended to illustrate best practices for a lookup scenario. For lookup, bind the `authorId` and query the database in an action method. This approach separates model binding failures from `NotFound` cases.

The following code shows how to use the `AuthorEntityBinder` in an action method:

```
[HttpGet("get/{authorId}")]
public IActionResult Get(Author author)
{
    if (author == null)
    {
        return NotFound();
    }

    return Ok(author);
}
```

The `ModelBinder` attribute can be used to apply the `AuthorEntityBinder` to parameters that don't use default conventions:

```
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")] Author author)
{
    if (author == null)
    {
        return NotFound();
    }

    return Ok(author);
}
```

In this example, since the name of the argument isn't the default `authorId`, it's specified on the parameter using the `ModelBinder` attribute. Both the controller and action method are simplified compared to looking up the entity in the action method. The logic to fetch the author using Entity Framework Core is moved to the model binder. This can be a considerable simplification when you have several methods that bind to the `Author` model.

You can apply the `ModelBinder` attribute to individual model properties (such as on a viewmodel) or to action method parameters to specify a certain model binder or model name for just that type or action.

Implementing a ModelBinderProvider

Instead of applying an attribute, you can implement `IModelBinderProvider`. This is how the built-in framework binders are implemented. When you specify the type your binder operates on, you specify the type of argument it produces, **not** the input your binder accepts. The following binder provider works with the `AuthorEntityBinder`. When it's added to MVC's collection of providers, you don't need to use the `ModelBinder` attribute on `Author` or `Author`-typed parameters.

```
using CustomModelBindingSample.Data;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using System;

namespace CustomModelBindingSample.Binders
{
    public class AuthorEntityBinderProvider : IModelBinderProvider
    {
        public IModelBinder GetBinder(ModelBinderProviderContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException(nameof(context));
            }

            if (context.Metadata.ModelType == typeof(Author))
            {
                return new BinderTypeModelBinder(typeof(AuthorEntityBinder));
            }

            return null;
        }
    }
}
```

> Note: The preceding code returns a `BinderTypeModelBinder`. `BinderTypeModelBinder` acts as a factory for model binders and provides dependency injection (DI). The `AuthorEntityBinder` requires DI to access EF Core. Use `BinderTypeModelBinder` if your model binder requires services from DI.

To use a custom model binder provider, add it in `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AuthorContext>(options => options.UseInMemoryDatabase("Authors"));

    services.AddControllers(options =>
    {
        options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
    });
}
```

When evaluating model binders, the collection of providers is examined in order. The first provider that returns a binder that matches the input model is used. Adding your provider to the end of the collection may thus result in a built-in model binder being called before your custom binder has a chance. In this example, the custom provider is added to the beginning of the collection to ensure it's always used for `Author` action arguments.

Polymorphic model binding

Binding to different models of derived types is known as polymorphic model binding. Polymorphic custom model binding is required when the request value must be bound to the specific derived model type. Polymorphic model binding:

- Isn't typical for a REST API that's designed to interoperate with all languages.
- Makes it difficult to reason about the bound models.

However, if an app requires polymorphic model binding, an implementation might look like the following code:

```
public abstract class Device
{
```

```
    public string Kind { get; set; }
}

public class Laptop : Device
{
    public string CPUIndex { get; set; }
}

public class SmartPhone : Device
{
    public string ScreenSize { get; set; }
}

public class DeviceModelBinderProvider : IModelBinderProvider
{
    public IModelBinder GetBinder(ModelBinderProviderContext context)
    {
        if (context.Metadata.ModelType != typeof(Device))
        {
            return null;
        }

        var subclasses = new[] { typeof(Laptop), typeof(SmartPhone), };

        var binders = new Dictionary<Type, (ModelMetadata, IModelBinder)>();
        foreach (var type in subclasses)
        {
            var modelMetadata = context.MetadataProvider.GetMetadataForType(type);
            binders[type] = (modelMetadata, context.CreateBinder(modelMetadata));
        }

        return new DeviceModelBinder(binders);
    }
}

public class DeviceModelBinder : IModelBinder
{
    private Dictionary<Type, (ModelMetadata, IModelBinder)> binders;

    public DeviceModelBinder(Dictionary<Type, (ModelMetadata, IModelBinder)> binders)
    {
        this.binders = binders;
    }

    public async Task BindModelAsync(ModelBindingContext bindingContext)
    {
        var modelKindName = ModelNames.CreatePropertyModelName(bindingContext.ModelName, nameof(Device.Kind));
        var modelTypeValue = bindingContext.ValueProvider.GetValue(modelKindName).FirstValue;

        IModelBinder modelBinder;
        ModelMetadata modelMetadata;
        if (modelTypeValue == "Laptop")
        {
            (modelMetadata, modelBinder) = binders[typeof(Laptop)];
        }
        else if (modelTypeValue == "SmartPhone")
        {
            (modelMetadata, modelBinder) = binders[typeof(SmartPhone)];
        }
        else
        {
            bindingContext.Result = ModelBindingResult.Failed();
            return;
        }

        var newBindingContext = DefaultModelBindingContext.CreateBindingContext(
            bindingContext.ActionContext,
            bindingContext.ValueProvider,
            modelMetadata,
```

```
            bindingInfo: null,
            bindingContext.ModelName);

        await modelBinder.BindModelAsync(newBindingContext);
        bindingContext.Result = newBindingContext.Result;

        if (newBindingContext.Result.IsModelSet)
        {
            // Setting the ValidationState ensures properties on derived types are correctly
            bindingContext.ValidationState[newBindingContext.Result] = new ValidationStateEntry
            {
                Metadata = modelMetadata,
            };
        }
    }
}
```

Recommendations and best practices

Custom model binders:

- Shouldn't attempt to set status codes or return results (for example, 404 Not Found). If model binding fails, an action filter or logic within the action method itself should handle the failure.
- Are most useful for eliminating repetitive code and cross-cutting concerns from action methods.
- Typically shouldn't be used to convert a string into a custom type, a TypeConverter is usually a better option.

By Steve Smith

Model binding allows controller actions to work directly with model types (passed in as method arguments), rather than HTTP requests. Mapping between incoming request data and application models is handled by model binders. Developers can extend the built-in model binding functionality by implementing custom model binders (though typically, you don't need to write your own provider).

View or download sample code (how to download)

Default model binder limitations

The default model binders support most of the common .NET Core data types and should meet most developers' needs. They expect to bind text-based input from the request directly to model types. You might need to transform the input prior to binding it. For example, when you have a key that can be used to look up model data. You can use a custom model binder to fetch data based on the key.

Model binding review

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string in the input. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter`. We recommended you create a type converter if you have a simple `string` -> `SomeType` mapping that doesn't require external resources.

Before creating your own custom model binder, it's worth reviewing how existing model binders are implemented. Consider the ByteArrayModelBinder which can be used to convert base64-encoded strings into byte arrays. The byte arrays are often stored as files or database BLOB fields.

Working with the ByteArrayModelBinder

Base64-encoded strings can be used to represent binary data. For example, an image can be encoded as a string. The sample includes an image as a base64-encoded string in Base64String.txt.

ASP.NET Core MVC can take a base64-encoded string and use a `ByteArrayModelBinder` to convert it into a byte

array. The ByteArrayModelBinderProvider maps `byte[]` arguments to `ByteArrayModelBinder`:

```
public IModelBinder GetBinder(ModelBinderProviderContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (context.Metadata.ModelType == typeof(byte[]))
    {
        return new ByteArrayModelBinder();
    }

    return null;
}
```

When creating your own custom model binder, you can implement your own `IModelBinderProvider` type, or use the ModelBinderAttribute.

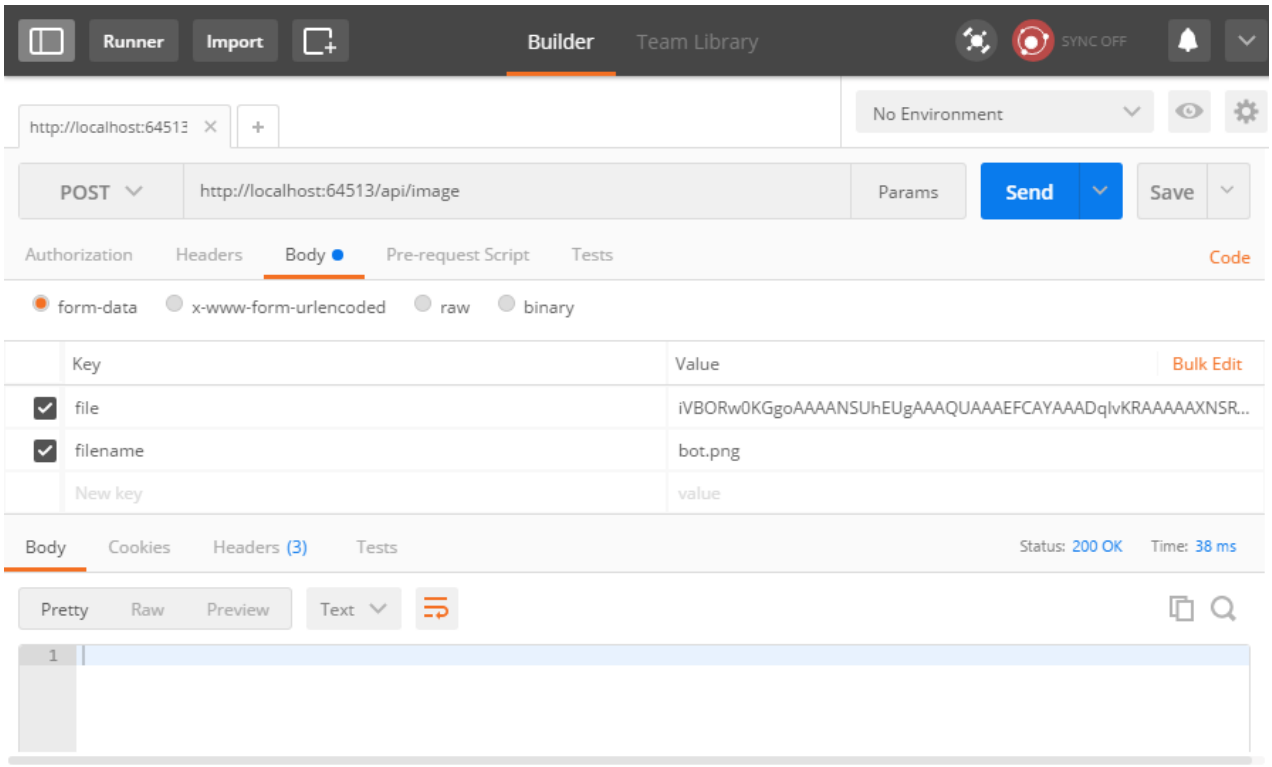
The following example shows how to use `ByteArrayModelBinder` to convert a base64-encoded string to a `byte[]` and save the result to a file:

```
[HttpPost]
public void Post([FromForm] byte[] file, string filename)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
    }

    System.IO.File.WriteAllBytes(filePath, file);
}
```

You can POST a base64-encoded string to this api method using a tool like Postman:



As long as the binder can bind request data to appropriately named properties or arguments, model binding will succeed. The following example shows how to use `ByteArrayModelBinder` with a view model:

```
[HttpPost("Profile")]
public void SaveProfile([FromForm] ProfileViewModel model)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
    }

    System.IO.File.WriteAllBytes(filePath, model.File);
}

public class ProfileViewModel
{
    public byte[] File { get; set; }
    public string FileName { get; set; }
}
```

Custom model binder sample

In this section we'll implement a custom model binder that:

- Converts incoming request data into strongly typed key arguments.
- Uses Entity Framework Core to fetch the associated entity.
- Passes the associated entity as an argument to the action method.

The following sample uses the `ModelBinder` attribute on the `Author` model:

```
using CustomModelBindingSample.Binders;
using Microsoft.AspNetCore.Mvc;

namespace CustomModelBindingSample.Data
{
    [ModelBinder(BinderType = typeof(AuthorEntityBinder))]
    public class Author
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string GitHub { get; set; }
        public string Twitter { get; set; }
        public string BlogUrl { get; set; }
    }
}
```

In the preceding code, the `ModelBinder` attribute specifies the type of `IModelBinder` that should be used to bind `Author` action parameters.

The following `AuthorEntityBinder` class binds an `Author` parameter by fetching the entity from a data source using Entity Framework Core and an `authorId`:

```
public class AuthorEntityBinder : IModelBinder
{
    private readonly AppDbContext _db;

    public AuthorEntityBinder(AppDbContext db)
    {
        _db = db;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException(nameof(bindingContext));
        }

        var modelName = bindingContext.ModelName;

        // Try to fetch the value of the argument by name
        var valueProviderResult = bindingContext.ValueProvider.GetValue(modelName);

        if (valueProviderResult == ValueProviderResult.None)
        {
            return Task.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName, valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return Task.CompletedTask;
        }

        if (!int.TryParse(value, out var id))
        {
            // Non-integer arguments result in model state errors
            bindingContext.ModelState.TryAddModelError(
                modelName, "Author Id must be an integer.");

            return Task.CompletedTask;
        }

        // Model will be null if not found, including for
        // out of range id values (0, -3, etc.)
        var model = _db.Authors.Find(id);
        bindingContext.Result = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}
```

NOTE

The preceding `AuthorEntityBinder` class is intended to illustrate a custom model binder. The class isn't intended to illustrate best practices for a lookup scenario. For lookup, bind the `authorId` and query the database in an action method. This approach separates model binding failures from `NotFound` cases.

The following code shows how to use the `AuthorEntityBinder` in an action method:

```
[HttpGet("get/{authorId}")]
public IActionResult Get(Author author)
{
    if (author == null)
    {
        return NotFound();
    }

    return Ok(author);
}
```

The `ModelBinder` attribute can be used to apply the `AuthorEntityBinder` to parameters that don't use default conventions:

```
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")] Author author)
{
    if (author == null)
    {
        return NotFound();
    }

    return Ok(author);
}
```

In this example, since the name of the argument isn't the default `authorId`, it's specified on the parameter using the `ModelBinder` attribute. Both the controller and action method are simplified compared to looking up the entity in the action method. The logic to fetch the author using Entity Framework Core is moved to the model binder. This can be a considerable simplification when you have several methods that bind to the `Author` model.

You can apply the `ModelBinder` attribute to individual model properties (such as on a viewmodel) or to action method parameters to specify a certain model binder or model name for just that type or action.

Implementing a ModelBinderProvider

Instead of applying an attribute, you can implement `IModelBinderProvider`. This is how the built-in framework binders are implemented. When you specify the type your binder operates on, you specify the type of argument it produces, **not** the input your binder accepts. The following binder provider works with the `AuthorEntityBinder`. When it's added to MVC's collection of providers, you don't need to use the `ModelBinder` attribute on `Author` or `Author`-typed parameters.

```
using CustomModelBindingSample.Data;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using System;

namespace CustomModelBindingSample.Binders
{
    public class AuthorEntityBinderProvider : IModelBinderProvider
    {
        public IModelBinder GetBinder(ModelBinderProviderContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException(nameof(context));
            }

            if (context.Metadata.ModelType == typeof(Author))
            {
                return new BinderTypeModelBinder(typeof(AuthorEntityBinder));
            }

            return null;
        }
    }
}
```

> Note: The preceding code returns a `BinderTypeModelBinder`. `BinderTypeModelBinder` acts as a factory for model binders and provides dependency injection (DI). The `AuthorEntityBinder` requires DI to access EF Core. Use `BinderTypeModelBinder` if your model binder requires services from DI.

To use a custom model binder provider, add it in `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options => options.UseInMemoryDatabase("App"));

    services.AddMvc(options =>
        {
            // add custom binder to beginning of collection
            options.ModelBinderProviders.Insert(0, new AuthorEntityBinderProvider());
        })
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

When evaluating model binders, the collection of providers is examined in order. The first provider that returns a binder is used. Adding your provider to the end of the collection may result in a built-in model binder being called before your custom binder has a chance. In this example, the custom provider is added to the beginning of the collection to ensure it's used for `Author` action arguments.

Polymorphic model binding

Binding to different models of derived types is known as polymorphic model binding. Polymorphic custom model binding is required when the request value must be bound to the specific derived model type. Polymorphic model binding:

- Isn't typical for a REST API that's designed to interoperate with all languages.
- Makes it difficult to reason about the bound models.

However, if an app requires polymorphic model binding, an implementation might look like the following code:

```
public abstract class Device
{
    public string Kind { get; set; }
}

public class Laptop : Device
{
    public string CPUIndex { get; set; }
}

public class SmartPhone : Device
{
    public string ScreenSize { get; set; }
}

public class DeviceModelBinderProvider : IModelBinderProvider
{
    public IModelBinder GetBinder(ModelBinderProviderContext context)
    {
        if (context.Metadata.ModelType != typeof(Device))
        {
            return null;
        }

        var subclasses = new[] { typeof(Laptop), typeof(SmartPhone), };

        var binders = new Dictionary<Type, (ModelMetadata, IModelBinder)>();
        foreach (var type in subclasses)
        {
            var modelMetadata = context.MetadataProvider.GetMetadataForType(type);
            binders[type] = (modelMetadata, context.CreateBinder(modelMetadata));
        }

        return new DeviceModelBinder(binders);
    }
}

public class DeviceModelBinder : IModelBinder
{
    private Dictionary<Type, (ModelMetadata, IModelBinder)> binders;

    public DeviceModelBinder(Dictionary<Type, (ModelMetadata, IModelBinder)> binders)
    {
        this.binders = binders;
    }

    public async Task BindModelAsync(ModelBindingContext bindingContext)
    {
        var modelKindName = ModelNames.CreatePropertyModelName(bindingContext.ModelName, nameof(Device.Kind));
        var modelTypeValue = bindingContext.ValueProvider.GetValue(modelKindName).FirstValue;

        IModelBinder modelBinder;
        ModelMetadata modelMetadata;
        if (modelTypeValue == "Laptop")
        {
            (modelMetadata, modelBinder) = binders[typeof(Laptop)];
        }
        else if (modelTypeValue == "SmartPhone")
        {
            (modelMetadata, modelBinder) = binders[typeof(SmartPhone)];
        }
        else
        {
            bindingContext.Result = ModelBindingResult.Failed();
            return;
        }

        var newBindingContext = DefaultModelBindingContext.CreateBindingContext(
            bindingContext.ActionContext,
```

```
            bindingContext.ValueProvider,
            modelMetadata,
            bindingInfo: null,
            bindingContext.ModelName);

        await modelBinder.BindModelAsync(newBindingContext);
        bindingContext.Result = newBindingContext.Result;

        if (newBindingContext.Result.IsModelSet)
        {
            // Setting the ValidationState ensures properties on derived types are correctly
            bindingContext.ValidationState[newBindingContext.Result] = new ValidationStateEntry
            {
                Metadata = modelMetadata,
            };
        }
    }
}
```

Recommendations and best practices

Custom model binders:

- Shouldn't attempt to set status codes or return results (for example, 404 Not Found). If model binding fails, an action filter or logic within the action method itself should handle the failure.
- Are most useful for eliminating repetitive code and cross-cutting concerns from action methods.
- Typically shouldn't be used to convert a string into a custom type, a TypeConverter is usually a better option.

Model validation in ASP.NET Core MVC and Razor Pages

9/22/2020 • 35 minutes to read • Edit Online

By Kirk Larkin

This article explains how to validate user input in an ASP.NET Core MVC or Razor Pages app.

View or download sample code (how to download).

Model state

Model state represents errors that come from two subsystems: model binding and model validation. Errors that originate from model binding are generally data conversion errors. For example, an "x" is entered in an integer field. Model validation occurs after model binding and reports errors where data doesn't conform to business rules. For example, a 0 is entered in a field that expects a rating between 1 and 5.

Both model binding and model validation occur before the execution of a controller action or a Razor Pages handler method. For web apps, it's the app's responsibility to inspect `ModelState.IsValid` and react appropriately. Web apps typically redisplay the page with an error message:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movies.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Web API controllers don't have to check `ModelState.IsValid` if they have the `[ApiController]` attribute. In that case, an automatic HTTP 400 response containing error details is returned when model state is invalid. For more information, see Automatic HTTP 400 responses.

Rerun validation

Validation is automatic, but you might want to repeat it manually. For example, you might compute a value for a property and want to rerun validation after setting the property to the computed value. To rerun validation, call the `TryValidateModel` method, as shown here:

```
Movie.ReleaseDate = modifiedReleaseDate;

if (!TryValidateModel(Movie, nameof(Movie)))
{
    return Page();
}

_context.Movies.Add(Movie);
await _context.SaveChangesAsync();

return RedirectToPage("./Index");
```

Validation attributes

Validation attributes let you specify validation rules for model properties. The following example from the sample app shows a model class that is annotated with validation attributes. The `[ClassicMovie]` attribute is a custom validation attribute and the others are built-in. Not shown is `[ClassicMovieWithClientValidator]`. `[ClassicMovieWithClientValidator]` shows an alternative way to implement a custom attribute.

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

Built-in attributes

Here are some of the built-in validation attributes:

- `[CreditCard]`: Validates that the property has a credit card format. Requires jQuery Validation Additional Methods.
- `[Compare]`: Validates that two properties in a model match.
- `[EmailAddress]`: Validates that the property has an email format.
- `[Phone]`: Validates that the property has a telephone number format.
- `[Range]`: Validates that the property value falls within a specified range.
- `[RegularExpression]`: Validates that the property value matches a specified regular expression.
- `[Required]`: Validates that the field is not null. See `[Required]` attribute for details about this

attribute's behavior.

- `[StringLength]`: Validates that a string property value doesn't exceed a specified length limit.
- `[Url]`: Validates that the property has a URL format.
- `[Remote]`: Validates input on the client by calling an action method on the server. See `[Remote]` attribute for details about this attribute's behavior.

A complete list of validation attributes can be found in the System.ComponentModel.DataAnnotations namespace.

Error messages

Validation attributes let you specify the error message to be displayed for invalid input. For example:

```
[StringLength(8, ErrorMessage = "Name length can't be more than 8.")]
```

Internally, the attributes call `String.Format` with a placeholder for the field name and sometimes additional placeholders. For example:

```
[StringLength(8, ErrorMessage = "{0} length must be between {2} and {1}.", MinimumLength = 6)]
```

When applied to a `Name` property, the error message created by the preceding code would be "Name length must be between 6 and 8.".

To find out which parameters are passed to `String.Format` for a particular attribute's error message, see the DataAnnotations source code.

[Required] attribute

The validation system in .NET Core 3.0 and later treats non-nullable parameters or bound properties as if they had a `[Required]` attribute. Value types such as `decimal` and `int` are non-nullable. This behavior can be disabled by configuring SuppressImplicitRequiredAttributeForNonNullableReferenceTypes in `Startup.ConfigureServices`:

```
services.AddControllers(options =>
options.SuppressImplicitRequiredAttributeForNonNullableReferenceTypes = true);
```

[Required] validation on the server

On the server, a required value is considered missing if the property is null. A non-nullable field is always valid, and the `[Required]` attribute's error message is never displayed.

However, model binding for a non-nullable property may fail, resulting in an error message such as `The value '' is invalid`. To specify a custom error message for server-side validation of non-nullable types, you have the following options:

- Make the field nullable (for example, `decimal?` instead of `decimal`). Nullable<T> value types are treated like standard nullable types.
- Specify the default error message to be used by model binding, as shown in the following example:

```
services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

services.AddSingleton<IValidationAttributeAdapterProvider,
    CustomValidationAttributeAdapterProvider>();
```

For more information about model binding errors that you can set default messages for, see DefaultModelBindingMessageProvider.

[Required] validation on the client

Non-nullable types and strings are handled differently on the client compared to the server. On the client:

- A value is considered present only if input is entered for it. Therefore, client-side validation handles non-nullable types the same as nullable types.
- Whitespace in a string field is considered valid input by the jQuery Validation required method. Server-side validation considers a required string field invalid if only whitespace is entered.

As noted earlier, non-nullable types are treated as though they had a `[Required]` attribute. That means you get client-side validation even if you don't apply the `[Required]` attribute. But if you don't use the attribute, you get a default error message. To specify a custom error message, use the attribute.

[Remote] attribute

The `[Remote]` attribute implements client-side validation that requires calling a method on the server to determine whether field input is valid. For example, the app may need to verify whether a user name is already in use.

To implement remote validation:

1. Create an action method for JavaScript to call. The jQuery Validation remote method expects a JSON response:

   - `true` means the input data is valid.
   - `false`, `undefined`, or `null` means the input is invalid. Display the default error message.
   - Any other string means the input is invalid. Display the string as a custom error message.

   Here's an example of an action method that returns a custom error message:

   ```
   [AcceptVerbs("GET", "POST")]
   public IActionResult VerifyEmail(string email)
   {
       if (!_userService.VerifyEmail(email))
       {
           return Json($"Email {email} is already in use.");
       }

       return Json(true);
   }
   ```

2. In the model class, annotate the property with a `[Remote]` attribute that points to the

validation action method, as shown in the following example:

```
[Remote(action: "VerifyEmail", controller: "Users")]
public string Email { get; set; }
```

The `[Remote]` attribute is in the `Microsoft.AspNetCore.Mvc` namespace.

Additional fields

The `AdditionalFields` property of the `[Remote]` attribute lets you validate combinations of fields against data on the server. For example, if the `User` model had `FirstName` and `LastName` properties, you might want to verify that no existing users already have that pair of names. The following example shows how to use `AdditionalFields`:

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(LastName))]
[Display(Name = "First Name")]
public string FirstName { get; set; }

[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName))]
[Display(Name = "Last Name")]
public string LastName { get; set; }
```

`AdditionalFields` could be set explicitly to the strings "FirstName" and "LastName", but using the nameof operator simplifies later refactoring. The action method for this validation must accept both `firstName` and `lastName` arguments:

```
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userService.VerifyName(firstName, lastName))
    {
        return Json($"A user named {firstName} {lastName} already exists.");
    }

    return Json(true);
}
```

When the user enters a first or last name, JavaScript makes a remote call to see if that pair of names has been taken.

To validate two or more additional fields, provide them as a comma-delimited list. For example, to add a `MiddleName` property to the model, set the `[Remote]` attribute as shown in the following example:

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName) + "," +
nameof(LastName))]
public string MiddleName { get; set; }
```

`AdditionalFields`, like all attribute arguments, must be a constant expression. Therefore, don't use an interpolated string or call Join to initialize `AdditionalFields`.

Alternatives to built-in attributes

If you need validation not provided by built-in attributes, you can:

- Create custom attributes.
- Implement IValidatableObject.

Custom attributes

For scenarios that the built-in validation attributes don't handle, you can create custom validation attributes. Create a class that inherits from ValidationAttribute, and override the IsValid method.

The `IsValid` method accepts an object named *value*, which is the input to be validated. An overload also accepts a `ValidationContext` object, which provides additional information, such as the model instance created by model binding.

The following example validates that the release date for a movie in the *Classic* genre isn't later than a specified year. The `[ClassicMovie]` attribute:

- Is only run on the server.
- For Classic movies, validates the release date:

```
public class ClassicMovieAttribute : ValidationAttribute
{
    public ClassicMovieAttribute(int year)
    {
        Year = year;
    }

    public int Year { get; }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than {Year}.";

    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value).Year;

        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
}
```

The `movie` variable in the preceding example represents a `Movie` object that contains the data from the form submission. When validation fails, a `ValidationResult` with an error message is returned.

IValidatableObject

The preceding example works only with `Movie` types. Another option for class-level validation is to implement `IValidatableObject` in the model class, as shown in the following example:

```
public class ValidatableMovie : IValidatableObject
{
    private const int _classicYear = 1960;

    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
        {
            yield return new ValidationResult(
                $"Classic movies must have a release year no later than {_classicYear}.",
                new[] { nameof(ReleaseDate) });
        }
    }
}
```

Top-level node validation

Top-level nodes include:

- Action parameters
- Controller properties
- Page handler parameters
- Page model properties

Model-bound top-level nodes are validated in addition to validating model properties. In the following example from the sample app, the `VerifyPhone` method uses the RegularExpressionAttribute to validate the `phone` action parameter:

```
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyPhone(
    [RegularExpression(@"^\d{3}-\d{3}-\d{4}$")] string phone)
{
    if (!ModelState.IsValid)
    {
        return Json($"Phone {phone} has an invalid format. Format: ###-###-####");
    }

    return Json(true);
}
```

Top-level nodes can use BindRequiredAttribute with validation attributes. In the following example from the sample app, the `CheckAge` method specifies that the `age` parameter must be bound from the query string when the form is submitted:

```
[HttpPost]
public IActionResult CheckAge([BindRequired, FromQuery] int age)
{
```

In the Check Age page (*CheckAge.cshtml*), there are two forms. The first form submits an `Age` value of `99` as a query string parameter: `https://localhost:5001/Users/CheckAge?Age=99`.

When a properly formatted `age` parameter from the query string is submitted, the form validates.

The second form on the Check Age page submits the `Age` value in the body of the request, and validation fails. Binding fails because the `age` parameter must come from a query string.

Maximum errors

Validation stops when the maximum number of errors is reached (200 by default). You can configure this number with the following code in `Startup.ConfigureServices`:

```
services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

services.AddSingleton<IValidationAttributeAdapterProvider,
    CustomValidationAttributeAdapterProvider>();
```

Maximum recursion

ValidationVisitor traverses the object graph of the model being validated. For models that are deep or are infinitely recursive, validation may result in stack overflow. MvcOptions.MaxValidationDepth provides a way to stop validation early if the visitor recursion exceeds a configured depth. The default value of `MvcOptions.MaxValidationDepth` is 32.

Automatic short-circuit

Validation is automatically short-circuited (skipped) if the model graph doesn't require validation. Objects that the runtime skips validation for include collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`) and complex object graphs that don't have any validators.

Disable validation

To disable validation:

1. Create an implementation of `IObjectModelValidator` that doesn't mark any fields as invalid.

```
public class NullObjectModelValidator : IObjectModelValidator
{
    public void Validate(ActionContext actionContext,
        ValidationStateDictionary validationState, string prefix, object model)
    {

    }
}
```

2. Add the following code to `Startup.ConfigureServices` to replace the default `IObjectModelValidator` implementation in the dependency injection container.

```
services.AddSingleton<IObjectModelValidator, NullObjectModelValidator>();
```

You might still see model state errors that originate from model binding.

Client-side validation

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

Client-side validation avoids an unnecessary round trip to the server when there are input errors on a form. The following script references in *_Layout.cshtml* and *_ValidationScriptsPartial.cshtml* support client-side validation:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validate/1.19.1/jquery.validate.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-
unobtrusive/3.2.11/jquery.validate.unobtrusive.min.js"></script>
```

The jQuery Unobtrusive Validation script is a custom Microsoft front-end library that builds on the popular jQuery Validation plugin. Without jQuery Unobtrusive Validation, you would have to code the same validation logic in two places: once in the server-side validation attributes on model properties, and then again in client-side scripts. Instead, Tag Helpers and HTML helpers use the validation attributes and type metadata from model properties to render HTML 5 `data-` attributes for the form elements that need validation. jQuery Unobtrusive Validation parses the `data-` attributes and passes the logic to jQuery Validation, effectively "copying" the server-side validation logic to the client. You can display validation errors on the client using tag helpers as shown here:

```
<div class="form-group">
    <label asp-for="Movie.ReleaseDate" class="control-label"></label>
    <input asp-for="Movie.ReleaseDate" class="form-control" />
    <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
</div>
```

The preceding tag helpers render the following HTML:

```
<div class="form-group">
    <label class="control-label" for="Movie_ReleaseDate">Release Date</label>
    <input class="form-control" type="date" data-val="true"
        data-val-required="The Release Date field is required."
        id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
    <span class="text-danger field-validation-valid"
        data-valmsg-for="Movie.ReleaseDate" data-valmsg-replace="true"></span>
</div>
```

Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `Movie.ReleaseDate` property. The `data-val-required` attribute contains an error message to display if the user doesn't fill in the release date field. jQuery Unobtrusive Validation passes this value to the jQuery Validation required() method, which then displays that message in the accompanying `<span>` element.

Data type validation is based on the .NET type of a property, unless that is overridden by a `[DataType]` attribute. Browsers have their own default error messages, but the jQuery Validation Unobtrusive Validation package can override those messages. `[DataType]` attributes and subclasses such as `[EmailAddress]` let you specify the error message.

Unobtrusive validation

For information on unobtrusive validation, see this GitHub issue.

Add Validation to Dynamic Forms

jQuery Unobtrusive Validation passes validation logic and parameters to jQuery Validation when the page first loads. Therefore, validation doesn't work automatically on dynamically generated forms. To enable validation, tell jQuery Unobtrusive Validation to parse the dynamic form immediately after you create it. For example, the following code sets up client-side validation on a form added via AJAX.

```
$.get({
    url: "https://url/that/returns/a/form",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add form. " + errorThrown);
    },
    success: function(newFormHTML) {
        var container = document.getElementById("form-container");
        container.insertAdjacentHTML("beforeend", newFormHTML);
        var forms = container.getElementsByTagName("form");
        var newForm = forms[forms.length - 1];
        $.validator.unobtrusive.parse(newForm);
    }
})
```

The `$.validator.unobtrusive.parse()` method accepts a jQuery selector for its one argument. This method tells jQuery Unobtrusive Validation to parse the `data-` attributes of forms within that selector. The values of those attributes are then passed to the jQuery Validation plugin.

Add Validation to Dynamic Controls

The `$.validator.unobtrusive.parse()` method works on an entire form, not on individual dynamically generated controls, such as `<input>` and `<select/>`. To reparse the form, remove the validation data that was added when the form was parsed earlier, as shown in the following example:

```
$.get({
    url: "https://url/that/returns/a/control",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add control. " + errorThrown);
    },
    success: function(newInputHTML) {
        var form = document.getElementById("my-form");
        form.insertAdjacentHTML("beforeend", newInputHTML);
        $(form).removeData("validator")    // Added by jQuery Validation
               .removeData("unobtrusiveValidation");   // Added by jQuery Unobtrusive
Validation
        $.validator.unobtrusive.parse(form);
    }
})
```

Custom client-side validation

Custom client-side validation is done by generating `data-` HTML attributes that work with a custom jQuery Validation adapter. The following sample adapter code was written for the `[ClassicMovie]` and `[ClassicMovieWithClientValidator]` attributes that were introduced earlier in this article:

```
$.validator.addMethod('classicmovie', function (value, element, params) {
    var genre = $(params[0]).val(), year = params[1], date = new Date(value);

    // The Classic genre has a value of '0'.
    if (genre && genre.length > 0 && genre[0] === '0') {
        // The release date for a Classic is valid if it's no greater than the given year.
        return date.getUTCFullYear() <= year;
    }

    return true;
});

$.validator.unobtrusive.adapters.add('classicmovie', ['year'], function (options) {
    var element = $(options.form).find('select#Movie_Genre')[0];

    options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
    options.messages['classicmovie'] = options.message;
});
```

For information about how to write adapters, see the jQuery Validation documentation.

The use of an adapter for a given field is triggered by `data-` attributes that:

- Flag the field as being subject to validation (`data-val="true"`).
- Identify a validation rule name and error message text (for example, `data-val-rulename="Error message."`).
- Provide any additional parameters the validator needs (for example, `data-val-rulename-param1="value"`).

The following example shows the `data-` attributes for the sample app's `ClassicMovie` attribute:

```
<input class="form-control" type="date"
    data-val="true"
    data-val-classicmovie="Classic movies must have a release year no later than 1960."
    data-val-classicmovie-year="1960"
    data-val-required="The Release Date field is required."
    id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
```

As noted earlier, Tag Helpers and HTML helpers use information from validation attributes to render `data-` attributes. There are two options for writing code that results in the creation of custom `data-` HTML attributes:

- Create a class that derives from `AttributeAdapterBase<TAttribute>` and a class that implements `IValidationAttributeAdapterProvider`, and register your attribute and its adapter in DI. This method follows the single responsibility principal in that server-related and client-related validation code is in separate classes. The adapter also has the advantage that since it is registered in DI, other services in DI are available to it if needed.
- Implement `IClientModelValidator` in your `ValidationAttribute` class. This method might be appropriate if the attribute doesn't do any server-side validation and doesn't need any services from DI.

AttributeAdapter for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovie` attribute in the sample app. To add client validation by using this method:

1. Create an attribute adapter class for the custom validation attribute. Derive the class from AttributeAdapterBase<T>. Create an `AddValidation` method that adds `data-` attributes to the rendered output, as shown in this example:

   ```
   public class ClassicMovieAttributeAdapter : AttributeAdapterBase<ClassicMovieAttribute>
   {
       public ClassicMovieAttributeAdapter(ClassicMovieAttribute attribute,
           IStringLocalizer stringLocalizer)
           : base(attribute, stringLocalizer)
       {

       }

       public override void AddValidation(ClientModelValidationContext context)
       {
           MergeAttribute(context.Attributes, "data-val", "true");
           MergeAttribute(context.Attributes, "data-val-classicmovie",
   GetErrorMessage(context));

           var year = Attribute.Year.ToString(CultureInfo.InvariantCulture);
           MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
       }

       public override string GetErrorMessage(ModelValidationContextBase validationContext)
   =>
           Attribute.GetErrorMessage();
   }
   ```

2. Create an adapter provider class that implements IValidationAttributeAdapterProvider. In the `GetAttributeAdapter` method pass in the custom attribute to the adapter's constructor, as shown in this example:

```
public class CustomValidationAttributeAdapterProvider :
IValidationAttributeAdapterProvider
{
    private readonly IValidationAttributeAdapterProvider baseProvider =
        new ValidationAttributeAdapterProvider();

    public IAttributeAdapter GetAttributeAdapter(ValidationAttribute attribute,
        IStringLocalizer stringLocalizer)
    {
        if (attribute is ClassicMovieAttribute classicMovieAttribute)
        {
            return new ClassicMovieAttributeAdapter(classicMovieAttribute,
stringLocalizer);
        }

        return baseProvider.GetAttributeAdapter(attribute, stringLocalizer);
    }
}
```

3. Register the adapter provider for DI in `Startup.ConfigureServices`:

```
services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

services.AddSingleton<IValidationAttributeAdapterProvider,
    CustomValidationAttributeAdapterProvider>();
```

IClientModelValidator for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovieWithClientValidator` attribute in the sample app. To add client validation by using this method:

- In the custom validation attribute, implement the `IClientModelValidator` interface and create an `AddValidation` method. In the `AddValidation` method, add `data-` attributes for validation, as shown in the following example:

```
public class ClassicMovieWithClientValidatorAttribute :
    ValidationAttribute, IClientModelValidator
{
    public ClassicMovieWithClientValidatorAttribute(int year)
    {
        Year = year;
    }

    public int Year { get; }

    public void AddValidation(ClientModelValidationContext context)
    {
        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMessage());

        var year = Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
    }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than {Year}.";

    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value).Year;

        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }

    private bool MergeAttribute(IDictionary<string, string> attributes, string key,
string value)
    {
        if (attributes.ContainsKey(key))
        {
            return false;
        }

        attributes.Add(key, value);
        return true;
    }
}
```

Disable client-side validation

The following code disables client validation in Razor Pages:

```
services.AddRazorPages()
    .AddViewOptions(options =>
    {
        options.HtmlHelperOptions.ClientValidationEnabled = false;
    });
```

Other options to disable client-side validation:

- Comment out the reference to `_ValidationScriptsPartial` in all the *.cshtml* files.
- Remove the contents of the *Pages\Shared_ValidationScriptsPartial.cshtml* file.

The preceding approach won't prevent client side validation of ASP.NET Core Identity Razor Class Library. For more information, see Scaffold Identity in ASP.NET Core projects.

Additional resources

- System.ComponentModel.DataAnnotations namespace
- Model Binding

This article explains how to validate user input in an ASP.NET Core MVC or Razor Pages app.

View or download sample code (how to download).

Model state

Model state represents errors that come from two subsystems: model binding and model validation. Errors that originate from model binding are generally data conversion errors (for example, an "x" is entered in a field that expects an integer). Model validation occurs after model binding and reports errors where the data doesn't conform to business rules (for example, a 0 is entered in a field that expects a rating between 1 and 5).

Both model binding and validation occur before the execution of a controller action or a Razor Pages handler method. For web apps, it's the app's responsibility to inspect `ModelState.IsValid` and react appropriately. Web apps typically redisplay the page with an error message:

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Web API controllers don't have to check `ModelState.IsValid` if they have the `[ApiController]` attribute. In that case, an automatic HTTP 400 response containing error details is returned when model state is invalid. For more information, see Automatic HTTP 400 responses.

Rerun validation

Validation is automatic, but you might want to repeat it manually. For example, you might compute a value for a property and want to rerun validation after setting the property to the computed value. To rerun validation, call the `TryValidateModel` method, as shown here:

```
var movie = new Movie
{
    Title = title,
    Genre = genre,
    ReleaseDate = modifiedReleaseDate,
    Description = description,
    Price = price,
    Preorder = preorder,
};

TryValidateModel(movie);

if (ModelState.IsValid)
{
    _context.AddMovie(movie);
    _context.SaveChanges();

    return RedirectToAction(actionName: nameof(Index));
}

return View(movie);
```

Validation attributes

Validation attributes let you specify validation rules for model properties. The following example from the sample app shows a model class that is annotated with validation attributes. The `[ClassicMovie]` attribute is a custom validation attribute and the others are built-in. Not shown is `[ClassicMovie2]`, which shows an alternative way to implement a custom attribute.

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    [Required]
    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

Built-in attributes

Built-in validation attributes include:

- `[CreditCard]`: Validates that the property has a credit card format.
- `[Compare]`: Validates that two properties in a model match. For example, the *Register.cshtml.cs*

file uses `[Compare]` to validate the two entered passwords match. Scaffold Identity to see the Register code.

- `[EmailAddress]`: Validates that the property has an email format.
- `[Phone]`: Validates that the property has a telephone number format.
- `[Range]`: Validates that the property value falls within a specified range.
- `[RegularExpression]`: Validates that the property value matches a specified regular expression.
- `[Required]`: Validates that the field is not null. See `[Required]` attribute for details about this attribute's behavior.
- `[StringLength]`: Validates that a string property value doesn't exceed a specified length limit.
- `[Url]`: Validates that the property has a URL format.
- `[Remote]`: Validates input on the client by calling an action method on the server. See `[Remote]` attribute for details about this attribute's behavior.

When using the `[RegularExpression]` attribute with client-side validation, the regex is executed in JavaScript on the client. This means ECMAScript matching behavior will be used. For more information, see this GitHub issue.

A complete list of validation attributes can be found in the System.ComponentModel.DataAnnotations namespace.

Error messages

Validation attributes let you specify the error message to be displayed for invalid input. For example:

```
[StringLength(8, ErrorMessage = "Name length can't be more than 8.")]
```

Internally, the attributes call `String.Format` with a placeholder for the field name and sometimes additional placeholders. For example:

```
[StringLength(8, ErrorMessage = "{0} length must be between {2} and {1}.", MinimumLength = 6)]
```

When applied to a `Name` property, the error message created by the preceding code would be "Name length must be between 6 and 8.".

To find out which parameters are passed to `String.Format` for a particular attribute's error message, see the DataAnnotations source code.

[Required] attribute

By default, the validation system treats non-nullable parameters or properties as if they had a `[Required]` attribute. Value types such as `decimal` and `int` are non-nullable.

[Required] validation on the server

On the server, a required value is considered missing if the property is null. A non-nullable field is always valid, and the [Required] attribute's error message is never displayed.

However, model binding for a non-nullable property may fail, resulting in an error message such as `The value '' is invalid`. To specify a custom error message for server-side validation of non-nullable types, you have the following options:

- Make the field nullable (for example, `decimal?` instead of `decimal`). Nullable<T> value types are treated like standard nullable types.

- Specify the default error message to be used by model binding, as shown in the following example:

```
services.AddMvc(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            (_) => "The field is required.");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
services.AddSingleton
    <IValidationAttributeAdapterProvider,
     CustomValidationAttributeAdapterProvider>();
```

  For more information about model binding errors that you can set default messages for, see DefaultModelBindingMessageProvider.

[Required] validation on the client

Non-nullable types and strings are handled differently on the client compared to the server. On the client:

- A value is considered present only if input is entered for it. Therefore, client-side validation handles non-nullable types the same as nullable types.
- Whitespace in a string field is considered valid input by the jQuery Validation required method. Server-side validation considers a required string field invalid if only whitespace is entered.

As noted earlier, non-nullable types are treated as though they had a `[Required]` attribute. That means you get client-side validation even if you don't apply the `[Required]` attribute. But if you don't use the attribute, you get a default error message. To specify a custom error message, use the attribute.

[Remote] attribute

The `[Remote]` attribute implements client-side validation that requires calling a method on the server to determine whether field input is valid. For example, the app may need to verify whether a user name is already in use.

To implement remote validation:

1. Create an action method for JavaScript to call. The jQuery Validate remote method expects a JSON response:
   - `"true"` means the input data is valid.
   - `"false"`, `undefined`, or `null` means the input is invalid. Display the default error message.
   - Any other string means the input is invalid. Display the string as a custom error message.

   Here's an example of an action method that returns a custom error message:

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyEmail(string email)
{
    if (!_userRepository.VerifyEmail(email))
    {
        return Json($"Email {email} is already in use.");
    }

    return Json(true);
}
```

2. In the model class, annotate the property with a `[Remote]` attribute that points to the validation action method, as shown in the following example:

```
[Remote(action: "VerifyEmail", controller: "Users")]
public string Email { get; set; }
```

The `[Remote]` attribute is in the `Microsoft.AspNetCore.Mvc` namespace. Install the Microsoft.AspNetCore.Mvc.ViewFeatures NuGet package if you're not using the `Microsoft.AspNetCore.App` or `Microsoft.AspNetCore.All` metapackage.

Additional fields

The `AdditionalFields` property of the `[Remote]` attribute lets you validate combinations of fields against data on the server. For example, if the `User` model had `FirstName` and `LastName` properties, you might want to verify that no existing users already have that pair of names. The following example shows how to use `AdditionalFields`:

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(LastName))]
public string FirstName { get; set; }
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName))]
public string LastName { get; set; }
```

`AdditionalFields` could be set explicitly to the strings `"FirstName"` and `"LastName"`, but using the nameof operator simplifies later refactoring. The action method for this validation must accept both first name and last name arguments:

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userRepository.VerifyName(firstName, lastName))
    {
        return Json($"A user named {firstName} {lastName} already exists.");
    }

    return Json(true);
}
```

When the user enters a first or last name, JavaScript makes a remote call to see if that pair of names has been taken.

To validate two or more additional fields, provide them as a comma-delimited list. For example, to add a `MiddleName` property to the model, set the `[Remote]` attribute as shown in the following example:

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields = nameof(FirstName) + "," +
nameof(LastName))]
public string MiddleName { get; set; }
```

`AdditionalFields`, like all attribute arguments, must be a constant expression. Therefore, don't use an interpolated string or call Join to initialize `AdditionalFields`.

Alternatives to built-in attributes

If you need validation not provided by built-in attributes, you can:

- Create custom attributes.
- Implement IValidatableObject.

Custom attributes

For scenarios that the built-in validation attributes don't handle, you can create custom validation attributes. Create a class that inherits from ValidationAttribute, and override the IsValid method.

The `IsValid` method accepts an object named *value*, which is the input to be validated. An overload also accepts a `ValidationContext` object, which provides additional information, such as the model instance created by model binding.

The following example validates that the release date for a movie in the *Classic* genre isn't later than a specified year. The `[ClassicMovie2]` attribute checks the genre first and continues only if it's *Classic*. For movies identified as classics, it checks the release date to make sure it's not later than the limit passed to the attribute constructor.)

```
public class ClassicMovieAttribute : ValidationAttribute
{
    private int _year;

    public ClassicMovieAttribute(int year)
    {
        _year = year;
    }

    protected override ValidationResult IsValid(
        object value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value).Year;

        if (movie.Genre == Genre.Classic && releaseYear > _year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }

    public int Year => _year;

    public string GetErrorMessage()
    {
        return $"Classic movies must have a release year no later than {_year}.";
    }
}
```

The `movie` variable in the preceding example represents a `Movie` object that contains the data from the form submission. The `IsValid` method checks the date and genre. Upon successful validation, `IsValid` returns a `ValidationResult.Success` code. When validation fails, a `ValidationResult` with an error message is returned.

IValidatableObject

The preceding example works only with `Movie` types. Another option for class-level validation is to implement `IValidatableObject` in the model class, as shown in the following example:

```
public class MovieIValidatable : IValidatableObject
{
    private const int _classicYear = 1960;

    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [Required]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    [Required]
    public Genre Genre { get; set; }

    public bool Preorder { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
        {
            yield return new ValidationResult(
                $"Classic movies must have a release year earlier than {_classicYear}.",
                new[] { "ReleaseDate" });
        }
    }
}
```

Top-level node validation

Top-level nodes include:

- Action parameters
- Controller properties
- Page handler parameters
- Page model properties

Model-bound top-level nodes are validated in addition to validating model properties. In the following example from the sample app, the `VerifyPhone` method uses the RegularExpressionAttribute to validate the `phone` action parameter:

```
[AcceptVerbs("Get", "Post")]
public IActionResult VerifyPhone(
    [RegularExpression(@"^\d{3}-\d{3}-\d{4}$")] string phone)
{
    if (!ModelState.IsValid)
    {
        return Json($"Phone {phone} has an invalid format. Format: ###-###-####");
    }

    return Json(true);
}
```

Top-level nodes can use BindRequiredAttribute with validation attributes. In the following example from the sample app, the `CheckAge` method specifies that the `age` parameter must be bound from the query string when the form is submitted:

```
[HttpPost]
public IActionResult CheckAge(
    [BindRequired, FromQuery] int age)
{
```

In the Check Age page (*CheckAge.cshtml*), there are two forms. The first form submits an `Age` value of `99` as a query string: `https://localhost:5001/Users/CheckAge?Age=99`.

When a properly formatted `age` parameter from the query string is submitted, the form validates.

The second form on the Check Age page submits the `Age` value in the body of the request, and validation fails. Binding fails because the `age` parameter must come from a query string.

When running with `CompatibilityVersion.Version_2_1` or later, top-level node validation is enabled by default. Otherwise, top-level node validation is disabled. The default option can be overridden by setting the AllowValidatingTopLevelNodes property in (`Startup.ConfigureServices`), as shown here:

```
services.AddMvc(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.AllowValidatingTopLevelNodes = false;
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Maximum errors

Validation stops when the maximum number of errors is reached (200 by default). You can configure this number with the following code in `Startup.ConfigureServices`:

```
services.AddMvc(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            (_) => "The field is required.");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
services.AddSingleton
    <IValidationAttributeAdapterProvider,
     CustomValidationAttributeAdapterProvider>();
```

Maximum recursion

ValidationVisitor traverses the object graph of the model being validated. For models that are very deep or are infinitely recursive, validation may result in stack overflow. MvcOptions.MaxValidationDepth provides a way to stop validation early if the visitor recursion exceeds a configured depth. The default value of `MvcOptions.MaxValidationDepth` is 32 when running with `CompatibilityVersion.Version_2_2` or later. For earlier versions, the value is null, which means no depth constraint.

Automatic short-circuit

Validation is automatically short-circuited (skipped) if the model graph doesn't require validation. Objects that the runtime skips validation for include collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`) and complex object graphs that don't have any validators.

Disable validation

To disable validation:

1. Create an implementation of `IObjectModelValidator` that doesn't mark any fields as invalid.

   ```
   public class NullObjectModelValidator : IObjectModelValidator
   {
       public void Validate(
           ActionContext actionContext,
           ValidationStateDictionary validationState,
           string prefix,
           object model)
       {
       }
   }
   ```

2. Add the following code to `Startup.ConfigureServices` to replace the default `IObjectModelValidator` implementation in the dependency injection container.

   ```
   // There is only one `IObjectModelValidator` object,
   // so AddSingleton replaces the default one.
   services.AddSingleton<IObjectModelValidator>(new NullObjectModelValidator());
   ```

You might still see model state errors that originate from model binding.

Client-side validation

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

Client-side validation avoids an unnecessary round trip to the server when there are input errors on a form. The following script references in *_Layout.cshtml* and *_ValidationScriptsPartial.cshtml* support client-side validation:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validate/1.17.0/jquery.validate.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-
unobtrusive/3.2.11/jquery.validate.unobtrusive.min.js"></script>
```

The jQuery Unobtrusive Validation script is a custom Microsoft front-end library that builds on the popular jQuery Validate plugin. Without jQuery Unobtrusive Validation, you would have to code the same validation logic in two places: once in the server-side validation attributes on model properties, and then again in client-side scripts. Instead, Tag Helpers and HTML helpers use the validation attributes and type metadata from model properties to render HTML 5 `data-` attributes for the form elements that need validation. jQuery Unobtrusive Validation parses the `data-` attributes and passes the logic to jQuery Validate, effectively "copying" the server-side validation logic to the client. You can display validation errors on the client using tag helpers as shown here:

```
<div class="form-group">
    <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="ReleaseDate" class="form-control" />
        <span asp-validation-for="ReleaseDate" class="text-danger"></span>
    </div>
</div>
```

The preceding tag helpers render the following HTML.

```
<form action="/Movies/Create" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <div class="text-danger"></div>
        <div class="form-group">
            <label class="col-md-2 control-label" for="ReleaseDate">ReleaseDate</label>
            <div class="col-md-10">
                <input class="form-control" type="datetime"
                data-val="true" data-val-required="The ReleaseDate field is required."
                id="ReleaseDate" name="ReleaseDate" value="">
                <span class="text-danger field-validation-valid"
                data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>
            </div>
        </div>
    </div>
</form>
```

Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `ReleaseDate` property. The `data-val-required` attribute contains an error message to display if the user doesn't fill in the release date field. jQuery Unobtrusive Validation passes this value to the jQuery Validate required() method, which then displays that message in the accompanying `<span>` element.

Data type validation is based on the .NET type of a property, unless that is overridden by a `[DataType]` attribute. Browsers have their own default error messages, but the jQuery Validation Unobtrusive Validation package can override those messages. `[DataType]` attributes and subclasses such as `[EmailAddress]` let you specify the error message.

Add Validation to Dynamic Forms

jQuery Unobtrusive Validation passes validation logic and parameters to jQuery Validate when the page first loads. Therefore, validation doesn't work automatically on dynamically generated forms. To enable validation, tell jQuery Unobtrusive Validation to parse the dynamic form immediately

after you create it. For example, the following code sets up client-side validation on a form added via AJAX.

```
$.get({
    url: "https://url/that/returns/a/form",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add form. " + errorThrown);
    },
    success: function(newFormHTML) {
        var container = document.getElementById("form-container");
        container.insertAdjacentHTML("beforeend", newFormHTML);
        var forms = container.getElementsByTagName("form");
        var newForm = forms[forms.length - 1];
        $.validator.unobtrusive.parse(newForm);
    }
})
```

The `$.validator.unobtrusive.parse()` method accepts a jQuery selector for its one argument. This method tells jQuery Unobtrusive Validation to parse the `data-` attributes of forms within that selector. The values of those attributes are then passed to the jQuery Validate plugin.

Add Validation to Dynamic Controls

The `$.validator.unobtrusive.parse()` method works on an entire form, not on individual dynamically generated controls, such as `<input>` and `<select/>`. To reparse the form, remove the validation data that was added when the form was parsed earlier, as shown in the following example:

```
$.get({
    url: "https://url/that/returns/a/control",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add control. " + errorThrown);
    },
    success: function(newInputHTML) {
        var form = document.getElementById("my-form");
        form.insertAdjacentHTML("beforeend", newInputHTML);
        $(form).removeData("validator")    // Added by jQuery Validate
               .removeData("unobtrusiveValidation");   // Added by jQuery Unobtrusive
Validation
        $.validator.unobtrusive.parse(form);
    }
})
```

Custom client-side validation

Custom client-side validation is done by generating `data-` HTML attributes that work with a custom jQuery Validate adapter. The following sample adapter code was written for the `ClassicMovie` and `ClassicMovie2` attributes that were introduced earlier in this article:

```
$.validator.addMethod('classicmovie',
    function (value, element, params) {
        // Get element value. Classic genre has value '0'.
        var genre = $(params[0]).val(),
            year = params[1],
            date = new Date(value);
        if (genre && genre.length > 0 && genre[0] === '0') {
            // Since this is a classic movie, invalid if release date is after given year.
            return date.getUTCFullYear() <= year;
        }

        return true;
    });

$.validator.unobtrusive.adapters.add('classicmovie',
    ['year'],
    function (options) {
        var element = $(options.form).find('select#Genre')[0];
        options.rules['classicmovie'] = [element, parseInt(options.params['year'])];
        options.messages['classicmovie'] = options.message;
    });
```

For information about how to write adapters, see the jQuery Validate documentation.

The use of an adapter for a given field is triggered by `data-` attributes that:

- Flag the field as being subject to validation (`data-val="true"`).
- Identify a validation rule name and error message text (for example, `data-val-rulename="Error message."`).
- Provide any additional parameters the validator needs (for example, `data-val-rulename-parm1="value"`).

The following example shows the `data-` attributes for the sample app's `ClassicMovie` attribute:

```
<input class="form-control" type="datetime"
    data-val="true"
    data-val-classicmovie1="Classic movies must have a release year earlier than 1960."
    data-val-classicmovie1-year="1960"
    data-val-required="The ReleaseDate field is required."
    id="ReleaseDate" name="ReleaseDate" value="">
```

As noted earlier, Tag Helpers and HTML helpers use information from validation attributes to render `data-` attributes. There are two options for writing code that results in the creation of custom `data-` HTML attributes:

- Create a class that derives from `AttributeAdapterBase<TAttribute>` and a class that implements `IValidationAttributeAdapterProvider` , and register your attribute and its adapter in DI. This method follows the single responsibility principal in that server-related and client-related validation code is in separate classes. The adapter also has the advantage that since it is registered in DI, other services in DI are available to it if needed.
- Implement `IClientModelValidator` in your `ValidationAttribute` class. This method might be appropriate if the attribute doesn't do any server-side validation and doesn't need any services from DI.

AttributeAdapter for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovie` attribute in the sample app. To add client validation by using this method:

1. Create an attribute adapter class for the custom validation attribute. Derive the class from AttributeAdapterBase<T>. Create an `AddValidation` method that adds `data-` attributes to the rendered output, as shown in this example:

```
public class ClassicMovieAttributeAdapter : AttributeAdapterBase<ClassicMovieAttribute>
{
    private int _year;

    public ClassicMovieAttributeAdapter(ClassicMovieAttribute attribute,
IStringLocalizer stringLocalizer) : base (attribute, stringLocalizer)
    {
        _year = attribute.Year;
    }
    public override void AddValidation(ClientModelValidationContext context)
    {
        if (context == null)
        {
            throw new ArgumentNullException(nameof(context));
        }

        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie",
GetErrorMessage(context));

        var year = Attribute.Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
    }
    public override string GetErrorMessage(ModelValidationContextBase validationContext)
    {
        return Attribute.GetErrorMessage();
    }
}
```

2. Create an adapter provider class that implements IValidationAttributeAdapterProvider. In the `GetAttributeAdapter` method pass in the custom attribute to the adapter's constructor, as shown in this example:

```
public class CustomValidationAttributeAdapterProvider :
    IValidationAttributeAdapterProvider
{
    IValidationAttributeAdapterProvider baseProvider =
        new ValidationAttributeAdapterProvider();
    public IAttributeAdapter GetAttributeAdapter(ValidationAttribute attribute,
        IStringLocalizer stringLocalizer)
    {
        if (attribute is ClassicMovieAttribute classicMovieAttribute)
        {
            return new ClassicMovieAttributeAdapter(classicMovieAttribute,
stringLocalizer);
        }
        else
        {
            return baseProvider.GetAttributeAdapter(attribute, stringLocalizer);
        }
    }
}
```

3. Register the adapter provider for DI in `Startup.ConfigureServices`:

```
services.AddMvc(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            (_) => "The field is required.");
    })
    .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
services.AddSingleton
    <IValidationAttributeAdapterProvider,
     CustomValidationAttributeAdapterProvider>();
```

IClientModelValidator for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovie2` attribute in the sample app. To add client validation by using this method:

- In the custom validation attribute, implement the `IClientModelValidator` interface and create an `AddValidation` method. In the `AddValidation` method, add `data-` attributes for validation, as shown in the following example:

```
public class ClassicMovie2Attribute : ValidationAttribute, IClientModelValidator
{
    private int _year;

    public ClassicMovie2Attribute(int year)
    {
        _year = year;
    }

    protected override ValidationResult IsValid(
        object value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value).Year;

        if (movie.Genre == Genre.Classic && releaseYear > _year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }

    public void AddValidation(ClientModelValidationContext context)
    {
        if (context == null)
        {
            throw new ArgumentNullException(nameof(context));
        }

        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie", GetErrorMessage());

        var year = _year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-year", year);
    }
    private bool MergeAttribute(IDictionary<string, string> attributes, string key,
string value)
    {
        if (attributes.ContainsKey(key))
        {
            return false;
        }

        attributes.Add(key, value);
        return true;
    }
    protected string GetErrorMessage()
    {
        return $"Classic movies must have a release year no later than {_year} [from
attribute 2].";
    }
}
```

Disable client-side validation

The following code disables client validation in MVC views:

```
services.AddMvc().AddViewOptions(options =>
{
    if (_env.IsDevelopment())
    {
        options.HtmlHelperOptions.ClientValidationEnabled = false;
    }
});
```

And in Razor Pages:

```
services.Configure<HtmlHelperOptions>(o => o.ClientValidationEnabled = false);
```

Another option for disabling client validation is to comment out the reference to `_ValidationScriptsPartial` in your *.cshtml* file.

Additional resources

- System.ComponentModel.DataAnnotations namespace
- Model Binding

Compatibility version for ASP.NET Core MVC

9/22/2020 • 2 minutes to read • Edit Online

By Rick Anderson

The SetCompatibilityVersion method is a no-op for ASP.NET Core 3.0 apps. That is, calling `SetCompatibilityVersion` with any value of CompatibilityVersion has no impact on the application.

- The next minor version of ASP.NET Core may provide a new `CompatibilityVersion` value.
- `CompatibilityVersion` values `Version_2_0` through `Version_2_2` are marked `[Obsolete(...)]`.
- See Breaking API changes in Antiforgery, CORS, Diagnostics, Mvc, and Routing. This list includes breaking changes for compatibility switches.

To see how `SetCompatibilityVersion` works with ASP.NET Core 2.x apps, select the .

The SetCompatibilityVersion method allows an ASP.NET Core 2.x app to opt-in or opt-out of potentially breaking behavior changes introduced in ASP.NET Core MVC 2.1 or 2.2. These potentially breaking behavior changes are generally in how the MVC subsystem behaves and how **your code** is called by the runtime. By opting in, you get the latest behavior, and the long-term behavior of ASP.NET Core.

The following code sets the compatibility mode to ASP.NET Core 2.2:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

We recommend you test your app using the latest version (`CompatibilityVersion.Latest`). We anticipate that most apps won't have breaking behavior changes using the latest version.

Apps that call `SetCompatibilityVersion(CompatibilityVersion.Version_2_0)` are protected from potentially breaking behavior changes introduced in the ASP.NET Core 2.1/2.2 MVC versions. This protection:

- Does not apply to all 2.1 and later changes, it's targeted to potentially breaking ASP.NET Core runtime behavior changes in the MVC subsystem.
- Does not extend to ASP.NET Core 3.0.

The default compatibility for ASP.NET Core 2.1 and 2.2 apps that do **not** call `SetCompatibilityVersion` is 2.0 compatibility. That is, not calling `SetCompatibilityVersion` is the same as calling `SetCompatibilityVersion(CompatibilityVersion.Version_2_0)`.

The following code sets the compatibility mode to ASP.NET Core 2.2, except for the following behaviors:

- AllowCombiningAuthorizeFilters
- InputFormatterExceptionPolicy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        // Include the 2.2 behaviors
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
        // Except for the following.
        .AddMvcOptions(options =>
        {
            // Don't combine authorize filters (keep 2.0 behavior).
            options.AllowCombiningAuthorizeFilters = false;
            // All exceptions thrown by an IInputFormatter are treated
            // as model state errors (keep 2.0 behavior).
            options.InputFormatterExceptionPolicy =
                InputFormatterExceptionPolicy.AllExceptions;
        });
}
```

For apps that encounter breaking behavior changes, using the appropriate compatibility switches:

- Allows you to use the latest release and opt out of specific breaking behavior changes.
- Gives you time to update your app so it works with the latest changes.

The MvcOptions documentation has a good explanation of what changed and why the changes are an improvement for most users.

With ASP.NET Core 3.0, old behaviors supported by compatibility switches have been removed. We feel these are positive changes benefitting nearly all users. By introducing these changes in 2.1 and 2.2, most apps can benefit, while others have time to update.

Write custom ASP.NET Core middleware

9/22/2020 • 3 minutes to read • Edit Online

By Rick Anderson and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. ASP.NET Core provides a rich set of built-in middleware components, but in some scenarios you might want to write a custom middleware.

> **NOTE**
> This topic describes how to write *convention-based* middleware. For an approach that uses strong typing and per-request activation, see Factory-based middleware activation in ASP.NET Core.

Middleware class

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following middleware, which sets the culture for the current request from a query string:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;
            }

            // Call the next delegate/middleware in the pipeline
            await next();
        });

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });

    }
}
```

The preceding sample code is used to demonstrate creating a middleware component. For ASP.NET Core's built-in localization support, see Globalization and localization in ASP.NET Core.

Test the middleware by passing in the culture. For example, request `https://localhost:5001/?culture=no`.

The following code moves the middleware delegate to a class:

```
using Microsoft.AspNetCore.Http;
using System.Globalization;
using System.Threading.Tasks;

namespace Culture
{
    public class RequestCultureMiddleware
    {
        private readonly RequestDelegate _next;

        public RequestCultureMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task InvokeAsync(HttpContext context)
        {
            var cultureQuery = context.Request.Query["culture"];
            if (!string.IsNullOrWhiteSpace(cultureQuery))
            {
                var culture = new CultureInfo(cultureQuery);

                CultureInfo.CurrentCulture = culture;
                CultureInfo.CurrentUICulture = culture;

            }

            // Call the next delegate/middleware in the pipeline
            await _next(context);
        }
    }
}
```

The middleware class must include:

- A public constructor with a parameter of type RequestDelegate.
- A public method named `Invoke` or `InvokeAsync`. This method must:
  - Return a `Task`.
  - Accept a first parameter of type HttpContext.

Additional parameters for the constructor and `Invoke` / `InvokeAsync` are populated by dependency injection (DI).

Middleware dependencies

Middleware should follow the Explicit Dependencies Principle by exposing its dependencies in its constructor. Middleware is constructed once per *application lifetime*. See the Per-request middleware dependencies section if you need to share services with middleware within a request.

Middleware components can resolve their dependencies from dependency injection (DI) through constructor parameters. UseMiddleware<T> can also accept additional parameters directly.

Per-request middleware dependencies

Because middleware is constructed at app startup, not per-request, *scoped* lifetime services used by middleware constructors aren't shared with other dependency-injected types during each request. If you must share a *scoped* service between your middleware and other types, add these services to the `Invoke` method's signature. The `Invoke` method can accept additional parameters that are populated by DI:

```
public class CustomMiddleware
{
    private readonly RequestDelegate _next;

    public CustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    // IMyScopedService is injected into Invoke
    public async Task Invoke(HttpContext httpContext, IMyScopedService svc)
    {
        svc.MyProperty = 1000;
        await _next(httpContext);
    }
}
```

Lifetime and registration options contains a complete sample of middleware with *scoped* lifetime services.

Middleware extension method

The following extension method exposes the middleware through IApplicationBuilder:

```
using Microsoft.AspNetCore.Builder;

namespace Culture
{
    public static class RequestCultureMiddlewareExtensions
    {
        public static IApplicationBuilder UseRequestCulture(
            this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<RequestCultureMiddleware>();
        }
    }
}
```

The following code calls the middleware from `Startup.Configure`:

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRequestCulture();

        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(
                $"Hello {CultureInfo.CurrentCulture.DisplayName}");
        });
    }
}
```

Additional resources

- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- ASP.NET Core Middleware
- Test ASP.NET Core middleware

- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

Request and response operations in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

By Justin Kotalik

This article explains how to read from the request body and write to the response body. Code for these operations might be required when writing middleware. Outside of writing middleware, custom code isn't generally required because the operations are handled by MVC and Razor Pages.

There are two abstractions for the request and response bodies: Stream and Pipe. For request reading, HttpRequest.Body is a Stream, and `HttpRequest.BodyReader` is a PipeReader. For response writing, HttpResponse.Body is a Stream, and `HttpResponse.BodyWriter` is a PipeWriter.

Pipelines are recommended over streams. Streams can be easier to use for some simple operations, but pipelines have a performance advantage and are easier to use in most scenarios. ASP.NET Core is starting to use pipelines instead of streams internally. Examples include:

- `FormReader`
- `TextReader`
- `TextWriter`
- `HttpResponse.WriteAsync`

Streams aren't being removed from the framework. Streams continue to be used throughout .NET, and many stream types don't have pipe equivalents, such as `FileStreams` and `ResponseCompression`.

Stream examples

Suppose the goal is to create a middleware that reads the entire request body as a list of strings, splitting on new lines. A simple stream implementation might look like the following example:

> **WARNING**
>
> The following code:
>
> - Is used to demonstrate the problems with not using a pipe to read the request body.
> - Is not intended to be used in production apps.

```
private async Task<List<string>> GetListOfStringsFromStream(Stream requestBody)
{
    // Build up the request body in a string builder.
    StringBuilder builder = new StringBuilder();

    // Rent a shared buffer to write the request body into.
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0, buffer.Length);
        if (bytesRemaining == 0)
        {
            break;
        }

        // Append the encoded string into the string builder.
        var encodedString = Encoding.UTF8.GetString(buffer, 0, bytesRemaining);
        builder.Append(encodedString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    var entireRequestBody = builder.ToString();

    // Split on \n in the string.
    return new List<string>(entireRequestBody.Split("\n"));
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

This code works, but there are some issues:

- Before appending to the `StringBuilder`, the example creates another string (`encodedString`) that is thrown away immediately. This process occurs for all bytes in the stream, so the result is extra memory allocation the size of the entire request body.
- The example reads the entire string before splitting on new lines. It's more efficient to check for new lines in the byte array.

Here's an example that fixes some of the preceding issues:

> **WARNING**
>
> The following code:
>
> - Is used to demonstrate the solutions to some problems in the preceding code while not solving all the problems.
> - Is not intended to be used in production apps.

```
private async Task<List<string>> GetListOfStringsFromStreamMoreEfficient(Stream requestBody)
{
    StringBuilder builder = new StringBuilder();
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);
    List<string> results = new List<string>();

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0, buffer.Length);

        if (bytesRemaining == 0)
        {
            results.Add(builder.ToString());
            break;
        }

        // Instead of adding the entire buffer into the StringBuilder
        // only add the remainder after the last \n in the array.
        var prevIndex = 0;
        int index;
        while (true)
        {
            index = Array.IndexOf(buffer, (byte)'\n', prevIndex);
            if (index == -1)
            {
                break;
            }

            var encodedString = Encoding.UTF8.GetString(buffer, prevIndex, index - prevIndex);

            if (builder.Length > 0)
            {
                // If there was a remainder in the string buffer, include it in the next string.
                results.Add(builder.Append(encodedString).ToString());
                builder.Clear();
            }
            else
            {
                results.Add(encodedString);
            }

            // Skip past last \n
            prevIndex = index + 1;
        }

        var remainingString = Encoding.UTF8.GetString(buffer, prevIndex, bytesRemaining - prevIndex);
        builder.Append(remainingString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    return results;
}
```

This preceding example:

- Doesn't buffer the entire request body in a `StringBuilder` unless there aren't any newline characters.
- Doesn't call `Split` on the string.

However, there are still are a few issues:

- If newline characters are sparse, much of the request body is buffered in the string.
- The code continues to create strings (`remainingString`) and adds them to the string buffer, which results in an extra allocation.

These issues are fixable, but the code is becoming progressively more complicated with little improvement. Pipelines provide a way to solve these problems with minimal code complexity.

Pipelines

The following example shows how the same scenario can be handled using a PipeReader:

```
private async Task<List<string>> GetListOfStringFromPipe(PipeReader reader)
{
    List<string> results = new List<string>();

    while (true)
    {
        ReadResult readResult = await reader.ReadAsync();
        var buffer = readResult.Buffer;

        SequencePosition? position = null;

        do
        {
            // Look for a EOL in the buffer
            position = buffer.PositionOf((byte)'\n');

            if (position != null)
            {
                var readOnlySequence = buffer.Slice(0, position.Value);
                AddStringToList(ref results, in readOnlySequence);

                // Skip the line + the \n character (basically position)
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
        }
        while (position != null);


        if (readResult.IsCompleted && buffer.Length > 0)
        {
            AddStringToList(ref results, in buffer);
        }

        reader.AdvanceTo(buffer.Start, buffer.End);

        // At this point, buffer will be updated to point one byte after the last
        // \n character.
        if (readResult.IsCompleted)
        {
            break;
        }
    }

    return results;
}

private static void AddStringToList(ref List<string> results, in ReadOnlySequence<byte> readOnlySequence)
{
    // Separate method because Span/ReadOnlySpan cannot be used in async methods
    ReadOnlySpan<byte> span = readOnlySequence.IsSingleSegment ? readOnlySequence.First.Span :
readOnlySequence.ToArray().AsSpan();
    results.Add(Encoding.UTF8.GetString(span));
}
```

This example fixes many issues that the streams implementations had:

- There's no need for a string buffer because the `PipeReader` handles bytes that haven't been used.

- Encoded strings are directly added to the list of returned strings.
- Other than the `ToArray` call, and the memory used by the string, string creation is allocation free.

Adapters

The `Body`, `BodyReader`, and `BodyWriter` properties are available for `HttpRequest` and `HttpResponse`. When you set `Body` to a different stream, a new set of adapters automatically adapt each type to the other. If you set `HttpRequest.Body` to a new stream, `HttpRequest.BodyReader` is automatically set to a new `PipeReader` that wraps `HttpRequest.Body`.

StartAsync

`HttpResponse.StartAsync` is used to indicate that headers are unmodifiable and to run `OnStarting` callbacks. When using Kestrel as a server, calling `StartAsync` before using the `PipeReader` guarantees that memory returned by `GetMemory` belongs to Kestrel's internal Pipe rather than an external buffer.

Additional resources

- System.IO.Pipelines in .NET
- Write custom ASP.NET Core middleware

URL Rewriting Middleware in ASP.NET Core

9/22/2020 • 32 minutes to read • Edit Online

By Mikael Mengistu

This document introduces URL rewriting with instructions on how to use URL Rewriting Middleware in ASP.NET Core apps.

URL rewriting is the act of modifying request URLs based on one or more predefined rules. URL rewriting creates an abstraction between resource locations and their addresses so that the locations and addresses aren't tightly linked. URL rewriting is valuable in several scenarios to:

- Move or replace server resources temporarily or permanently and maintain stable locators for those resources.
- Split request processing across different apps or across areas of one app.
- Remove, add, or reorganize URL segments on incoming requests.
- Optimize public URLs for Search Engine Optimization (SEO).
- Permit the use of friendly public URLs to help visitors predict the content returned by requesting a resource.
- Redirect insecure requests to secure endpoints.
- Prevent hotlinking, where an external site uses a hosted static asset on another site by linking the asset into its own content.

> **NOTE**
> URL rewriting can reduce the performance of an app. Where feasible, limit the number and complexity of rules.

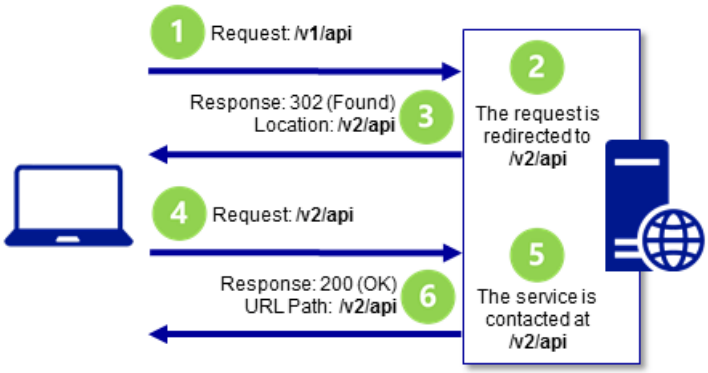
View or download sample code (how to download)

URL redirect and URL rewrite

The difference in wording between *URL redirect* and *URL rewrite* is subtle but has important implications for providing resources to clients. ASP.NET Core's URL Rewriting Middleware is capable of meeting the need for both.

A *URL redirect* involves a client-side operation, where the client is instructed to access a resource at a different address than the client originally requested. This requires a round trip to the server. The redirect URL returned to the client appears in the browser's address bar when the client makes a new request for the resource.

If `/resource` is *redirected* to `/different-resource`, the server responds that the client should obtain the resource at `/different-resource` with a status code indicating that the redirect is either temporary or permanent.



When redirecting requests to a different URL, indicate whether the redirect is permanent or temporary by specifying the status code with the response:

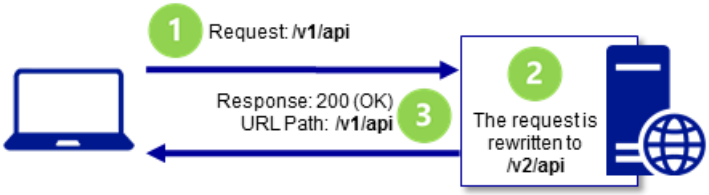
- The *301 - Moved Permanently* status code is used where the resource has a new, permanent URL and you wish to instruct the client that all future requests for the resource should use the new URL. *The client may cache and reuse the response when a 301 status code is received.*
- The *302 - Found* status code is used where the redirection is temporary or generally subject to change. The 302 status code indicates to the client not to store the URL and use it in the future.

For more information on status codes, see RFC 2616: Status Code Definitions.

A *URL rewrite* is a server-side operation that provides a resource from a different resource address than the client requested. Rewriting a URL doesn't require a round trip to the server. The rewritten URL isn't returned to the client and doesn't appear in the browser's address bar.

If `/resource` is *rewritten* to `/different-resource`, the server *internally* fetches and returns the resource at `/different-resource`.

Although the client might be able to retrieve the resource at the rewritten URL, the client isn't informed that the resource exists at the rewritten URL when it makes its request and receives the response.



URL rewriting sample app

You can explore the features of the URL Rewriting Middleware with the sample app. The app applies redirect and rewrite rules and shows the redirected or rewritten URL for several scenarios.

When to use URL Rewriting Middleware

Use URL Rewriting Middleware when you're unable to use the following approaches:

- URL Rewrite module with IIS on Windows Server
- Apache mod_rewrite module on Apache Server
- URL rewriting on Nginx

Also, use the middleware when the app is hosted on HTTP.sys server (formerly called WebListener).

The main reasons to use the server-based URL rewriting technologies in IIS, Apache, and Nginx are:

- The middleware doesn't support the full features of these modules.

  Some of the features of the server modules don't work with ASP.NET Core projects, such as the `IsFile` and `IsDirectory` constraints of the IIS Rewrite module. In these scenarios, use the middleware instead.

- The performance of the middleware probably doesn't match that of the modules.

  Benchmarking is the only way to know for sure which approach degrades performance the most or if degraded performance is negligible.

Package

URL Rewriting Middleware is provided by the Microsoft.AspNetCore.Rewrite package, which is implicitly included

in ASP.NET Core apps.

Extension and options

Establish URL rewrite and redirect rules by creating an instance of the RewriteOptions class with extension methods for each of your rewrite rules. Chain multiple rules in the order that you would like them processed. The `RewriteOptions` are passed into the URL Rewriting Middleware as it's added to the request pipeline with UseRewriter:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

Redirect non-www to www

Three options permit the app to redirect non-`www` requests to `www`:

- AddRedirectToWwwPermanent: Permanently redirect the request to the `www` subdomain if the request is non-`www`. Redirects with a Status308PermanentRedirect status code.
- AddRedirectToWww: Redirect the request to the `www` subdomain if the incoming request is non-`www`. Redirects with a Status307TemporaryRedirect status code. An overload permits you to provide the status code for the response. Use a field of the StatusCodes class for a status code assignment.

URL redirect

Use AddRedirect to redirect requests. The first parameter contains your regex for matching on the path of the incoming URL. The second parameter is the replacement string. The third parameter, if present, specifies the status code. If you don't specify the status code, the status code defaults to *302 - Found*, which indicates that the resource is temporarily moved or replaced.

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

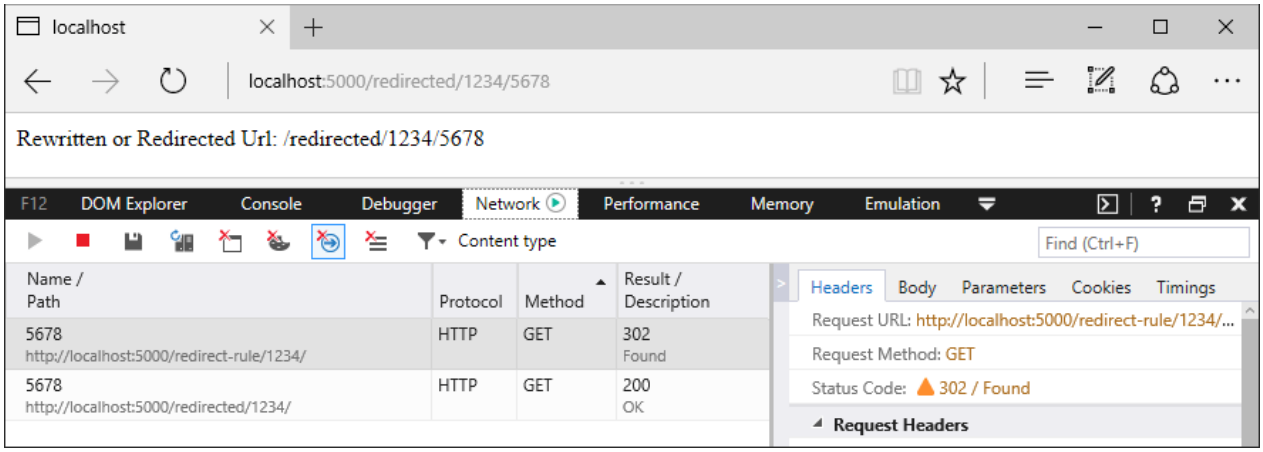
In a browser with developer tools enabled, make a request to the sample app with the path `/redirect-rule/1234/5678`. The regex matches the request path on `redirect-rule/(.*)`, and the path is replaced with `/redirected/1234/5678`. The redirect URL is sent back to the client with a *302 - Found* status code. The browser makes a new request at the redirect URL, which appears in the browser's address bar. Since no rules in the sample app match on the redirect URL:

- The second request receives a *200 - OK* response from the app.
- The body of the response shows the redirect URL.

A round trip is made to the server when a URL is *redirected*.

> **WARNING**
>
> Be cautious when establishing redirect rules. Redirect rules are evaluated on every request to the app, including after a redirect. It's easy to accidentally create a *loop of infinite redirects*.

Original Request: `/redirect-rule/1234/5678`



The part of the expression contained within parentheses is called a *capture group*. The dot (`.`) of the expression means *match any character*. The asterisk (`*`) indicates *match the preceding character zero or more times*. Therefore, the last two path segments of the URL, `1234/5678` , are captured by capture group `(.*)` . Any value you provide in the request URL after `redirect-rule/` is captured by this single capture group.

In the replacement string, captured groups are injected into the string with the dollar sign (`$`) followed by the sequence number of the capture. The first capture group value is obtained with `$1` , the second with `$2` , and they continue in sequence for the capture groups in your regex. There's only one captured group in the redirect rule regex in the sample app, so there's only one injected group in the replacement string, which is `$1` . When the rule is applied, the URL becomes `/redirected/1234/5678` .

URL redirect to a secure endpoint

Use AddRedirectToHttps to redirect HTTP requests to the same host and path using the HTTPS protocol. If the status code isn't supplied, the middleware defaults to *302 - Found*. If the port isn't supplied:

- The middleware defaults to `null` .
- The scheme changes to `https` (HTTPS protocol), and the client accesses the resource on port 443.

The following example shows how to set the status code to *301 - Moved Permanently* and change the port to 5001.

```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttps(301, 5001);

    app.UseRewriter(options);
}
```

Use AddRedirectToHttpsPermanent to redirect insecure requests to the same host and path with secure HTTPS protocol on port 443. The middleware sets the status code to *301 - Moved Permanently*.

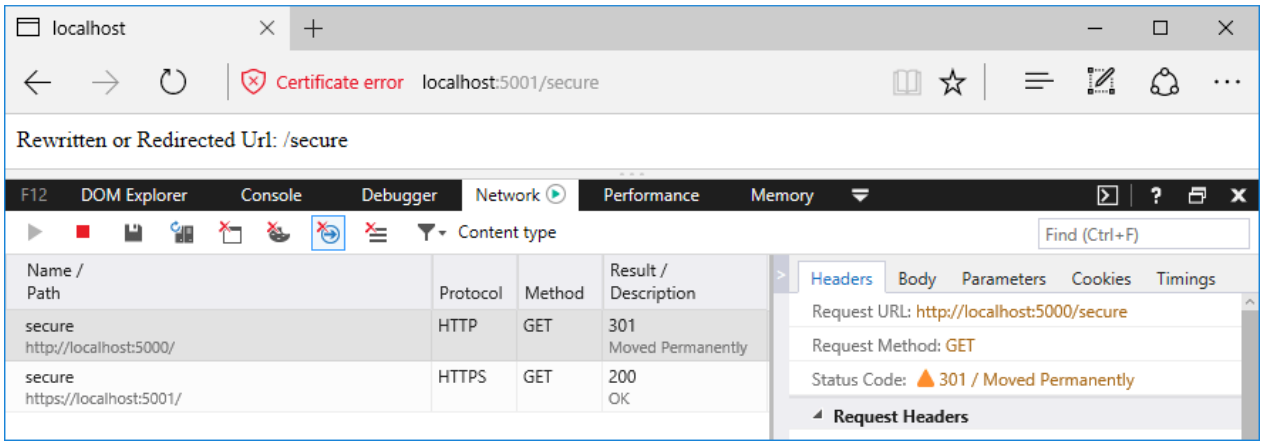
```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent();

    app.UseRewriter(options);
}
```

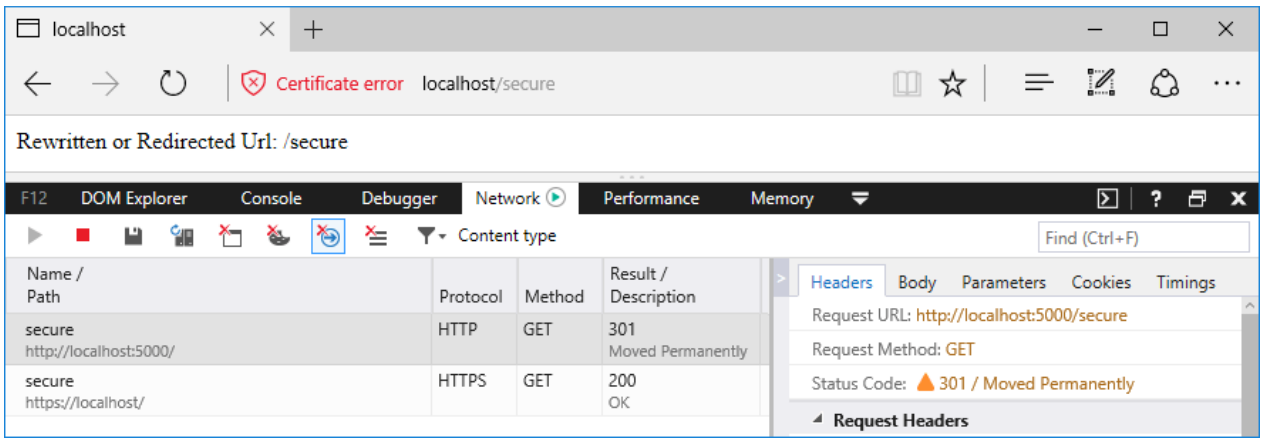
> **NOTE**
> When redirecting to a secure endpoint without the requirement for additional redirect rules, we recommend using HTTPS Redirection Middleware. For more information, see the Enforce HTTPS topic.

The sample app is capable of demonstrating how to use `AddRedirectToHttps` or `AddRedirectToHttpsPermanent` . Add the extension method to the `RewriteOptions` . Make an insecure request to the app at any URL. Dismiss the browser security warning that the self-signed certificate is untrusted or create an exception to trust the certificate.

Original Request using `AddRedirectToHttps(301, 5001)` : `http://localhost:5000/secure`



Original Request using `AddRedirectToHttpsPermanent` : `http://localhost:5000/secure`



URL rewrite

Use AddRewrite to create a rule for rewriting URLs. The first parameter contains the regex for matching on the incoming URL path. The second parameter is the replacement string. The third parameter, `skipRemainingRules: {true|false}`, indicates to the middleware whether or not to skip additional rewrite rules if the current rule is applied.

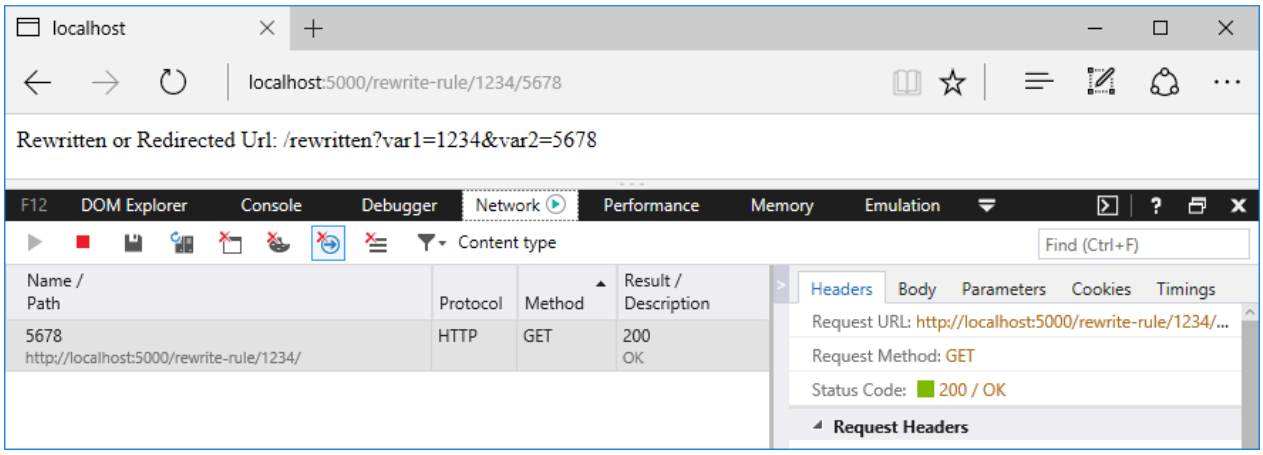
```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

Original Request: `/rewrite-rule/1234/5678`



The carat (`^`) at the beginning of the expression means that matching starts at the beginning of the URL path.

In the earlier example with the redirect rule, `redirect-rule/(.*)`, there's no carat (`^`) at the start of the regex. Therefore, any characters may precede `redirect-rule/` in the path for a successful match.

PATH	MATCH
`/redirect-rule/1234/5678`	Yes
`/my-cool-redirect-rule/1234/5678`	Yes
`/anotherredirect-rule/1234/5678`	Yes

The rewrite rule, `^rewrite-rule/(\d+)/(\d+)`, only matches paths if they start with `rewrite-rule/`. In the following table, note the difference in matching.

PATH	MATCH
`/rewrite-rule/1234/5678`	Yes
`/my-cool-rewrite-rule/1234/5678`	No
`/anotherrewrite-rule/1234/5678`	No

Following the `^rewrite-rule/` portion of the expression, there are two capture groups, `(\d+)/(\d+)`. The `\d` signifies *match a digit (number)*. The plus sign (`+`) means *match one or more of the preceding character*. Therefore, the URL must contain a number followed by a forward-slash followed by another number. These capture groups are injected into the rewritten URL as `$1` and `$2`. The rewrite rule replacement string places the captured groups into the query string. The requested path of `/rewrite-rule/1234/5678` is rewritten to obtain the resource at `/rewritten?var1=1234&var2=5678`. If a query string is present on the original request, it's preserved when the URL is rewritten.

There's no round trip to the server to obtain the resource. If the resource exists, it's fetched and returned to the client with a *200 - OK* status code. Because the client isn't redirected, the URL in the browser's address bar doesn't change. Clients can't detect that a URL rewrite operation occurred on the server.

> **NOTE**
>
> Use `skipRemainingRules: true` whenever possible because matching rules is computationally expensive and increases app response time. For the fastest app response:
>
> - Order rewrite rules from the most frequently matched rule to the least frequently matched rule.
> - Skip the processing of the remaining rules when a match occurs and no additional rule processing is required.

Apache mod_rewrite

Apply Apache mod_rewrite rules with AddApacheModRewrite. Make sure that the rules file is deployed with the app. For more information and examples of mod_rewrite rules, see Apache mod_rewrite.

A StreamReader is used to read the rules from the *ApacheModRewrite.txt* rules file:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

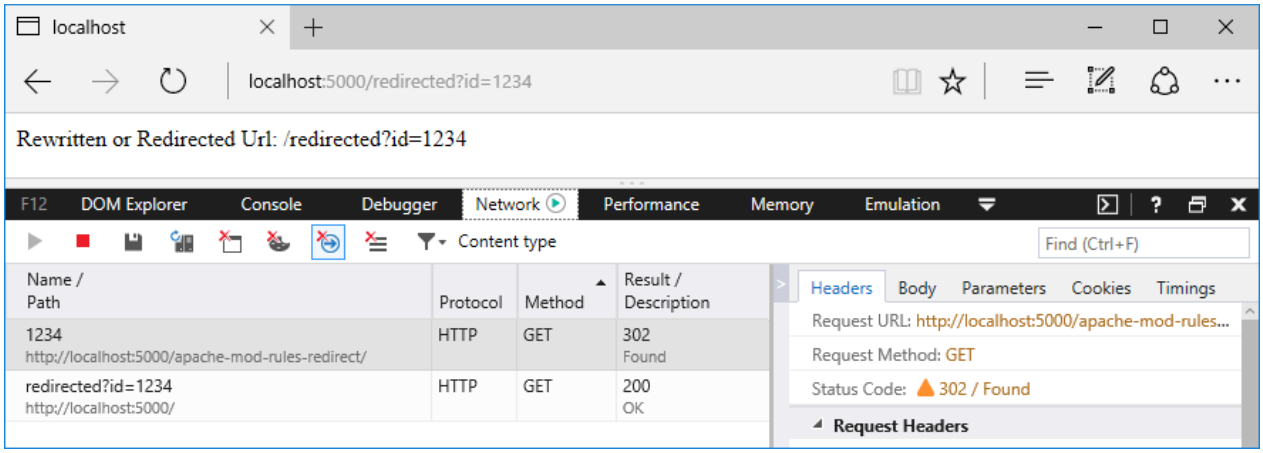
    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The sample app redirects requests from `/apache-mod-rules-redirect/(.\*)` to `/redirected?id=$1`. The response status code is *302 - Found*.

```
# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

Original Request: `/apache-mod-rules-redirect/1234`



The middleware supports the following Apache mod_rewrite server variables:

- CONN_REMOTE_ADDR
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_FORWARDED
- HTTP_HOST
- HTTP_REFERER

- HTTP_USER_AGENT
- HTTPS
- IPV6
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_METHOD
- REQUEST_SCHEME
- REQUEST_URI
- SCRIPT_FILENAME
- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL
- TIME
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_MON
- TIME_SEC
- TIME_WDAY
- TIME_YEAR

IIS URL Rewrite Module rules

To use the same rule set that applies to the IIS URL Rewrite Module, use AddIISUrlRewrite. Make sure that the rules file is deployed with the app. Don't direct the middleware to use the app's *web.config* file when running on Windows Server IIS. With IIS, these rules should be stored outside of the app's *web.config* file in order to avoid conflicts with the IIS Rewrite module. For more information and examples of IIS URL Rewrite Module rules, see Using Url Rewrite Module 2.0 and URL Rewrite Module Configuration Reference.

A StreamReader is used to read the rules from the *IISUrlRewrite.xml* rules file:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

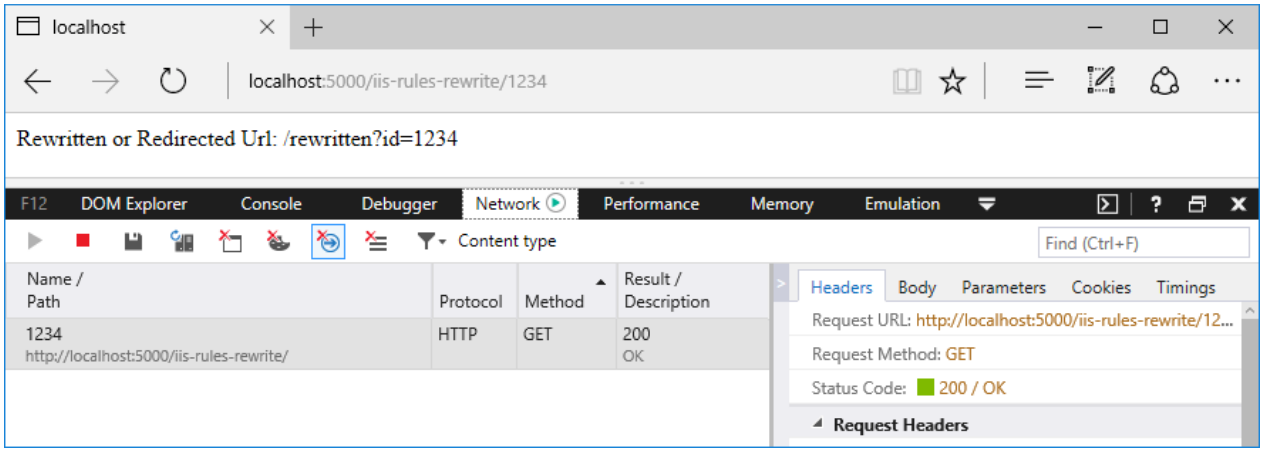
    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The sample app rewrites requests from `/iis-rules-rewrite/(.*)` to `/rewritten?id=$1`. The response is sent to the client with a *200 - OK* status code.

```
<rewrite>
  <rules>
    <rule name="Rewrite segment to id querystring" stopProcessing="true">
      <match url="^iis-rules-rewrite/(.*)$" />
      <action type="Rewrite" url="rewritten?id={R:1}" appendQueryString="false"/>
    </rule>
  </rules>
</rewrite>
```

Original Request: `/iis-rules-rewrite/1234`



If you have an active IIS Rewrite Module with server-level rules configured that would impact your app in undesirable ways, you can disable the IIS Rewrite Module for an app. For more information, see Disabling IIS modules.

Unsupported features

The middleware released with ASP.NET Core 2.x doesn't support the following IIS URL Rewrite Module features:

- Outbound Rules
- Custom Server Variables
- Wildcards
- LogRewrittenUrl

Supported server variables

The middleware supports the following IIS URL Rewrite Module server variables:

- CONTENT_LENGTH
- CONTENT_TYPE
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_HOST
- HTTP_REFERER
- HTTP_URL
- HTTP_USER_AGENT
- HTTPS
- LOCAL_ADDR
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_URI

> **NOTE**
>
> You can also obtain an IFileProvider via a PhysicalFileProvider. This approach may provide greater flexibility for the location of your rewrite rules files. Make sure that your rewrite rules files are deployed to the server at the path you provide.
>
> ```
> PhysicalFileProvider fileProvider = new PhysicalFileProvider(Directory.GetCurrentDirectory());
> ```

Method-based rule

Use Add to implement your own rule logic in a method. `Add` exposes the RewriteContext, which makes available the HttpContext for use in your method. The RewriteContext.Result determines how additional pipeline processing is handled. Set the value to one of the RuleResult fields described in the following table.

REWRITE CONTEXT RESULT	ACTION
`RuleResult.ContinueRules` (default)	Continue applying rules.
`RuleResult.EndResponse`	Stop applying rules and send the response.
`RuleResult.SkipRemainingRules`	Stop applying rules and send the context to the next middleware.

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The sample app demonstrates a method that redirects requests for paths that end with *.xml*. If a request is made for `/file.xml`, the request is redirected to `/xmlfiles/file.xml`. The status code is set to *301 - Moved Permanently*. When the browser makes a new request for */xmlfiles/file.xml*, Static File Middleware serves the file to the client from the *wwwroot/xmlfiles* folder. For a redirect, explicitly set the status code of the response. Otherwise, a *200 - OK* status code is returned, and the redirect doesn't occur on the client.

RewriteRules.cs:

```
public static void RedirectXmlFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    // Because the client is redirecting back to the same app, stop
    // processing if the request has already been redirected.
    if (request.Path.StartsWithSegments(new PathString("/xmlfiles")))
    {
        return;
    }

    if (request.Path.Value.EndsWith(".xml", StringComparison.OrdinalIgnoreCase))
    {
        var response = context.HttpContext.Response;
        response.StatusCode = StatusCodes.Status301MovedPermanently;
        context.Result = RuleResult.EndResponse;
        response.Headers[HeaderNames.Location] =
            "/xmlfiles" + request.Path + request.QueryString;
    }
}
```

This approach can also rewrite requests. The sample app demonstrates rewriting the path for any text file request to serve the *file.txt* text file from the *wwwroot* folder. Static File Middleware serves the file based on the updated request path:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

RewriteRules.cs:

```
public static void RewriteTextFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    if (request.Path.Value.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
    {
        context.Result = RuleResult.SkipRemainingRules;
        request.Path = "/file.txt";
    }
}
```

IRule-based rule

Use Add to use rule logic in a class that implements the IRule interface. `IRule` provides greater flexibility over using the method-based rule approach. Your implementation class may include a constructor that allows you can pass in parameters for the ApplyRule method.

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The values of the parameters in the sample app for the `extension` and the `newPath` are checked to meet several conditions. The `extension` must contain a value, and the value must be *.png*, *.jpg*, or *.gif*. If the `newPath` isn't valid, an ArgumentException is thrown. If a request is made for *image.png*, the request is redirected to `/png-images/image.png`. If a request is made for *image.jpg*, the request is redirected to `/jpg-images/image.jpg`. The status code is set to *301 - Moved Permanently*, and the `context.Result` is set to stop processing rules and send the response.

```
public class RedirectImageRequests : IRule
{
    private readonly string _extension;
    private readonly PathString _newPath;

    public RedirectImageRequests(string extension, string newPath)
    {
        if (string.IsNullOrEmpty(extension))
        {
            throw new ArgumentException(nameof(extension));
        }

        if (!Regex.IsMatch(extension, @"^\.(png|jpg|gif)$"))
        {
            throw new ArgumentException("Invalid extension", nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"(/[A-Za-z0-9]+)+?"))
        {
            throw new ArgumentException("Invalid path", nameof(newPath));
        }

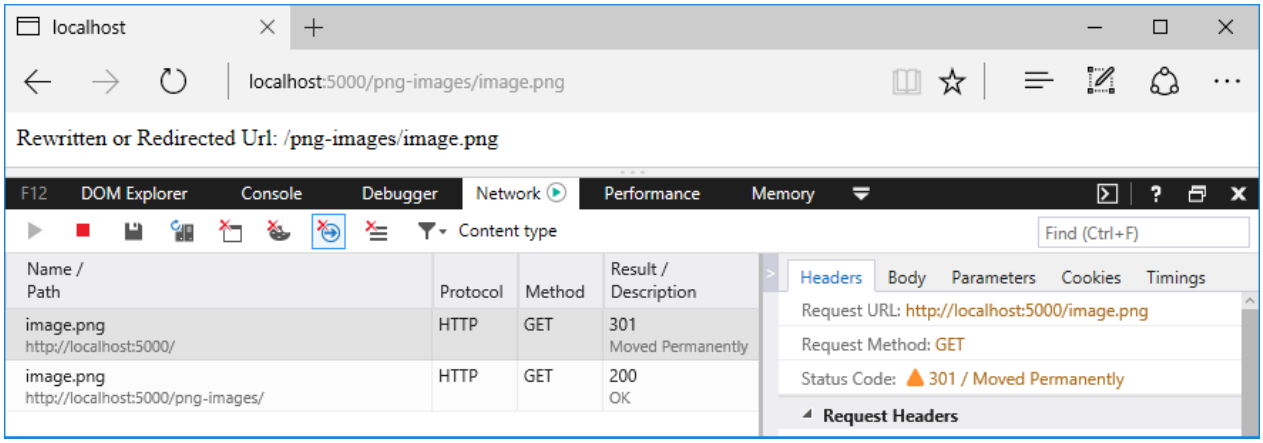
        _extension = extension;
        _newPath = new PathString(newPath);
    }

    public void ApplyRule(RewriteContext context)
    {
        var request = context.HttpContext.Request;

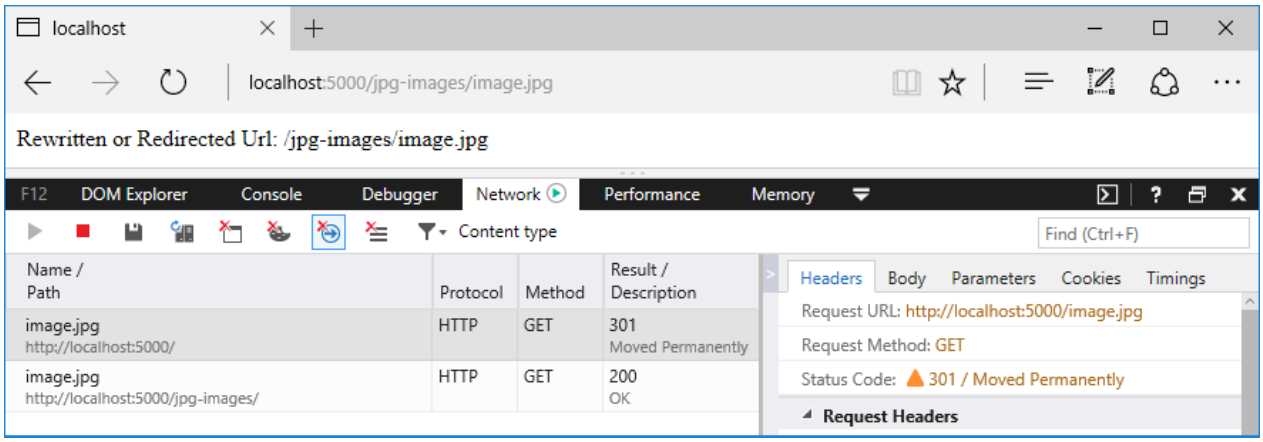
        // Because we're redirecting back to the same app, stop
        // processing if the request has already been redirected
        if (request.Path.StartsWithSegments(new PathString(_newPath)))
        {
            return;
        }

        if (request.Path.Value.EndsWith(_extension, StringComparison.OrdinalIgnoreCase))
        {
            var response = context.HttpContext.Response;
            response.StatusCode = StatusCodes.Status301MovedPermanently;
            context.Result = RuleResult.EndResponse;
            response.Headers[HeaderNames.Location] =
                _newPath + request.Path + request.QueryString;
        }
    }
}
```

Original Request: `/image.png`



Original Request: `/image.jpg`



Regex examples

GOAL	REGEX STRING & MATCH EXAMPLE	REPLACEMENT STRING & OUTPUT EXAMPLE
Rewrite path into querystring	`^path/(.*)/(.*)` `/path/abc/123`	`path?var1=$1&var2=$2` `/path?var1=abc&var2=123`
Strip trailing slash	`(.*)/$` `/path/`	`$1` `/path`
Enforce trailing slash	`(.*[^/])$` `/path`	`$1/` `/path/`
Avoid rewriting specific requests	`^(.*)(?<!\.axd)$` or `^(?!.*\.axd$)(.*)$` Yes: `/resource.htm` No: `/resource.axd`	`rewritten/$1` `/rewritten/resource.htm` `/resource.axd`
Rearrange URL segments	`path/(.*)/(.*)/(.*)` `path/1/2/3`	`path/$3/$2/$1` `path/3/2/1`
Replace a URL segment	`^(.*)/segment2/(.*)` `/segment1/segment2/segment3`	`$1/replaced/$2` `/segment1/replaced/segment3`

This document introduces URL rewriting with instructions on how to use URL Rewriting Middleware in ASP.NET Core apps.

URL rewriting is the act of modifying request URLs based on one or more predefined rules. URL rewriting creates an abstraction between resource locations and their addresses so that the locations and addresses aren't tightly linked. URL rewriting is valuable in several scenarios to:

- Move or replace server resources temporarily or permanently and maintain stable locators for those resources.
- Split request processing across different apps or across areas of one app.
- Remove, add, or reorganize URL segments on incoming requests.
- Optimize public URLs for Search Engine Optimization (SEO).
- Permit the use of friendly public URLs to help visitors predict the content returned by requesting a resource.
- Redirect insecure requests to secure endpoints.
- Prevent hotlinking, where an external site uses a hosted static asset on another site by linking the asset into its own content.

> **NOTE**
> URL rewriting can reduce the performance of an app. Where feasible, limit the number and complexity of rules.

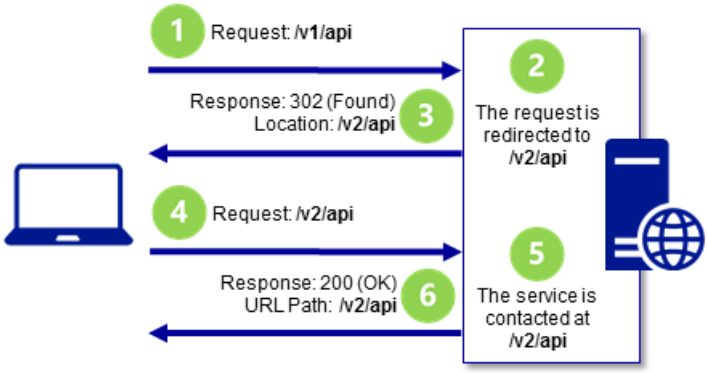
View or download sample code (how to download)

URL redirect and URL rewrite

The difference in wording between *URL redirect* and *URL rewrite* is subtle but has important implications for providing resources to clients. ASP.NET Core's URL Rewriting Middleware is capable of meeting the need for both.

A *URL redirect* involves a client-side operation, where the client is instructed to access a resource at a different address than the client originally requested. This requires a round trip to the server. The redirect URL returned to the client appears in the browser's address bar when the client makes a new request for the resource.

If `/resource` is *redirected* to `/different-resource`, the server responds that the client should obtain the resource at `/different-resource` with a status code indicating that the redirect is either temporary or permanent.



When redirecting requests to a different URL, indicate whether the redirect is permanent or temporary by specifying the status code with the response:

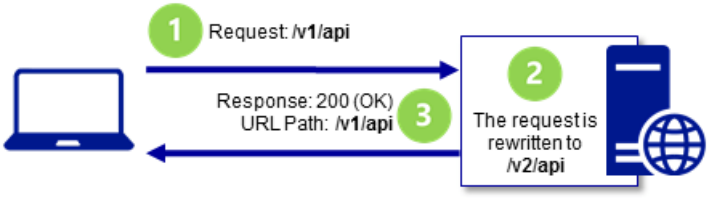
- The *301 - Moved Permanently* status code is used where the resource has a new, permanent URL and you wish to instruct the client that all future requests for the resource should use the new URL. *The client may cache and reuse the response when a 301 status code is received.*
- The *302 - Found* status code is used where the redirection is temporary or generally subject to change. The 302 status code indicates to the client not to store the URL and use it in the future.

For more information on status codes, see RFC 2616: Status Code Definitions.

A *URL rewrite* is a server-side operation that provides a resource from a different resource address than the client requested. Rewriting a URL doesn't require a round trip to the server. The rewritten URL isn't returned to the client and doesn't appear in the browser's address bar.

If `/resource` is *rewritten* to `/different-resource`, the server *internally* fetches and returns the resource at `/different-resource`.

Although the client might be able to retrieve the resource at the rewritten URL, the client isn't informed that the resource exists at the rewritten URL when it makes its request and receives the response.



URL rewriting sample app

You can explore the features of the URL Rewriting Middleware with the sample app. The app applies redirect and rewrite rules and shows the redirected or rewritten URL for several scenarios.

When to use URL Rewriting Middleware

Use URL Rewriting Middleware when you're unable to use the following approaches:

- URL Rewrite module with IIS on Windows Server
- Apache mod_rewrite module on Apache Server
- URL rewriting on Nginx

Also, use the middleware when the app is hosted on HTTP.sys server (formerly called WebListener).

The main reasons to use the server-based URL rewriting technologies in IIS, Apache, and Nginx are:

- The middleware doesn't support the full features of these modules.

  Some of the features of the server modules don't work with ASP.NET Core projects, such as the `IsFile` and `IsDirectory` constraints of the IIS Rewrite module. In these scenarios, use the middleware instead.

- The performance of the middleware probably doesn't match that of the modules.

  Benchmarking is the only way to know for sure which approach degrades performance the most or if degraded performance is negligible.

Package

To include the middleware in your project, add a package reference to the Microsoft.AspNetCore.App metapackage in the project file, which contains the Microsoft.AspNetCore.Rewrite package.

When not using the `Microsoft.AspNetCore.App` metapackage, add a project reference to the `Microsoft.AspNetCore.Rewrite` package.

Extension and options

Establish URL rewrite and redirect rules by creating an instance of the RewriteOptions class with extension methods for each of your rewrite rules. Chain multiple rules in the order that you would like them processed. The `RewriteOptions` are passed into the URL Rewriting Middleware as it's added to the request pipeline with UseRewriter:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

Redirect non-www to www

Three options permit the app to redirect non-`www` requests to `www`:

- AddRedirectToWwwPermanent: Permanently redirect the request to the `www` subdomain if the request is non-`www`. Redirects with a Status308PermanentRedirect status code.
- AddRedirectToWww: Redirect the request to the `www` subdomain if the incoming request is non-`www`. Redirects with a Status307TemporaryRedirect status code. An overload permits you to provide the status code for the response. Use a field of the StatusCodes class for a status code assignment.

URL redirect

Use AddRedirect to redirect requests. The first parameter contains your regex for matching on the path of the incoming URL. The second parameter is the replacement string. The third parameter, if present, specifies the status code. If you don't specify the status code, the status code defaults to *302 - Found*, which indicates that the resource is temporarily moved or replaced.

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

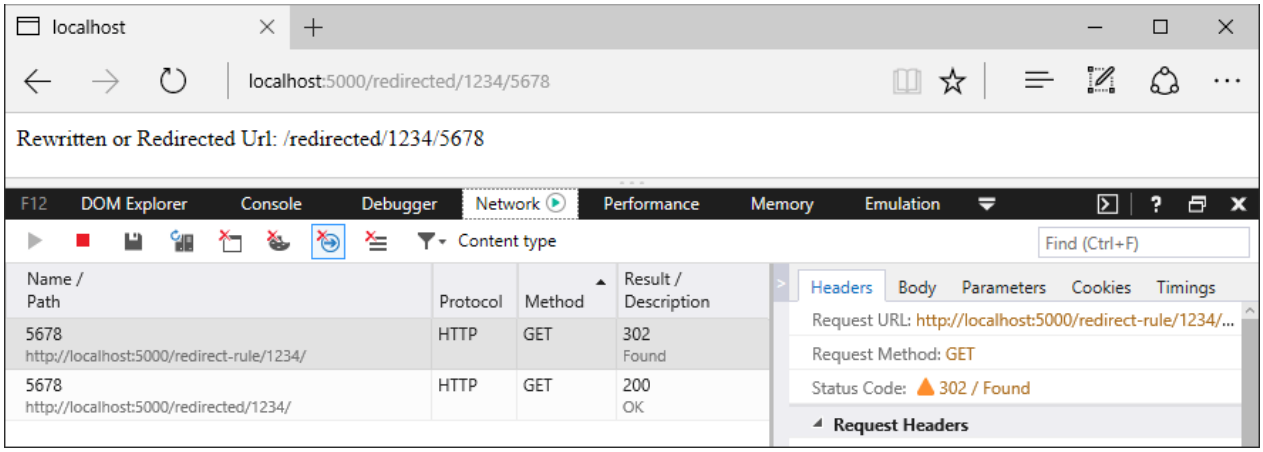
In a browser with developer tools enabled, make a request to the sample app with the path `/redirect-rule/1234/5678`. The regex matches the request path on `redirect-rule/(.*)`, and the path is replaced with `/redirected/1234/5678`. The redirect URL is sent back to the client with a *302 - Found* status code. The browser makes a new request at the redirect URL, which appears in the browser's address bar. Since no rules in the sample app match on the redirect URL:

- The second request receives a *200 - OK* response from the app.
- The body of the response shows the redirect URL.

A round trip is made to the server when a URL is *redirected*.

> **WARNING**
>
> Be cautious when establishing redirect rules. Redirect rules are evaluated on every request to the app, including after a redirect. It's easy to accidentally create a *loop of infinite redirects*.

Original Request: `/redirect-rule/1234/5678`



The part of the expression contained within parentheses is called a *capture group*. The dot (`.`) of the expression means *match any character*. The asterisk (`*`) indicates *match the preceding character zero or more times*. Therefore, the last two path segments of the URL, `1234/5678` , are captured by capture group `(.*)` . Any value you provide in the request URL after `redirect-rule/` is captured by this single capture group.

In the replacement string, captured groups are injected into the string with the dollar sign (`$`) followed by the sequence number of the capture. The first capture group value is obtained with `$1` , the second with `$2` , and they continue in sequence for the capture groups in your regex. There's only one captured group in the redirect rule regex in the sample app, so there's only one injected group in the replacement string, which is `$1` . When the rule is applied, the URL becomes `/redirected/1234/5678` .

URL redirect to a secure endpoint

Use AddRedirectToHttps to redirect HTTP requests to the same host and path using the HTTPS protocol. If the status code isn't supplied, the middleware defaults to *302 - Found*. If the port isn't supplied:

- The middleware defaults to `null` .
- The scheme changes to `https` (HTTPS protocol), and the client accesses the resource on port 443.

The following example shows how to set the status code to *301 - Moved Permanently* and change the port to 5001.

```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttps(301, 5001);

    app.UseRewriter(options);
}
```

Use AddRedirectToHttpsPermanent to redirect insecure requests to the same host and path with secure HTTPS protocol on port 443. The middleware sets the status code to *301 - Moved Permanently*.

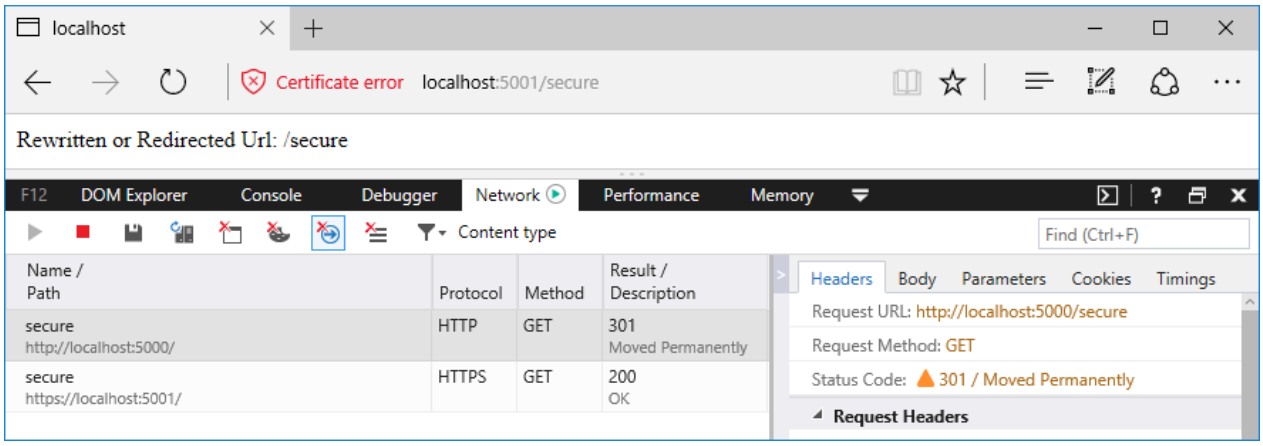
```
public void Configure(IApplicationBuilder app)
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent();

    app.UseRewriter(options);
}
```

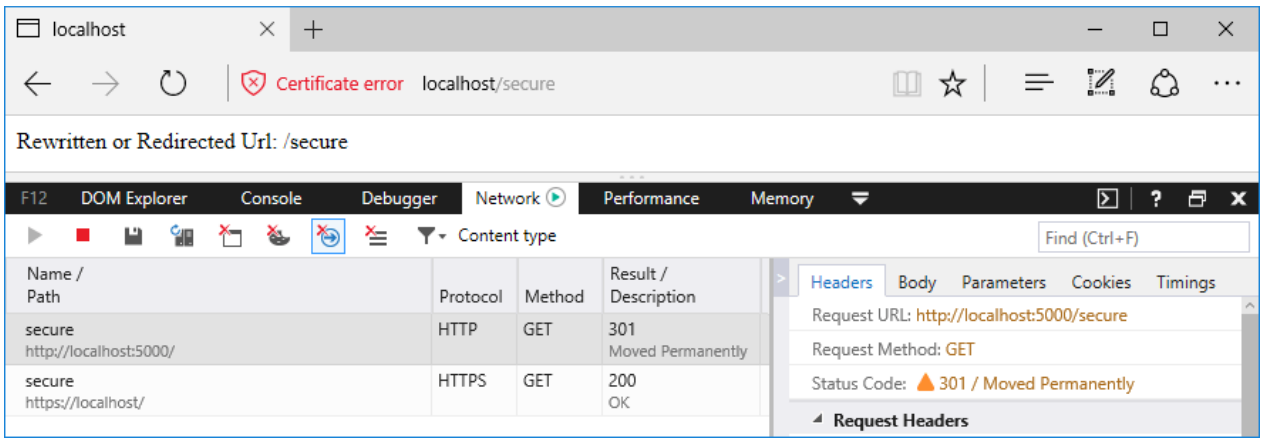
> **NOTE**
> When redirecting to a secure endpoint without the requirement for additional redirect rules, we recommend using HTTPS Redirection Middleware. For more information, see the Enforce HTTPS topic.

The sample app is capable of demonstrating how to use `AddRedirectToHttps` or `AddRedirectToHttpsPermanent` . Add the extension method to the `RewriteOptions` . Make an insecure request to the app at any URL. Dismiss the browser security warning that the self-signed certificate is untrusted or create an exception to trust the certificate.

Original Request using `AddRedirectToHttps(301, 5001)` : `http://localhost:5000/secure`



Original Request using `AddRedirectToHttpsPermanent` : `http://localhost:5000/secure`



URL rewrite

Use AddRewrite to create a rule for rewriting URLs. The first parameter contains the regex for matching on the incoming URL path. The second parameter is the replacement string. The third parameter, `skipRemainingRules: {true|false}`, indicates to the middleware whether or not to skip additional rewrite rules if the current rule is applied.

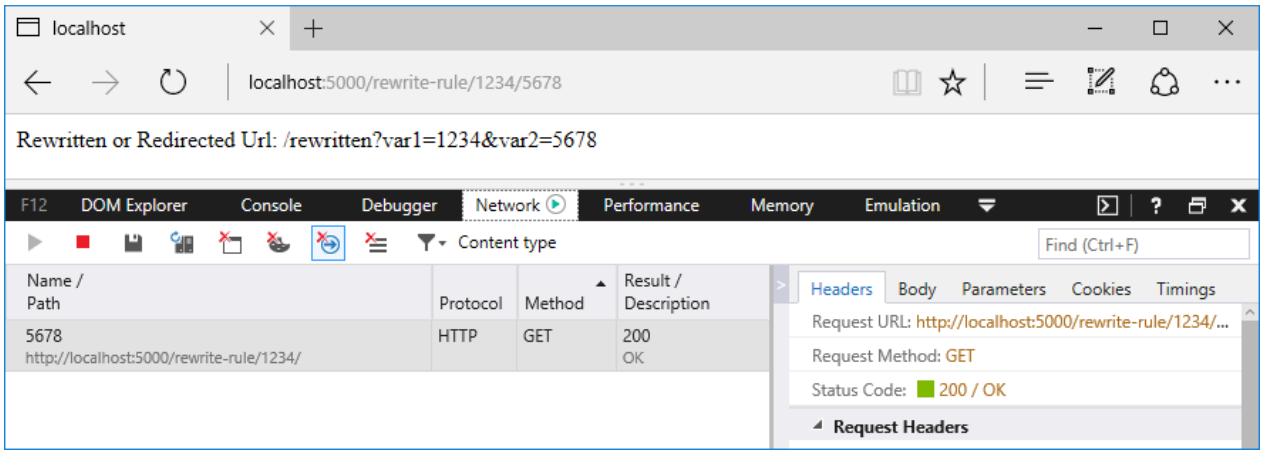
```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

Original Request: `/rewrite-rule/1234/5678`



The carat (`^`) at the beginning of the expression means that matching starts at the beginning of the URL path.

In the earlier example with the redirect rule, `redirect-rule/(.*)`, there's no carat (`^`) at the start of the regex. Therefore, any characters may precede `redirect-rule/` in the path for a successful match.

PATH	MATCH
`/redirect-rule/1234/5678`	Yes
`/my-cool-redirect-rule/1234/5678`	Yes
`/anotherredirect-rule/1234/5678`	Yes

The rewrite rule, `^rewrite-rule/(\d+)/(\d+)`, only matches paths if they start with `rewrite-rule/`. In the following table, note the difference in matching.

PATH	MATCH
`/rewrite-rule/1234/5678`	Yes
`/my-cool-rewrite-rule/1234/5678`	No
`/anotherrewrite-rule/1234/5678`	No

Following the `^rewrite-rule/` portion of the expression, there are two capture groups, `(\d+)/(\d+)`. The `\d` signifies *match a digit (number)*. The plus sign (`+`) means *match one or more of the preceding character*. Therefore, the URL must contain a number followed by a forward-slash followed by another number. These capture groups are injected into the rewritten URL as `$1` and `$2`. The rewrite rule replacement string places the captured groups into the query string. The requested path of `/rewrite-rule/1234/5678` is rewritten to obtain the resource at `/rewritten?var1=1234&var2=5678`. If a query string is present on the original request, it's preserved when the URL is rewritten.

There's no round trip to the server to obtain the resource. If the resource exists, it's fetched and returned to the client with a *200 - OK* status code. Because the client isn't redirected, the URL in the browser's address bar doesn't change. Clients can't detect that a URL rewrite operation occurred on the server.

> **NOTE**
>
> Use `skipRemainingRules: true` whenever possible because matching rules is computationally expensive and increases app response time. For the fastest app response:
>
> - Order rewrite rules from the most frequently matched rule to the least frequently matched rule.
> - Skip the processing of the remaining rules when a match occurs and no additional rule processing is required.

Apache mod_rewrite

Apply Apache mod_rewrite rules with AddApacheModRewrite. Make sure that the rules file is deployed with the app. For more information and examples of mod_rewrite rules, see Apache mod_rewrite.

A StreamReader is used to read the rules from the *ApacheModRewrite.txt* rules file:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

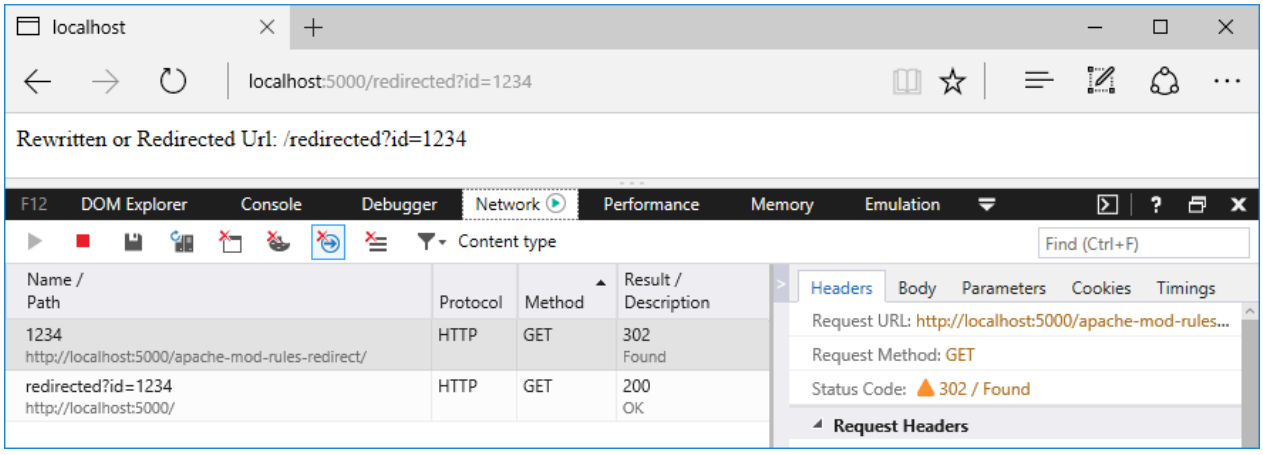
    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The sample app redirects requests from `/apache-mod-rules-redirect/(.\*)` to `/redirected?id=$1`. The response status code is *302 - Found*.

```
# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

Original Request: `/apache-mod-rules-redirect/1234`



The middleware supports the following Apache mod_rewrite server variables:

- CONN_REMOTE_ADDR
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_FORWARDED
- HTTP_HOST
- HTTP_REFERER

- HTTP_USER_AGENT
- HTTPS
- IPV6
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_METHOD
- REQUEST_SCHEME
- REQUEST_URI
- SCRIPT_FILENAME
- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL
- TIME
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_MON
- TIME_SEC
- TIME_WDAY
- TIME_YEAR

IIS URL Rewrite Module rules

To use the same rule set that applies to the IIS URL Rewrite Module, use AddIISUrlRewrite. Make sure that the rules file is deployed with the app. Don't direct the middleware to use the app's *web.config* file when running on Windows Server IIS. With IIS, these rules should be stored outside of the app's *web.config* file in order to avoid conflicts with the IIS Rewrite module. For more information and examples of IIS URL Rewrite Module rules, see Using Url Rewrite Module 2.0 and URL Rewrite Module Configuration Reference.

A StreamReader is used to read the rules from the *IISUrlRewrite.xml* rules file:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

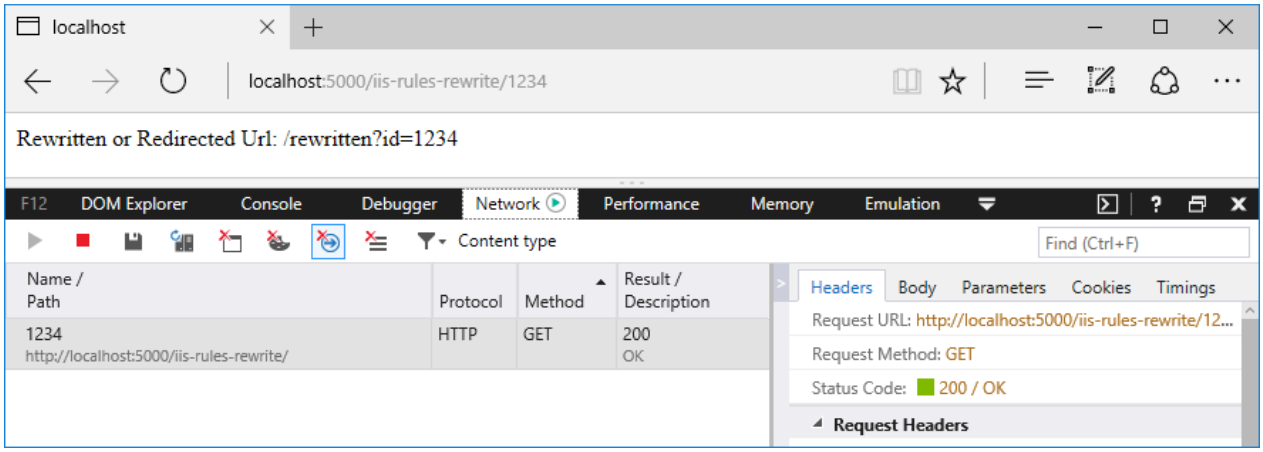
    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The sample app rewrites requests from `/iis-rules-rewrite/(.*)` to `/rewritten?id=$1`. The response is sent to the client with a *200 - OK* status code.

```
<rewrite>
  <rules>
    <rule name="Rewrite segment to id querystring" stopProcessing="true">
      <match url="^iis-rules-rewrite/(.*)$" />
      <action type="Rewrite" url="rewritten?id={R:1}" appendQueryString="false"/>
    </rule>
  </rules>
</rewrite>
```

Original Request: `/iis-rules-rewrite/1234`



If you have an active IIS Rewrite Module with server-level rules configured that would impact your app in undesirable ways, you can disable the IIS Rewrite Module for an app. For more information, see Disabling IIS modules.

Unsupported features

The middleware released with ASP.NET Core 2.x doesn't support the following IIS URL Rewrite Module features:

- Outbound Rules
- Custom Server Variables
- Wildcards
- LogRewrittenUrl

Supported server variables

The middleware supports the following IIS URL Rewrite Module server variables:

- CONTENT_LENGTH
- CONTENT_TYPE
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_HOST
- HTTP_REFERER
- HTTP_URL
- HTTP_USER_AGENT
- HTTPS
- LOCAL_ADDR
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_URI

> **NOTE**
>
> You can also obtain an IFileProvider via a PhysicalFileProvider. This approach may provide greater flexibility for the location of your rewrite rules files. Make sure that your rewrite rules files are deployed to the server at the path you provide.
>
> ```
> PhysicalFileProvider fileProvider = new PhysicalFileProvider(Directory.GetCurrentDirectory());
> ```

Method-based rule

Use Add to implement your own rule logic in a method. `Add` exposes the RewriteContext, which makes available the HttpContext for use in your method. The RewriteContext.Result determines how additional pipeline processing is handled. Set the value to one of the RuleResult fields described in the following table.

REWRITE CONTEXT RESULT	ACTION
`RuleResult.ContinueRules` (default)	Continue applying rules.
`RuleResult.EndResponse`	Stop applying rules and send the response.
`RuleResult.SkipRemainingRules`	Stop applying rules and send the context to the next middleware.

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The sample app demonstrates a method that redirects requests for paths that end with *.xml*. If a request is made for `/file.xml`, the request is redirected to `/xmlfiles/file.xml`. The status code is set to *301 - Moved Permanently*. When the browser makes a new request for */xmlfiles/file.xml*, Static File Middleware serves the file to the client from the *wwwroot/xmlfiles* folder. For a redirect, explicitly set the status code of the response. Otherwise, a *200 - OK* status code is returned, and the redirect doesn't occur on the client.

RewriteRules.cs:

```
public static void RedirectXmlFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    // Because the client is redirecting back to the same app, stop
    // processing if the request has already been redirected.
    if (request.Path.StartsWithSegments(new PathString("/xmlfiles")))
    {
        return;
    }

    if (request.Path.Value.EndsWith(".xml", StringComparison.OrdinalIgnoreCase))
    {
        var response = context.HttpContext.Response;
        response.StatusCode = StatusCodes.Status301MovedPermanently;
        context.Result = RuleResult.EndResponse;
        response.Headers[HeaderNames.Location] =
            "/xmlfiles" + request.Path + request.QueryString;
    }
}
```

This approach can also rewrite requests. The sample app demonstrates rewriting the path for any text file request to serve the *file.txt* text file from the *wwwroot* folder. Static File Middleware serves the file based on the updated request path:

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

RewriteRules.cs:

```
public static void RewriteTextFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    if (request.Path.Value.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
    {
        context.Result = RuleResult.SkipRemainingRules;
        request.Path = "/file.txt";
    }
}
```

IRule-based rule

Use Add to use rule logic in a class that implements the IRule interface. `IRule` provides greater flexibility over using the method-based rule approach. Your implementation class may include a constructor that allows you can pass in parameters for the ApplyRule method.

```
public void Configure(IApplicationBuilder app)
{
    using (StreamReader apacheModRewriteStreamReader =
        File.OpenText("ApacheModRewrite.txt"))
    using (StreamReader iisUrlRewriteStreamReader =
        File.OpenText("IISUrlRewrite.xml"))
    {
        var options = new RewriteOptions()
            .AddRedirect("redirect-rule/(.*)", "redirected/$1")
            .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?var1=$1&var2=$2",
                skipRemainingRules: true)
            .AddApacheModRewrite(apacheModRewriteStreamReader)
            .AddIISUrlRewrite(iisUrlRewriteStreamReader)
            .Add(MethodRules.RedirectXmlFileRequests)
            .Add(MethodRules.RewriteTextFileRequests)
            .Add(new RedirectImageRequests(".png", "/png-images"))
            .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

        app.UseRewriter(options);
    }

    app.UseStaticFiles();

    app.Run(context => context.Response.WriteAsync(
        $"Rewritten or Redirected Url: " +
        $"{context.Request.Path + context.Request.QueryString}"));
}
```

The values of the parameters in the sample app for the `extension` and the `newPath` are checked to meet several conditions. The `extension` must contain a value, and the value must be *.png*, *.jpg*, or *.gif*. If the `newPath` isn't valid, an ArgumentException is thrown. If a request is made for *image.png*, the request is redirected to `/png-images/image.png`. If a request is made for *image.jpg*, the request is redirected to `/jpg-images/image.jpg`. The status code is set to *301 - Moved Permanently*, and the `context.Result` is set to stop processing rules and send the response.

```
public class RedirectImageRequests : IRule
{
    private readonly string _extension;
    private readonly PathString _newPath;

    public RedirectImageRequests(string extension, string newPath)
    {
        if (string.IsNullOrEmpty(extension))
        {
            throw new ArgumentException(nameof(extension));
        }

        if (!Regex.IsMatch(extension, @"^\.(png|jpg|gif)$"))
        {
            throw new ArgumentException("Invalid extension", nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"(/[A-Za-z0-9]+)+?"))
        {
            throw new ArgumentException("Invalid path", nameof(newPath));
        }

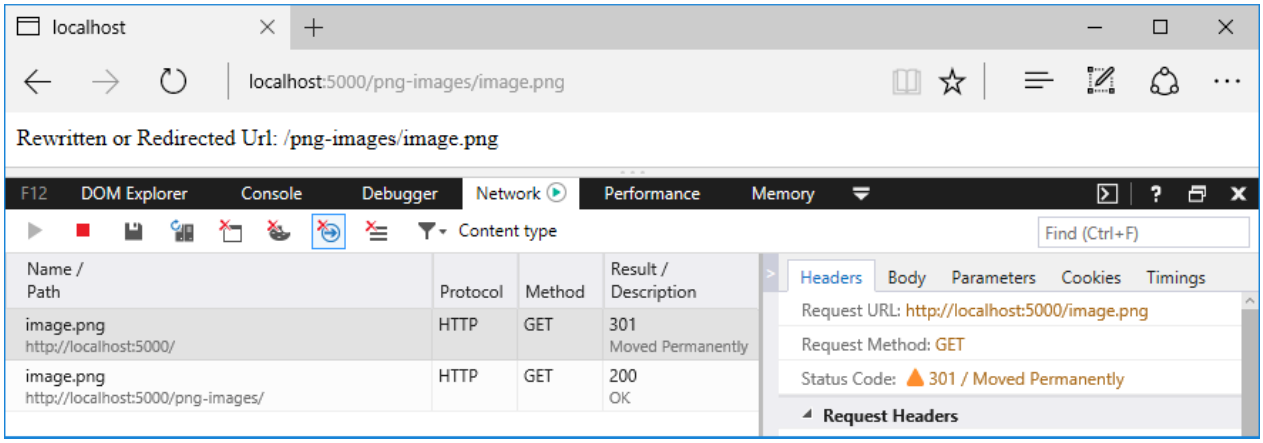
        _extension = extension;
        _newPath = new PathString(newPath);
    }

    public void ApplyRule(RewriteContext context)
    {
        var request = context.HttpContext.Request;

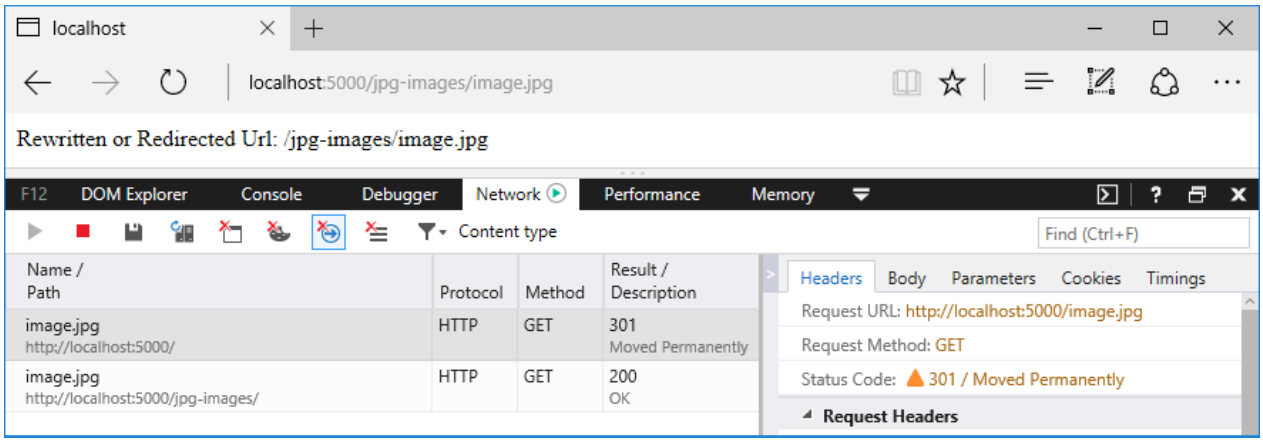
        // Because we're redirecting back to the same app, stop
        // processing if the request has already been redirected
        if (request.Path.StartsWithSegments(new PathString(_newPath)))
        {
            return;
        }

        if (request.Path.Value.EndsWith(_extension, StringComparison.OrdinalIgnoreCase))
        {
            var response = context.HttpContext.Response;
            response.StatusCode = StatusCodes.Status301MovedPermanently;
            context.Result = RuleResult.EndResponse;
            response.Headers[HeaderNames.Location] =
                _newPath + request.Path + request.QueryString;
        }
    }
}
```

Original Request: `/image.png`



Original Request: `/image.jpg`



Regex examples

GOAL	REGEX STRING & MATCH EXAMPLE	REPLACEMENT STRING & OUTPUT EXAMPLE
Rewrite path into querystring	`^path/(.*)/(.*)` `/path/abc/123`	`path?var1=$1&var2=$2` `/path?var1=abc&var2=123`
Strip trailing slash	`(.*)/$` `/path/`	`$1` `/path`
Enforce trailing slash	`(.*[^/])$` `/path`	`$1/` `/path/`
Avoid rewriting specific requests	`^(.*)(?<!\.axd)$` or `^(?!.*\.axd$)(.*)$` Yes: `/resource.htm` No: `/resource.axd`	`rewritten/$1` `/rewritten/resource.htm` `/resource.axd`
Rearrange URL segments	`path/(.*)/(.*)/(.*)` `path/1/2/3`	`path/$3/$2/$1` `path/3/2/1`
Replace a URL segment	`^(.*)/segment2/(.*)` `/segment1/segment2/segment3`	`$1/replaced/$2` `/segment1/replaced/segment3`

Additional resources

- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Regular expressions in .NET
- Regular expression language - quick reference
- Apache mod_rewrite
- Using Url Rewrite Module 2.0 (for IIS)
- URL Rewrite Module Configuration Reference
- IIS URL Rewrite Module Forum
- Keep a simple URL structure
- 10 URL Rewriting Tips and Tricks
- To slash or not to slash

File Providers in ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Steve Smith

ASP.NET Core abstracts file system access through the use of File Providers. File Providers are used throughout the ASP.NET Core framework. For example:

- IWebHostEnvironment exposes the app's content root and web root as `IFileProvider` types.
- Static File Middleware uses File Providers to locate static files.
- Razor uses File Providers to locate pages and views.
- .NET Core tooling uses File Providers and glob patterns to specify which files should be published.

View or download sample code (how to download)

File Provider interfaces

The primary interface is IFileProvider. `IFileProvider` exposes methods to:

- Obtain file information (IFileInfo).
- Obtain directory information (IDirectoryContents).
- Set up change notifications (using an IChangeToken).

`IFileInfo` provides methods and properties for working with files:

- Exists
- IsDirectory
- Name
- Length (in bytes)
- LastModified date

You can read from the file using the IFileInfo.CreateReadStream method.

The *FileProviderSample* sample app demonstrates how to configure a File Provider in `Startup.ConfigureServices` for use throughout the app via dependency injection.

File Provider implementations

The following table lists implementations of `IFileProvider`.

IMPLEMENTATION	DESCRIPTION
CompositeFileProvider	Used to provide combined access to files and directories from one or more other providers.
ManifestEmbeddedFileProvider	Used to access files embedded in assemblies.
PhysicalFileProvider	Used to access the system's physical files.

PhysicalFileProvider

The PhysicalFileProvider provides access to the physical file system. `PhysicalFileProvider` uses the System.IO.File

type (for the physical provider) and scopes all paths to a directory and its children. This scoping prevents access to the file system outside of the specified directory and its children. The most common scenario for creating and using a `PhysicalFileProvider` is to request an `IFileProvider` in a constructor through dependency injection.

When instantiating this provider directly, an absolute directory path is required and serves as the base path for all requests made using the provider. Glob patterns aren't supported in the directory path.

The following code shows how to use `PhysicalFileProvider` to obtain directory contents and file information:

```
var provider = new PhysicalFileProvider(applicationRoot);
var contents = provider.GetDirectoryContents(string.Empty);
var filePath = Path.Combine("wwwroot", "js", "site.js");
var fileInfo = provider.GetFileInfo(filePath);
```

Types in the preceding example:

- `provider` is an `IFileProvider`.
- `contents` is an `IDirectoryContents`.
- `fileInfo` is an `IFileInfo`.

The File Provider can be used to iterate through the directory specified by `applicationRoot` or call `GetFileInfo` to obtain a file's information. Glob patterns can't be passed to the `GetFileInfo` method. The File Provider has no access outside of the `applicationRoot` directory.

The *FileProviderSample* sample app creates the provider in the `Startup.ConfigureServices` method using IHostingEnvironment.ContentRootFileProvider:

```
var physicalProvider = _env.ContentRootFileProvider;
```

ManifestEmbeddedFileProvider

The ManifestEmbeddedFileProvider is used to access files embedded within assemblies. The `ManifestEmbeddedFileProvider` uses a manifest compiled into the assembly to reconstruct the original paths of the embedded files.

To generate a manifest of the embedded files:

1. Add the Microsoft.Extensions.FileProviders.Embedded NuGet package to your project.
2. Set the `<GenerateEmbeddedFilesManifest>` property to `true`. Specify the files to embed with <EmbeddedResource>:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.FileProviders.Embedded" Version="3.1.0" />
  </ItemGroup>

  <ItemGroup>
    <EmbeddedResource Include="Resource.txt" />
  </ItemGroup>

</Project>
```

Use glob patterns to specify one or more files to embed into the assembly.

The *FileProviderSample* sample app creates an `ManifestEmbeddedFileProvider` and passes the currently executing assembly to its constructor.

Startup.cs:

```
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
```

Additional overloads allow you to:

- Specify a relative file path.
- Scope files to a last modified date.
- Name the embedded resource containing the embedded file manifest.

OVERLOAD	DESCRIPTION
`ManifestEmbeddedFileProvider(Assembly, String)`	Accepts an optional `root` relative path parameter. Specify the `root` to scope calls to GetDirectoryContents to those resources under the provided path.
`ManifestEmbeddedFileProvider(Assembly, String, DateTimeOffset)`	Accepts an optional `root` relative path parameter and a `lastModified` date (DateTimeOffset) parameter. The `lastModified` date scopes the last modification date for the IFileInfo instances returned by the IFileProvider.
`ManifestEmbeddedFileProvider(Assembly, String, String, DateTimeOffset)`	Accepts an optional `root` relative path, `lastModified` date, and `manifestName` parameters. The `manifestName` represents the name of the embedded resource containing the manifest.

CompositeFileProvider

The CompositeFileProvider combines `IFileProvider` instances, exposing a single interface for working with files from multiple providers. When creating the `CompositeFileProvider`, pass one or more `IFileProvider` instances to its constructor.

In the *FileProviderSample* sample app, a `PhysicalFileProvider` and a `ManifestEmbeddedFileProvider` provide files to a `CompositeFileProvider` registered in the app's service container. The following code is found in the project's `Startup.ConfigureServices` method:

```
var physicalProvider = _env.ContentRootFileProvider;
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
var compositeProvider =
    new CompositeFileProvider(physicalProvider, manifestEmbeddedProvider);

services.AddSingleton<IFileProvider>(compositeProvider);
```

Watch for changes

The IFileProvider.Watch method provides a scenario to watch one or more files or directories for changes. The `Watch` method:

- Accepts a file path string, which can use glob patterns to specify multiple files.

- Returns an IChangeToken.

The resulting change token exposes:

- HasChanged: A property that can be inspected to determine if a change has occurred.
- RegisterChangeCallback: Called when changes are detected to the specified path string. Each change token only calls its associated callback in response to a single change. To enable constant monitoring, use a TaskCompletionSource<TResult> (shown below) or recreate `IChangeToken` instances in response to changes.

The *WatchConsole* sample app writes a message whenever a *.txt* file in the *TextFiles* directory is modified:

```
private static readonly string _fileFilter = Path.Combine("TextFiles", "*.txt");

public static void Main(string[] args)
{
    Console.WriteLine($"Monitoring for changes with filter '{_fileFilter}' (Ctrl + C to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
    }
}

private static async Task MainAsync()
{
    var fileProvider = new PhysicalFileProvider(Directory.GetCurrentDirectory());
    IChangeToken token = fileProvider.Watch(_fileFilter);
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("file changed");
}
```

Some file systems, such as Docker containers and network shares, may not reliably send change notifications. Set the `DOTNET_USE_POLLING_FILE_WATCHER` environment variable to `1` or `true` to poll the file system for changes every four seconds (not configurable).

Glob patterns

File system paths use wildcard patterns called *glob (or globbing) patterns*. Specify groups of files with these patterns. The two wildcard characters are `*` and `**`:

`*`
Matches anything at the current folder level, any filename, or any file extension. Matches are terminated by `/` and `.` characters in the file path.

`**`
Matches anything across multiple directory levels. Can be used to recursively match many files within a directory hierarchy.

The following table provides common examples of glob patterns.

PATTERN	DESCRIPTION
`directory/file.txt`	Matches a specific file in a specific directory.

PATTERN	DESCRIPTION
`directory/*.txt`	Matches all files with *.txt* extension in a specific directory.
`directory/*/appsettings.json`	Matches all *appsettings.json* files in directories exactly one level below the *directory* folder.
`directory/**/*.txt`	Matches all files with a *.txt* extension found anywhere under the *directory* folder.

ASP.NET Core abstracts file system access through the use of File Providers. File Providers are used throughout the ASP.NET Core framework:

- IHostingEnvironment exposes the app's content root and web root as `IFileProvider` types.
- Static File Middleware uses File Providers to locate static files.
- Razor uses File Providers to locate pages and views.
- .NET Core tooling uses File Providers and glob patterns to specify which files should be published.

View or download sample code (how to download)

File Provider interfaces

The primary interface is IFileProvider. `IFileProvider` exposes methods to:

- Obtain file information (IFileInfo).
- Obtain directory information (IDirectoryContents).
- Set up change notifications (using an IChangeToken).

`IFileInfo` provides methods and properties for working with files:

- Exists
- IsDirectory
- Name
- Length (in bytes)
- LastModified date

You can read from the file using the IFileInfo.CreateReadStream method.

The sample app demonstrates how to configure a File Provider in `Startup.ConfigureServices` for use throughout the app via dependency injection.

File Provider implementations

Three implementations of `IFileProvider` are available.

IMPLEMENTATION	DESCRIPTION
PhysicalFileProvider	The physical provider is used to access the system's physical files.
ManifestEmbeddedFileProvider	The manifest embedded provider is used to access files embedded in assemblies.

IMPLEMENTATION	DESCRIPTION
CompositeFileProvider	The composite provider is used to provide combined access to files and directories from one or more other providers.

PhysicalFileProvider

The PhysicalFileProvider provides access to the physical file system. `PhysicalFileProvider` uses the System.IO.File type (for the physical provider) and scopes all paths to a directory and its children. This scoping prevents access to the file system outside of the specified directory and its children. The most common scenario for creating and using a `PhysicalFileProvider` is to request an `IFileProvider` in a constructor through dependency injection.

When instantiating this provider directly, a directory path is required and serves as the base path for all requests made using the provider.

The following code shows how to create a `PhysicalFileProvider` and use it to obtain directory contents and file information:

```
var provider = new PhysicalFileProvider(applicationRoot);
var contents = provider.GetDirectoryContents(string.Empty);
var fileInfo = provider.GetFileInfo("wwwroot/js/site.js");
```

Types in the preceding example:

- `provider` is an `IFileProvider`.
- `contents` is an `IDirectoryContents`.
- `fileInfo` is an `IFileInfo`.

The File Provider can be used to iterate through the directory specified by `applicationRoot` or call `GetFileInfo` to obtain a file's information. The File Provider has no access outside of the `applicationRoot` directory.

The sample app creates the provider in the app's `Startup.ConfigureServices` class using IHostingEnvironment.ContentRootFileProvider:

```
var physicalProvider = _env.ContentRootFileProvider;
```

ManifestEmbeddedFileProvider

The ManifestEmbeddedFileProvider is used to access files embedded within assemblies. The `ManifestEmbeddedFileProvider` uses a manifest compiled into the assembly to reconstruct the original paths of the embedded files.

To generate a manifest of the embedded files, set the `<GenerateEmbeddedFilesManifest>` property to `true`. Specify the files to embed with <EmbeddedResource>:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
    <GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup>
    <EmbeddedResource Include="Resource.txt" />
  </ItemGroup>

</Project>
```

Use glob patterns to specify one or more files to embed into the assembly.

The sample app creates an `ManifestEmbeddedFileProvider` and passes the currently executing assembly to its constructor.

Startup.cs:

```
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
```

Additional overloads allow you to:

- Specify a relative file path.
- Scope files to a last modified date.
- Name the embedded resource containing the embedded file manifest.

OVERLOAD	DESCRIPTION
`ManifestEmbeddedFileProvider(Assembly, String)`	Accepts an optional `root` relative path parameter. Specify the `root` to scope calls to GetDirectoryContents to those resources under the provided path.
`ManifestEmbeddedFileProvider(Assembly, String, DateTimeOffset)`	Accepts an optional `root` relative path parameter and a `lastModified` date (DateTimeOffset) parameter. The `lastModified` date scopes the last modification date for the IFileInfo instances returned by the IFileProvider.
`ManifestEmbeddedFileProvider(Assembly, String, String, DateTimeOffset)`	Accepts an optional `root` relative path, `lastModified` date, and `manifestName` parameters. The `manifestName` represents the name of the embedded resource containing the manifest.

CompositeFileProvider

The CompositeFileProvider combines `IFileProvider` instances, exposing a single interface for working with files from multiple providers. When creating the `CompositeFileProvider`, pass one or more `IFileProvider` instances to its constructor.

In the sample app, a `PhysicalFileProvider` and a `ManifestEmbeddedFileProvider` provide files to a `CompositeFileProvider` registered in the app's service container:

```
var physicalProvider = _env.ContentRootFileProvider;
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
var compositeProvider =
    new CompositeFileProvider(physicalProvider, manifestEmbeddedProvider);

services.AddSingleton<IFileProvider>(compositeProvider);
```

Watch for changes

The IFileProvider.Watch method provides a scenario to watch one or more files or directories for changes. `Watch` accepts a path string, which can use glob patterns to specify multiple files. `Watch` returns an IChangeToken. The change token exposes:

- HasChanged: A property that can be inspected to determine if a change has occurred.
- RegisterChangeCallback: Called when changes are detected to the specified path string. Each change token only calls its associated callback in response to a single change. To enable constant monitoring, use a TaskCompletionSource<TResult> (shown below) or recreate `IChangeToken` instances in response to changes.

In the sample app, the *WatchConsole* console app is configured to display a message whenever a text file is modified:

```
private static PhysicalFileProvider _fileProvider =
    new PhysicalFileProvider(Directory.GetCurrentDirectory());

public static void Main(string[] args)
{
    Console.WriteLine("Monitoring quotes.txt for changes (Ctrl-c to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
    }
}

private static async Task MainAsync()
{
    IChangeToken token = _fileProvider.Watch("quotes.txt");
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("quotes.txt changed");
}
```

Some file systems, such as Docker containers and network shares, may not reliably send change notifications. Set the `DOTNET_USE_POLLING_FILE_WATCHER` environment variable to `1` or `true` to poll the file system for changes every four seconds (not configurable).

Glob patterns

File system paths use wildcard patterns called *glob (or globbing) patterns*. Specify groups of files with these patterns. The two wildcard characters are `*` and `**`:

`*`

Matches anything at the current folder level, any filename, or any file extension. Matches are terminated by `/` and

`.` characters in the file path.

`**`
Matches anything across multiple directory levels. Can be used to recursively match many files within a directory hierarchy.

Glob pattern examples

`directory/file.txt`
Matches a specific file in a specific directory.

`directory/*.txt`
Matches all files with *.txt* extension in a specific directory.

`directory/*/appsettings.json`
Matches all `appsettings.json` files in directories exactly one level below the *directory* folder.

`directory/**/*.txt`
Matches all files with *.txt* extension found anywhere under the *directory* folder.

Request Features in ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Steve Smith

Web server implementation details related to HTTP requests and responses are defined in interfaces. These interfaces are used by server implementations and middleware to create and modify the application's hosting pipeline.

Feature interfaces

ASP.NET Core defines a number of HTTP feature interfaces in `Microsoft.AspNetCore.Http.Features` which are used by servers to identify the features they support. The following feature interfaces handle requests and return responses:

`IHttpRequestFeature` Defines the structure of an HTTP request, including the protocol, path, query string, headers, and body.

`IHttpResponseFeature` Defines the structure of an HTTP response, including the status code, headers, and body of the response.

`IHttpAuthenticationFeature` Defines support for identifying users based on a `ClaimsPrincipal` and specifying an authentication handler.

`IHttpUpgradeFeature` Defines support for HTTP Upgrades, which allow the client to specify which additional protocols it would like to use if the server wishes to switch protocols.

`IHttpBufferingFeature` Defines methods for disabling buffering of requests and/or responses.

`IHttpConnectionFeature` Defines properties for local and remote addresses and ports.

`IHttpRequestLifetimeFeature` Defines support for aborting connections, or detecting if a request has been terminated prematurely, such as by a client disconnect.

`IHttpSendFileFeature` Defines a method for sending files asynchronously.

`IHttpWebSocketFeature` Defines an API for supporting web sockets.

`IHttpRequestIdentifierFeature` Adds a property that can be implemented to uniquely identify requests.

`ISessionFeature` Defines `ISessionFactory` and `ISession` abstractions for supporting user sessions.

`ITlsConnectionFeature` Defines an API for retrieving client certificates.

`ITlsTokenBindingFeature` Defines methods for working with TLS token binding parameters.

> **NOTE**
>
> `ISessionFeature` isn't a server feature, but is implemented by the `SessionMiddleware` (see Managing Application State).

Feature collections

The `Features` property of `HttpContext` provides an interface for getting and setting the available HTTP features

for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features.

Middleware and request features

While servers are responsible for creating the feature collection, middleware can both add to this collection and consume features from the collection. For example, the `StaticFileMiddleware` accesses the `IHttpSendFileFeature` feature. If the feature exists, it's used to send the requested static file from its physical path. Otherwise, a slower alternative method is used to send the file. When available, the `IHttpSendFileFeature` allows the operating system to open the file and perform a direct kernel mode copy to the network card.

Additionally, middleware can add to the feature collection established by the server. Existing features can even be replaced by middleware, allowing the middleware to augment the functionality of the server. Features added to the collection are available immediately to other middleware or the underlying application itself later in the request pipeline.

By combining custom server implementations and specific middleware enhancements, the precise set of features an application requires can be constructed. This allows missing features to be added without requiring a change in server, and ensures only the minimal amount of features are exposed, thus limiting attack surface area and improving performance.

Summary

Feature interfaces define specific HTTP features that a given request may support. Servers define collections of features, and the initial set of features supported by that server, but middleware can be used to enhance these features.

Additional resources

- Servers
- Middleware
- Open Web Interface for .NET (OWIN)

Access HttpContext in ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

ASP.NET Core apps access `HttpContext` through the IHttpContextAccessor interface and its default implementation HttpContextAccessor. It's only necessary to use `IHttpContextAccessor` when you need access to the `HttpContext` inside a service.

Use HttpContext from Razor Pages

The Razor Pages PageModel exposes the HttpContext property:

```
public class AboutModel : PageModel
{
    public string Message { get; set; }

    public void OnGet()
    {
        Message = HttpContext.Request.PathBase;
    }
}
```

Use HttpContext from a Razor view

Razor views expose the `HttpContext` directly via a RazorPage.Context property on the view. The following example retrieves the current username in an intranet app using Windows Authentication:

```
@{
    var username = Context.User.Identity.Name;

    ...
}
```

Use HttpContext from a controller

Controllers expose the ControllerBase.HttpContext property:

```
public class HomeController : Controller
{
    public IActionResult About()
    {
        var pathBase = HttpContext.Request.PathBase;

        ...

        return View();
    }
}
```

Use HttpContext from middleware

When working with custom middleware components, `HttpContext` is passed into the `Invoke` or `InvokeAsync` method and can be accessed when the middleware is configured:

```
public class MyCustomMiddleware
{
    public Task InvokeAsync(HttpContext context)
    {
        ...
    }
}
```

Use HttpContext from custom components

For other framework and custom components that require access to `HttpContext`, the recommended approach is to register a dependency using the built-in dependency injection container. The dependency injection container supplies the `IHttpContextAccessor` to any classes that declare it as a dependency in their constructors:

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddControllersWithViews();
     services.AddHttpContextAccessor();
     services.AddTransient<IUserRepository, UserRepository>();
}
```

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddMvc()
         .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
     services.AddHttpContextAccessor();
     services.AddTransient<IUserRepository, UserRepository>();
}
```

In the following example:

- `UserRepository` declares its dependency on `IHttpContextAccessor`.
- The dependency is supplied when dependency injection resolves the dependency chain and creates an instance of `UserRepository`.

```
public class UserRepository : IUserRepository
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public UserRepository(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
    }

    public void LogCurrentUser()
    {
        var username = _httpContextAccessor.HttpContext.User.Identity.Name;
        service.LogAccessRequest(username);
    }
}
```

HttpContext access from a background thread

`HttpContext` isn't thread-safe. Reading or writing properties of the `HttpContext` outside of processing a request can result in a NullReferenceException.

> **NOTE**
>
> If your app generates sporadic `NullReferenceException` errors, review parts of the code that start background processing or that continue processing after a request completes. Look for mistakes, such as defining a controller method as `async void`.

To safely perform background work with `HttpContext` data:

- Copy the required data during request processing.
- Pass the copied data to a background task.

To avoid unsafe code, never pass the `HttpContext` into a method that performs background work. Pass the required data instead. In the following example, `SendEmailCore` is called to start sending an email. The `correlationId` is passed to `SendEmailCore`, not the `HttpContext`. Code execution doesn't wait for `SendEmailCore` to complete:

```
public class EmailController : Controller
{
    public IActionResult SendEmail(string email)
    {
        var correlationId = HttpContext.Request.Headers["x-correlation-id"].ToString();

        _ = SendEmailCore(correlationId);

        return View();
    }

    private async Task SendEmailCore(string correlationId)
    {
        ...
    }
}
```

Blazor and shared state

Blazor server apps live in server memory. That means that there are multiple apps hosted within the same process. For each app session, Blazor starts a circuit with its own DI container scope. That means that scoped services are unique per Blazor session.

> **WARNING**
>
> We don't recommend apps on the same server share state using singleton services unless extreme care is taken, as this can introduce security vulnerabilities, such as leaking user state across circuits.

You can use stateful singleton services in Blazor apps if they are specifically designed for it. For example, it's ok to use a memory cache as a singleton because it requires a key to access a given entry, assuming users don't have control of what cache keys are used.

Additionally, again for security reasons, you must not use IHttpContextAccessor within Blazor apps. Blazor apps run outside of the context of the ASP.NET Core pipeline. The HttpContext isn't guaranteed to be available within the IHttpContextAccessor, nor is it guaranteed to be holding the context that started the Blazor app.

The recommended way to pass request state to the Blazor app is through parameters to the root component in the initial rendering of the app:

- Define a class with all the data you want to pass to the Blazor app.

- Populate that data from the Razor page using the HttpContext available at that time.
- Pass the data to the Blazor app as a parameter to the root component (App).
- Define a parameter in the root component to hold the data being passed to the app.
- Use the user-specific data within the app; or alternatively, copy that data into a scoped service within OnInitializedAsync so that it can be used across the app.

For more information and example code, see ASP.NET Core Blazor Server additional security scenarios.

Detect changes with change tokens in ASP.NET Core

9/22/2020 • 17 minutes to read • Edit Online

A *change token* is a general-purpose, low-level building block used to track state changes.

View or download sample code (how to download)

IChangeToken interface

IChangeToken propagates notifications that a change has occurred. `IChangeToken` resides in the Microsoft.Extensions.Primitives namespace. The Microsoft.Extensions.Primitives NuGet package is implicitly provided to the ASP.NET Core apps.

`IChangeToken` has two properties:

- ActiveChangeCallbacks indicate if the token proactively raises callbacks. If `ActiveChangedCallbacks` is set to `false`, a callback is never called, and the app must poll `HasChanged` for changes. It's also possible for a token to never be cancelled if no changes occur or the underlying change listener is disposed or disabled.
- HasChanged receives a value that indicates if a change has occurred.

The `IChangeToken` interface includes the RegisterChangeCallback(Action<Object>, Object) method, which registers a callback that's invoked when the token has changed. `HasChanged` must be set before the callback is invoked.

ChangeToken class

ChangeToken is a static class used to propagate notifications that a change has occurred. `ChangeToken` resides in the Microsoft.Extensions.Primitives namespace. The Microsoft.Extensions.Primitives NuGet package is implicitly provided to the ASP.NET Core apps.

The ChangeToken.OnChange(Func<IChangeToken>, Action) method registers an `Action` to call whenever the token changes:

- `Func<IChangeToken>` produces the token.
- `Action` is called when the token changes.

The ChangeToken.OnChange<TState>(Func<IChangeToken>, Action<TState>, TState) overload takes an additional `TState` parameter that's passed into the token consumer `Action`.

`OnChange` returns an IDisposable. Calling Dispose stops the token from listening for further changes and releases the token's resources.

Example uses of change tokens in ASP.NET Core

Change tokens are used in prominent areas of ASP.NET Core to monitor for changes to objects:

- For monitoring changes to files, IFileProvider's Watch method creates an `IChangeToken` for the specified files or folder to watch.
- `IChangeToken` tokens can be added to cache entries to trigger cache evictions on change.
- For `TOptions` changes, the default OptionsMonitor<TOptions> implementation of IOptionsMonitor<TOptions> has an overload that accepts one or more IOptionsChangeTokenSource<TOptions> instances. Each instance returns an `IChangeToken` to register a

change notification callback for tracking options changes.

Monitor for configuration changes

By default, ASP.NET Core templates use JSON configuration files (*appsettings.json*, *appsettings.Development.json*, and *appsettings.Production.json*) to load app configuration settings.

These files are configured using the AddJsonFile(IConfigurationBuilder, String, Boolean, Boolean) extension method on ConfigurationBuilder that accepts a `reloadOnChange` parameter. `reloadOnChange` indicates if configuration should be reloaded on file changes. This setting appears in the Host convenience method CreateDefaultBuilder:

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
      .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
          reloadOnChange: true);
```

File-based configuration is represented by FileConfigurationSource. `FileConfigurationSource` uses IFileProvider to monitor files.

By default, the `IFileMonitor` is provided by a PhysicalFileProvider, which uses FileSystemWatcher to monitor for configuration file changes.

The sample app demonstrates two implementations for monitoring configuration changes. If any of the *appsettings* files change, both of the file monitoring implementations execute custom code—the sample app writes a message to the console.

A configuration file's `FileSystemWatcher` can trigger multiple token callbacks for a single configuration file change. To ensure that the custom code is only run once when multiple token callbacks are triggered, the sample's implementation checks file hashes. The sample uses SHA1 file hashing. A retry is implemented with an exponential back-off. The retry is present because file locking may occur that temporarily prevents computing a new hash on a file.

Utilities/Utilities.cs:

```
public static byte[] ComputeHash(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fs = File.OpenRead(filePath))
                {
                    return System.Security.Cryptography.SHA1
                        .Create().ComputeHash(fs);
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return new byte[20];
}
```

Simple startup change token

Register a token consumer `Action` callback for change notifications to the configuration reload token.

In `Startup.Configure`:

```
ChangeToken.OnChange(
    () => config.GetReloadToken(),
    (state) => InvokeChanged(state),
    env);
```

`config.GetReloadToken()` provides the token. The callback is the `InvokeChanged` method:

```
private void InvokeChanged(IWebHostEnvironment env)
{
    byte[] appsettingsHash = ComputeHash("appSettings.json");
    byte[] appsettingsEnvHash =
        ComputeHash($"appSettings.{env.EnvironmentName}.json");

    if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
        !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
    {
        _appsettingsHash = appsettingsHash;
        _appsettingsEnvHash = appsettingsEnvHash;

        WriteConsole("Configuration changed (Simple Startup Change Token)");
    }
}
```

The `state` of the callback is used to pass in the `IWebHostEnvironment`, which is useful for specifying the correct *appsettings* configuration file to monitor (for example, *appsettings.Development.json* when in the Development environment). File hashes are used to prevent the `WriteConsole` statement from running multiple times due to multiple token callbacks when the configuration file has only changed once.

This system runs as long as the app is running and can't be disabled by the user.

Monitor configuration changes as a service

The sample implements:

- Basic startup token monitoring.
- Monitoring as a service.
- A mechanism to enable and disable monitoring.

The sample establishes an `IConfigurationMonitor` interface.

Extensions/ConfigurationMonitor.cs:

```
public interface IConfigurationMonitor
{
    bool MonitoringEnabled { get; set; }
    string CurrentState { get; set; }
}
```

The constructor of the implemented class, `ConfigurationMonitor`, registers a callback for change notifications:

```
public ConfigurationMonitor(IConfiguration config, IWebHostEnvironment env)
{
    _env = env;

    ChangeToken.OnChange<IConfigurationMonitor>(
        () => config.GetReloadToken(),
        InvokeChanged,
        this);
}

public bool MonitoringEnabled { get; set; } = false;
public string CurrentState { get; set; } = "Not monitoring";
```

`config.GetReloadToken()` supplies the token. `InvokeChanged` is the callback method. The `state` in this instance is a reference to the `IConfigurationMonitor` instance that's used to access the monitoring state. Two properties are used:

- `MonitoringEnabled`: Indicates if the callback should run its custom code.
- `CurrentState`: Describes the current monitoring state for use in the UI.

The `InvokeChanged` method is similar to the earlier approach, except that it:

- Doesn't run its code unless `MonitoringEnabled` is `true`.
- Outputs the current `state` in its `WriteConsole` output.

```
private void InvokeChanged(IConfigurationMonitor state)
{
    if (MonitoringEnabled)
    {
        byte[] appsettingsHash = ComputeHash("appSettings.json");
        byte[] appsettingsEnvHash =
            ComputeHash($"appSettings.{_env.EnvironmentName}.json");

        if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
            !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
        {
            string message = $"State updated at {DateTime.Now}";


            _appsettingsHash = appsettingsHash;
            _appsettingsEnvHash = appsettingsEnvHash;

            WriteConsole("Configuration changed (ConfigurationMonitor Class) " +
                $"{message}, state:{state.CurrentState}");
        }
    }
}
```

An instance `ConfigurationMonitor` is registered as a service in `Startup.ConfigureServices`:

```
services.AddSingleton<IConfigurationMonitor, ConfigurationMonitor>();
```

The Index page offers the user control over configuration monitoring. The instance of `IConfigurationMonitor` is injected into the `IndexModel`.

Pages/Index.cshtml.cs:

```
public IndexModel(
    IConfiguration config,
    IConfigurationMonitor monitor,
    FileService fileService)
{
    _config = config;
    _monitor = monitor;
    _fileService = fileService;
}
```

The configuration monitor (`_monitor`) is used to enable or disable monitoring and set the current state for UI feedback:

```
public IActionResult OnPostStartMonitoring()
{
    _monitor.MonitoringEnabled = true;
    _monitor.CurrentState = "Monitoring!";

    return RedirectToPage();
}

public IActionResult OnPostStopMonitoring()
{
    _monitor.MonitoringEnabled = false;
    _monitor.CurrentState = "Not monitoring";

    return RedirectToPage();
}
```

When `OnPostStartMonitoring` is triggered, monitoring is enabled, and the current state is cleared. When `OnPostStopMonitoring` is triggered, monitoring is disabled, and the state is set to reflect that monitoring isn't occurring.

Buttons in the UI enable and disable monitoring.

Pages/Index.cshtml:

```
<button class="btn btn-success" asp-page-handler="StartMonitoring">
    Start Monitoring
</button>

<button class="btn btn-danger" asp-page-handler="StopMonitoring">
    Stop Monitoring
</button>
```

Monitor cached file changes

File content can be cached in-memory using IMemoryCache. In-memory caching is described in the Cache in-memory topic. Without taking additional steps, such as the implementation described below, *stale* (outdated) data is returned from a cache if the source data changes.

For example, not taking into account the status of a cached source file when renewing a sliding expiration period leads to stale cached file data. Each request for the data renews the sliding expiration period, but the file is never reloaded into the cache. Any app features that use the file's cached content are subject to possibly receiving stale content.

Using change tokens in a file caching scenario prevents the presence of stale file content in the cache. The sample app demonstrates an implementation of the approach.

The sample uses `GetFileContent` to:

- Return file content.
- Implement a retry algorithm with exponential back-off to cover cases where a file lock temporarily prevents reading a file.

Utilities/Utilities.cs:

```
public async static Task<string> GetFileContent(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fileStreamReader = File.OpenText(filePath))
                {
                    return await fileStreamReader.ReadToEndAsync();
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                await Task.Delay(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return null;
}
```

A `FileService` is created to handle cached file lookups. The `GetFileContent` method call of the service attempts to obtain file content from the in-memory cache and return it to the caller (*Services/FileService.cs*).

If cached content isn't found using the cache key, the following actions are taken:

1. The file content is obtained using `GetFileContent`.
2. A change token is obtained from the file provider with IFileProviders.Watch. The token's callback is triggered when the file is modified.
3. The file content is cached with a sliding expiration period. The change token is attached with MemoryCacheEntryExtensions.AddExpirationToken to evict the cache entry if the file changes while it's cached.

In the following example, files are stored in the app's content root. `IWebHostEnvironment.ContentRootFileProvider` is used to obtain an IFileProvider pointing at the app's `IWebHostEnvironment.ContentRootPath`. The `filePath` is obtained with IFileInfo.PhysicalPath.

```
public class FileService
{
    private readonly IMemoryCache _cache;
    private readonly IFileProvider _fileProvider;
    private List<string> _tokens = new List<string>();

    public FileService(IMemoryCache cache, IWebHostEnvironment env)
    {
        _cache = cache;
        _fileProvider = env.ContentRootFileProvider;
    }

    public async Task<string> GetFileContents(string fileName)
    {
        var filePath = _fileProvider.GetFileInfo(fileName).PhysicalPath;
        string fileContent;

        // Try to obtain the file contents from the cache.
        if (_cache.TryGetValue(filePath, out fileContent))
        {
            return fileContent;
        }

        // The cache doesn't have the entry, so obtain the file
        // contents from the file itself.
        fileContent = await GetFileContent(filePath);

        if (fileContent != null)
        {
            // Obtain a change token from the file provider whose
            // callback is triggered when the file is modified.
            var changeToken = _fileProvider.Watch(fileName);

            // Configure the cache entry options for a five minute
            // sliding expiration and use the change token to
            // expire the file in the cache if the file is
            // modified.
            var cacheEntryOptions = new MemoryCacheEntryOptions()
                .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                .AddExpirationToken(changeToken);

            // Put the file contents into the cache.
            _cache.Set(filePath, fileContent, cacheEntryOptions);

            return fileContent;
        }

        return string.Empty;
    }
}
```

The `FileService` is registered in the service container along with the memory caching service.

In `Startup.ConfigureServices`:

```
services.AddMemoryCache();
services.AddSingleton<FileService>();
```

The page model loads the file's content using the service.

In the Index page's `OnGet` method (*Pages/Index.cshtml.cs*):

```
var fileContent = await _fileService.GetFileContents("poem.txt");
```

CompositeChangeToken class

For representing one or more `IChangeToken` instances in a single object, use the CompositeChangeToken class.

```
var firstCancellationTokenSource = new CancellationTokenSource();
var secondCancellationTokenSource = new CancellationTokenSource();

var firstCancellationToken = firstCancellationTokenSource.Token;
var secondCancellationToken = secondCancellationTokenSource.Token;

var firstCancellationChangeToken = new CancellationChangeToken(firstCancellationToken);
var secondCancellationChangeToken = new CancellationChangeToken(secondCancellationToken);

var compositeChangeToken =
    new CompositeChangeToken(
        new List<IChangeToken>
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken
        });
```

`HasChanged` on the composite token reports `true` if any represented token `HasChanged` is `true`. `ActiveChangeCallbacks` on the composite token reports `true` if any represented token `ActiveChangeCallbacks` is `true`. If multiple concurrent change events occur, the composite change callback is invoked one time.

A *change token* is a general-purpose, low-level building block used to track state changes.

View or download sample code (how to download)

IChangeToken interface

IChangeToken propagates notifications that a change has occurred. `IChangeToken` resides in the Microsoft.Extensions.Primitives namespace. For apps that don't use the Microsoft.AspNetCore.App metapackage, create a package reference for the Microsoft.Extensions.Primitives NuGet package.

`IChangeToken` has two properties:

- ActiveChangeCallbacks indicate if the token proactively raises callbacks. If `ActiveChangedCallbacks` is set to `false`, a callback is never called, and the app must poll `HasChanged` for changes. It's also possible for a token to never be cancelled if no changes occur or the underlying change listener is disposed or disabled.
- HasChanged receives a value that indicates if a change has occurred.

The `IChangeToken` interface includes the RegisterChangeCallback(Action<Object>, Object) method, which registers a callback that's invoked when the token has changed. `HasChanged` must be set before the callback is invoked.

ChangeToken class

ChangeToken is a static class used to propagate notifications that a change has occurred. `ChangeToken` resides in the Microsoft.Extensions.Primitives namespace. For apps that don't use the Microsoft.AspNetCore.App metapackage, create a package reference for the Microsoft.Extensions.Primitives NuGet package.

The ChangeToken.OnChange(Func<IChangeToken>, Action) method registers an `Action` to call whenever the token changes:

- `Func<IChangeToken>` produces the token.
- `Action` is called when the token changes.

The ChangeToken.OnChange<TState>(Func<IChangeToken>, Action<TState>, TState) overload takes an additional `TState` parameter that's passed into the token consumer `Action`.

`OnChange` returns an IDisposable. Calling Dispose stops the token from listening for further changes and releases the token's resources.

Example uses of change tokens in ASP.NET Core

Change tokens are used in prominent areas of ASP.NET Core to monitor for changes to objects:

- For monitoring changes to files, IFileProvider's Watch method creates an `IChangeToken` for the specified files or folder to watch.
- `IChangeToken` tokens can be added to cache entries to trigger cache evictions on change.
- For `TOptions` changes, the default OptionsMonitor<TOptions> implementation of IOptionsMonitor<TOptions> has an overload that accepts one or more IOptionsChangeTokenSource<TOptions> instances. Each instance returns an `IChangeToken` to register a change notification callback for tracking options changes.

Monitor for configuration changes

By default, ASP.NET Core templates use JSON configuration files (*appsettings.json*, *appsettings.Development.json*, and *appsettings.Production.json*) to load app configuration settings.

These files are configured using the AddJsonFile(IConfigurationBuilder, String, Boolean, Boolean) extension method on ConfigurationBuilder that accepts a `reloadOnChange` parameter. `reloadOnChange` indicates if configuration should be reloaded on file changes. This setting appears in the WebHost convenience method CreateDefaultBuilder:

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
      .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
          reloadOnChange: true);
```

File-based configuration is represented by FileConfigurationSource. `FileConfigurationSource` uses IFileProvider to monitor files.

By default, the `IFileMonitor` is provided by a PhysicalFileProvider, which uses FileSystemWatcher to monitor for configuration file changes.

The sample app demonstrates two implementations for monitoring configuration changes. If any of the *appsettings* files change, both of the file monitoring implementations execute custom code—the sample app writes a message to the console.

A configuration file's `FileSystemWatcher` can trigger multiple token callbacks for a single configuration file change. To ensure that the custom code is only run once when multiple token callbacks are triggered, the sample's implementation checks file hashes. The sample uses SHA1 file hashing. A retry is implemented with an exponential back-off. The retry is present because file locking may occur that temporarily prevents computing a new hash on a file.

Utilities/Utilities.cs:

```
public static byte[] ComputeHash(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fs = File.OpenRead(filePath))
                {
                    return System.Security.Cryptography.SHA1
                        .Create().ComputeHash(fs);
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return new byte[20];
}
```

Simple startup change token

Register a token consumer `Action` callback for change notifications to the configuration reload token.

In `Startup.Configure`:

```
ChangeToken.OnChange(
    () => config.GetReloadToken(),
    (state) => InvokeChanged(state),
    env);
```

`config.GetReloadToken()` provides the token. The callback is the `InvokeChanged` method:

```
private void InvokeChanged(IHostingEnvironment env)
{
    byte[] appsettingsHash = ComputeHash("appSettings.json");
    byte[] appsettingsEnvHash =
        ComputeHash($"appSettings.{env.EnvironmentName}.json");

    if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
        !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
    {
        _appsettingsHash = appsettingsHash;
        _appsettingsEnvHash = appsettingsEnvHash;

        WriteConsole("Configuration changed (Simple Startup Change Token)");
    }
}
```

The `state` of the callback is used to pass in the `IHostingEnvironment`, which is useful for specifying the correct *appsettings* configuration file to monitor (for example, *appsettings.Development.json* when in the Development environment). File hashes are used to prevent the `WriteConsole` statement from running multiple times due to multiple token callbacks when the configuration file has only changed once.

This system runs as long as the app is running and can't be disabled by the user.

Monitor configuration changes as a service

The sample implements:

- Basic startup token monitoring.
- Monitoring as a service.
- A mechanism to enable and disable monitoring.

The sample establishes an `IConfigurationMonitor` interface.

Extensions/ConfigurationMonitor.cs:

```
public interface IConfigurationMonitor
{
    bool MonitoringEnabled { get; set; }
    string CurrentState { get; set; }
}
```

The constructor of the implemented class, `ConfigurationMonitor`, registers a callback for change notifications:

```
public ConfigurationMonitor(IConfiguration config, IHostingEnvironment env)
{
    _env = env;

    ChangeToken.OnChange<IConfigurationMonitor>(
        () => config.GetReloadToken(),
        InvokeChanged,
        this);
}

public bool MonitoringEnabled { get; set; } = false;
public string CurrentState { get; set; } = "Not monitoring";
```

`config.GetReloadToken()` supplies the token. `InvokeChanged` is the callback method. The `state` in this instance is a reference to the `IConfigurationMonitor` instance that's used to access the monitoring state. Two properties are used:

- `MonitoringEnabled`: Indicates if the callback should run its custom code.
- `CurrentState`: Describes the current monitoring state for use in the UI.

The `InvokeChanged` method is similar to the earlier approach, except that it:

- Doesn't run its code unless `MonitoringEnabled` is `true`.
- Outputs the current `state` in its `WriteConsole` output.

```
private void InvokeChanged(IConfigurationMonitor state)
{
    if (MonitoringEnabled)
    {
        byte[] appsettingsHash = ComputeHash("appSettings.json");
        byte[] appsettingsEnvHash =
            ComputeHash($"appSettings.{_env.EnvironmentName}.json");

        if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
            !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
        {
            string message = $"State updated at {DateTime.Now}";


            _appsettingsHash = appsettingsHash;
            _appsettingsEnvHash = appsettingsEnvHash;

            WriteConsole("Configuration changed (ConfigurationMonitor Class) " +
                $"{message}, state:{state.CurrentState}");
        }
    }
}
```

An instance `ConfigurationMonitor` is registered as a service in `Startup.ConfigureServices`:

```
services.AddSingleton<IConfigurationMonitor, ConfigurationMonitor>();
```

The Index page offers the user control over configuration monitoring. The instance of `IConfigurationMonitor` is injected into the `IndexModel`.

Pages/Index.cshtml.cs:

```
public IndexModel(
    IConfiguration config,
    IConfigurationMonitor monitor,
    FileService fileService)
{
    _config = config;
    _monitor = monitor;
    _fileService = fileService;
}
```

The configuration monitor (`_monitor`) is used to enable or disable monitoring and set the current state for UI feedback:

```
public IActionResult OnPostStartMonitoring()
{
    _monitor.MonitoringEnabled = true;
    _monitor.CurrentState = "Monitoring!";

    return RedirectToPage();
}

public IActionResult OnPostStopMonitoring()
{
    _monitor.MonitoringEnabled = false;
    _monitor.CurrentState = "Not monitoring";

    return RedirectToPage();
}
```

When `OnPostStartMonitoring` is triggered, monitoring is enabled, and the current state is cleared. When `OnPostStopMonitoring` is triggered, monitoring is disabled, and the state is set to reflect that monitoring isn't occurring.

Buttons in the UI enable and disable monitoring.

Pages/Index.cshtml:

```
<button class="btn btn-success" asp-page-handler="StartMonitoring">
    Start Monitoring
</button>

<button class="btn btn-danger" asp-page-handler="StopMonitoring">
    Stop Monitoring
</button>
```

Monitor cached file changes

File content can be cached in-memory using IMemoryCache. In-memory caching is described in the Cache in-memory topic. Without taking additional steps, such as the implementation described below, *stale* (outdated) data is returned from a cache if the source data changes.

For example, not taking into account the status of a cached source file when renewing a sliding expiration period leads to stale cached file data. Each request for the data renews the sliding expiration period, but the file is never reloaded into the cache. Any app features that use the file's cached content are subject to possibly receiving stale content.

Using change tokens in a file caching scenario prevents the presence of stale file content in the cache. The sample app demonstrates an implementation of the approach.

The sample uses `GetFileContent` to:

- Return file content.
- Implement a retry algorithm with exponential back-off to cover cases where a file lock temporarily prevents reading a file.

Utilities/Utilities.cs:

```
public async static Task<string> GetFileContent(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fileStreamReader = File.OpenText(filePath))
                {
                    return await fileStreamReader.ReadToEndAsync();
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3 || ex.HResult != -2147024864)
            {
                throw;
            }
            else
            {
                await Task.Delay(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
                runCount++;
            }
        }
    }

    return null;
}
```

A `FileService` is created to handle cached file lookups. The `GetFileContent` method call of the service attempts to obtain file content from the in-memory cache and return it to the caller (*Services/FileService.cs*).

If cached content isn't found using the cache key, the following actions are taken:

1. The file content is obtained using `GetFileContent`.
2. A change token is obtained from the file provider with IFileProviders.Watch. The token's callback is triggered when the file is modified.
3. The file content is cached with a sliding expiration period. The change token is attached with MemoryCacheEntryExtensions.AddExpirationToken to evict the cache entry if the file changes while it's cached.

In the following example, files are stored in the app's content root. IHostingEnvironment.ContentRootFileProvider is used to obtain an IFileProvider pointing at the app's ContentRootPath. The `filePath` is obtained with IFileInfo.PhysicalPath.

```
public class FileService
{
    private readonly IMemoryCache _cache;
    private readonly IFileProvider _fileProvider;
    private List<string> _tokens = new List<string>();

    public FileService(IMemoryCache cache, IHostingEnvironment env)
    {
        _cache = cache;
        _fileProvider = env.ContentRootFileProvider;
    }

    public async Task<string> GetFileContents(string fileName)
    {
        var filePath = _fileProvider.GetFileInfo(fileName).PhysicalPath;
        string fileContent;

        // Try to obtain the file contents from the cache.
        if (_cache.TryGetValue(filePath, out fileContent))
        {
            return fileContent;
        }

        // The cache doesn't have the entry, so obtain the file
        // contents from the file itself.
        fileContent = await GetFileContent(filePath);

        if (fileContent != null)
        {
            // Obtain a change token from the file provider whose
            // callback is triggered when the file is modified.
            var changeToken = _fileProvider.Watch(fileName);

            // Configure the cache entry options for a five minute
            // sliding expiration and use the change token to
            // expire the file in the cache if the file is
            // modified.
            var cacheEntryOptions = new MemoryCacheEntryOptions()
                .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                .AddExpirationToken(changeToken);

            // Put the file contents into the cache.
            _cache.Set(filePath, fileContent, cacheEntryOptions);

            return fileContent;
        }

        return string.Empty;
    }
}
```

The `FileService` is registered in the service container along with the memory caching service.

In `Startup.ConfigureServices`:

```
services.AddMemoryCache();
services.AddSingleton<FileService>();
```

The page model loads the file's content using the service.

In the Index page's `OnGet` method (*Pages/Index.cshtml.cs*):

```
var fileContent = await _fileService.GetFileContents("poem.txt");
```

CompositeChangeToken class

For representing one or more `IChangeToken` instances in a single object, use the CompositeChangeToken class.

```
var firstCancellationTokenSource = new CancellationTokenSource();
var secondCancellationTokenSource = new CancellationTokenSource();

var firstCancellationToken = firstCancellationTokenSource.Token;
var secondCancellationToken = secondCancellationTokenSource.Token;

var firstCancellationChangeToken = new CancellationChangeToken(firstCancellationToken);
var secondCancellationChangeToken = new CancellationChangeToken(secondCancellationToken);

var compositeChangeToken =
    new CompositeChangeToken(
        new List<IChangeToken>
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken
        });
```

`HasChanged` on the composite token reports `true` if any represented token `HasChanged` is `true`. `ActiveChangeCallbacks` on the composite token reports `true` if any represented token `ActiveChangeCallbacks` is `true`. If multiple concurrent change events occur, the composite change callback is invoked one time.

Additional resources

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Open Web Interface for .NET (OWIN) with ASP.NET Core

9/22/2020 • 3 minutes to read • Edit Online

By Steve Smith and Rick Anderson

ASP.NET Core supports the Open Web Interface for .NET (OWIN). OWIN allows web apps to be decoupled from web servers. It defines a standard way for middleware to be used in a pipeline to handle requests and associated responses. ASP.NET Core applications and middleware can interoperate with OWIN-based applications, servers, and middleware.

OWIN provides a decoupling layer that allows two frameworks with disparate object models to be used together. The `Microsoft.AspNetCore.Owin` package provides two adapter implementations:

- ASP.NET Core to OWIN
- OWIN to ASP.NET Core

This allows ASP.NET Core to be hosted on top of an OWIN compatible server/host or for other OWIN compatible components to be run on top of ASP.NET Core.

> **NOTE**
> Using these adapters comes with a performance cost. Apps using only ASP.NET Core components shouldn't use the `Microsoft.AspNetCore.Owin` package or adapters.

View or download sample code (how to download)

Running OWIN middleware in the ASP.NET Core pipeline

ASP.NET Core's OWIN support is deployed as part of the `Microsoft.AspNetCore.Owin` package. You can import OWIN support into your project by installing this package.

OWIN middleware conforms to the OWIN specification, which requires a `Func<IDictionary<string, object>, Task>` interface, and specific keys be set (such as `owin.ResponseBody`). The following simple OWIN middleware displays "Hello World":

```
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);

    // OWIN Environment Keys: https://owin.org/spec/spec/owin-1.0.0.html
    var responseStream = (Stream)environment["owin.ResponseBody"];
    var responseHeaders = (IDictionary<string, string[]>)environment["owin.ResponseHeaders"];

    responseHeaders["Content-Length"] = new string[] {
responseBytes.Length.ToString(CultureInfo.InvariantCulture) };
    responseHeaders["Content-Type"] = new string[] { "text/plain" };

    return responseStream.WriteAsync(responseBytes, 0, responseBytes.Length);
}
```

The sample signature returns a `Task` and accepts an `IDictionary<string, object>` as required by OWIN.

The following code shows how to add the `OwinHello` middleware (shown above) to the ASP.NET Core pipeline with the `UseOwin` extension method.

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

You can configure other actions to take place within the OWIN pipeline.

> **NOTE**
> Response headers should only be modified prior to the first write to the response stream.

> **NOTE**
> Multiple calls to `UseOwin` is discouraged for performance reasons. OWIN components will operate best if grouped together.

```
app.UseOwin(pipeline =>
{
    pipeline(next =>
    {
        return async environment =>
        {
            // Do something before.
            await next(environment);
            // Do something after.
        };
    });
});
```

Run ASP.NET Core on an OWIN-based server and use its WebSockets support

Another example of how OWIN-based servers' features can be leveraged by ASP.NET Core is access to features like WebSockets. The .NET OWIN web server used in the previous example has support for Web Sockets built in, which can be leveraged by an ASP.NET Core application. The example below shows a simple web app that supports Web Sockets and echoes back everything sent to the server through WebSockets.

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            if (context.WebSockets.IsWebSocketRequest)
            {
                WebSocket webSocket = await context.WebSockets.AcceptWebSocketAsync();
                await EchoWebSocket(webSocket);
            }
            else
            {
                await next();
            }
        });

        app.Run(context =>
        {
            return context.Response.WriteAsync("Hello World");
        });
    }

    private async Task EchoWebSocket(WebSocket webSocket)
    {
        byte[] buffer = new byte[1024];
        WebSocketReceiveResult received = await webSocket.ReceiveAsync(
            new ArraySegment<byte>(buffer), CancellationToken.None);

        while (!webSocket.CloseStatus.HasValue)
        {
            // Echo anything we receive
            await webSocket.SendAsync(new ArraySegment<byte>(buffer, 0, received.Count),
                received.MessageType, received.EndOfMessage, CancellationToken.None);

            received = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer),
                CancellationToken.None);
        }

        await webSocket.CloseAsync(webSocket.CloseStatus.Value,
            webSocket.CloseStatusDescription, CancellationToken.None);
    }
}
```

OWIN environment

You can construct an OWIN environment using the `HttpContext`.

```
var environment = new OwinEnvironment(HttpContext);
var features = new OwinFeatureCollection(environment);
```

OWIN keys

OWIN depends on an `IDictionary<string,object>` object to communicate information throughout an HTTP Request/Response exchange. ASP.NET Core implements the keys listed below. See the primary specification, extensions, and OWIN Key Guidelines and Common Keys.

Request data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestScheme	`String`	
owin.RequestMethod	`String`	
owin.RequestPathBase	`String`	
owin.RequestPath	`String`	
owin.RequestQueryString	`String`	
owin.RequestProtocol	`String`	
owin.RequestHeaders	`IDictionary<string,string[]>`	
owin.RequestBody	`Stream`	

Request data (OWIN v1.1.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.RequestId	`String`	Optional

Response data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.ResponseStatusCode	`int`	Optional
owin.ResponseReasonPhrase	`String`	Optional
owin.ResponseHeaders	`IDictionary<string,string[]>`	
owin.ResponseBody	`Stream`	

Other data (OWIN v1.0.0)

KEY	VALUE (TYPE)	DESCRIPTION
owin.CallCancelled	`CancellationToken`	
owin.Version	`String`	

Common keys

KEY	VALUE (TYPE)	DESCRIPTION
ssl.ClientCertificate	`X509Certificate`	
ssl.LoadClientCertAsync	`Func<Task>`	

KEY	VALUE (TYPE)	DESCRIPTION
server.RemoteIpAddress	`String`	
server.RemotePort	`String`	
server.LocalIpAddress	`String`	
server.LocalPort	`String`	
server.IsLocal	`bool`	
server.OnSendingHeaders	`Action<Action<object>,object>`	

SendFiles v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
sendfile.SendAsync	See delegate signature	Per Request

Opaque v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
opaque.Version	`String`	
opaque.Upgrade	`OpaqueUpgrade`	See delegate signature
opaque.Stream	`Stream`	
opaque.CallCancelled	`CancellationToken`	

WebSocket v0.3.0

KEY	VALUE (TYPE)	DESCRIPTION
websocket.Version	`String`	
websocket.Accept	`WebSocketAccept`	See delegate signature
websocket.AcceptAlt		Non-spec
websocket.SubProtocol	`String`	See RFC6455 Section 4.2.2 Step 5.5
websocket.SendAsync	`WebSocketSendAsync`	See delegate signature
websocket.ReceiveAsync	`WebSocketReceiveAsync`	See delegate signature
websocket.CloseAsync	`WebSocketCloseAsync`	See delegate signature
websocket.CallCancelled	`CancellationToken`	

KEY	VALUE (TYPE)	DESCRIPTION
websocket.ClientCloseStatus	`int`	Optional
websocket.ClientCloseDescription	`String`	Optional

Additional resources

- Middleware
- Servers

Background tasks with hosted services in ASP.NET Core

9/22/2020 • 14 minutes to read • Edit Online

By Jeow Li Huan

In ASP.NET Core, background tasks can be implemented as *hosted services*. A hosted service is a class with background task logic that implements the IHostedService interface. This topic provides three hosted service examples:

- Background task that runs on a timer.
- Hosted service that activates a scoped service. The scoped service can use dependency injection (DI).
- Queued background tasks that run sequentially.

View or download sample code (how to download)

Worker Service template

The ASP.NET Core Worker Service template provides a starting point for writing long running service apps. An app created from the Worker Service template specifies the Worker SDK in its project file:

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

To use the template as a basis for a hosted services app:

- Visual Studio
- Visual Studio for Mac
- .NET Core CLI

1. Create a new project.
2. Select **Worker Service**. Select **Next**.
3. Provide a project name in the **Project name** field or accept the default project name. Select **Create**.
4. In the **Create a new Worker service** dialog, select **Create**.

Package

An app based on the Worker Service template uses the `Microsoft.NET.Sdk.Worker` SDK and has an explicit package reference to the Microsoft.Extensions.Hosting package. For example, see the sample app's project file (*BackgroundTasksSample.csproj*).

For web apps that use the `Microsoft.NET.Sdk.Web` SDK, the Microsoft.Extensions.Hosting package is referenced implicitly from the shared framework. An explicit package reference in the app's project file isn't required.

IHostedService interface

The IHostedService interface defines two methods for objects that are managed by the host:

- StartAsync(CancellationToken): `StartAsync` contains the logic to start the background task. `StartAsync` is called *before*:

- The app's request processing pipeline is configured (`Startup.Configure`).
- The server is started and IApplicationLifetime.ApplicationStarted is triggered.

The default behavior can be changed so that the hosted service's `StartAsync` runs after the app's pipeline has been configured and `ApplicationStarted` is called. To change the default behavior, add the hosted service (`VideosWatcher` in the following example) after calling `ConfigureWebHostDefaults`:

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            })
            .ConfigureServices(services =>
            {
                services.AddHostedService<VideosWatcher>();
            });
}
```

- StopAsync(CancellationToken): Triggered when the host is performing a graceful shutdown. `StopAsync` contains the logic to end the background task. Implement IDisposable and finalizers (destructors) to dispose of any unmanaged resources.

  The cancellation token has a default five second timeout to indicate that the shutdown process should no longer be graceful. When cancellation is requested on the token:

  - Any remaining background operations that the app is performing should be aborted.
  - Any methods called in `StopAsync` should return promptly.

  However, tasks aren't abandoned after cancellation is requested—the caller awaits all tasks to complete.

  If the app shuts down unexpectedly (for example, the app's process fails), `StopAsync` might not be called. Therefore, any methods called or operations conducted in `StopAsync` might not occur.

  To extend the default five second shutdown timeout, set:

  - ShutdownTimeout when using Generic Host. For more information, see .NET Generic Host.
  - Shutdown timeout host configuration setting when using Web Host. For more information, see ASP.NET Core Web Host.

The hosted service is activated once at app startup and gracefully shut down at app shutdown. If an error is thrown during background task execution, `Dispose` should be called even if `StopAsync` isn't called.

BackgroundService base class

BackgroundService is a base class for implementing a long running IHostedService.

ExecuteAsync(CancellationToken) is called to run the background service. The implementation returns a Task that represents the entire lifetime of the background service. No further services are started until

ExecuteAsync becomes asynchronous, such as by calling `await`. Avoid performing long, blocking initialization work in `ExecuteAsync`. The host blocks in StopAsync(CancellationToken) waiting for `ExecuteAsync` to complete.

The cancellation token is triggered when IHostedService.StopAsync is called. Your implementation of `ExecuteAsync` should finish promptly when the cancellation token is fired in order to gracefully shut down the service. Otherwise, the service ungracefully shuts down at the shutdown timeout. For more information, see the IHostedService interface section.

Timed background tasks

A timed background task makes use of the System.Threading.Timer class. The timer triggers the task's `DoWork` method. The timer is disabled on `StopAsync` and disposed when the service container is disposed on `Dispose`:

```
public class TimedHostedService : IHostedService, IDisposable
{
    private int executionCount = 0;
    private readonly ILogger<TimedHostedService> _logger;
    private Timer _timer;

    public TimedHostedService(ILogger<TimedHostedService> logger)
    {
        _logger = logger;
    }

    public Task StartAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service running.");

        _timer = new Timer(DoWork, null, TimeSpan.Zero,
            TimeSpan.FromSeconds(5));

        return Task.CompletedTask;
    }

    private void DoWork(object state)
    {
        var count = Interlocked.Increment(ref executionCount);

        _logger.LogInformation(
            "Timed Hosted Service is working. Count: {Count}", count);
    }

    public Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service is stopping.");

        _timer?.Change(Timeout.Infinite, 0);

        return Task.CompletedTask;
    }

    public void Dispose()
    {
        _timer?.Dispose();
    }
}
```

The Timer doesn't wait for previous executions of `DoWork` to finish, so the approach shown might not be suitable for every scenario. Interlocked.Increment is used to increment the execution counter as an atomic operation, which ensures that multiple threads don't update `executionCount` concurrently.

The service is registered in `IHostBuilder.ConfigureServices` (*Program.cs*) with the `AddHostedService` extension method:

```
services.AddHostedService<TimedHostedService>();
```

Consuming a scoped service in a background task

To use scoped services within a BackgroundService, create a scope. No scope is created for a hosted service by default.

The scoped background task service contains the background task's logic. In the following example:

- The service is asynchronous. The `DoWork` method returns a `Task`. For demonstration purposes, a delay of ten seconds is awaited in the `DoWork` method.
- An ILogger is injected into the service.

```
internal interface IScopedProcessingService
{
    Task DoWork(CancellationToken stoppingToken);
}

internal class ScopedProcessingService : IScopedProcessingService
{
    private int executionCount = 0;
    private readonly ILogger _logger;

    public ScopedProcessingService(ILogger<ScopedProcessingService> logger)
    {
        _logger = logger;
    }

    public async Task DoWork(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            executionCount++;

            _logger.LogInformation(
                "Scoped Processing Service is working. Count: {Count}", executionCount);

            await Task.Delay(10000, stoppingToken);
        }
    }
}
```

The hosted service creates a scope to resolve the scoped background task service to call its `DoWork` method. `DoWork` returns a `Task`, which is awaited in `ExecuteAsync`:

```
public class ConsumeScopedServiceHostedService : BackgroundService
{
    private readonly ILogger<ConsumeScopedServiceHostedService> _logger;

    public ConsumeScopedServiceHostedService(IServiceProvider services,
        ILogger<ConsumeScopedServiceHostedService> logger)
    {
        Services = services;
        _logger = logger;
    }

    public IServiceProvider Services { get; }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service running.");

        await DoWork(stoppingToken);
    }

    private async Task DoWork(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is working.");

        using (var scope = Services.CreateScope())
        {
            var scopedProcessingService =
                scope.ServiceProvider
                    .GetRequiredService<IScopedProcessingService>();

            await scopedProcessingService.DoWork(stoppingToken);
        }
    }

    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is stopping.");

        await base.StopAsync(stoppingToken);
    }
}
```

The services are registered in `IHostBuilder.ConfigureServices` (*Program.cs*). The hosted service is registered with the `AddHostedService` extension method:

```
services.AddHostedService<ConsumeScopedServiceHostedService>();
services.AddScoped<IScopedProcessingService, ScopedProcessingService>();
```

Queued background tasks

A background task queue is based on the .NET 4.x QueueBackgroundWorkItem:

```
public interface IBackgroundTaskQueue
{
    void QueueBackgroundWorkItem(Func<CancellationToken, Task> workItem);

    Task<Func<CancellationToken, Task>> DequeueAsync(
        CancellationToken cancellationToken);
}

public class BackgroundTaskQueue : IBackgroundTaskQueue
{
    private ConcurrentQueue<Func<CancellationToken, Task>> _workItems =
        new ConcurrentQueue<Func<CancellationToken, Task>>();
    private SemaphoreSlim _signal = new SemaphoreSlim(0);

    public void QueueBackgroundWorkItem(
        Func<CancellationToken, Task> workItem)
    {
        if (workItem == null)
        {
            throw new ArgumentNullException(nameof(workItem));
        }

        _workItems.Enqueue(workItem);
        _signal.Release();
    }

    public async Task<Func<CancellationToken, Task>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        await _signal.WaitAsync(cancellationToken);
        _workItems.TryDequeue(out var workItem);

        return workItem;
    }
}
```

In the following `QueueHostedService` example:

- The `BackgroundProcessing` method returns a `Task`, which is awaited in `ExecuteAsync`.
- Background tasks in the queue are dequeued and executed in `BackgroundProcessing`.
- Work items are awaited before the service stops in `StopAsync`.

```
public class QueuedHostedService : BackgroundService
{
    private readonly ILogger<QueuedHostedService> _logger;

    public QueuedHostedService(IBackgroundTaskQueue taskQueue,
        ILogger<QueuedHostedService> logger)
    {
        TaskQueue = taskQueue;
        _logger = logger;
    }

    public IBackgroundTaskQueue TaskQueue { get; }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            $"Queued Hosted Service is running.{Environment.NewLine}" +
            $"{Environment.NewLine}Tap W to add a work item to the " +
            $"background queue.{Environment.NewLine}");

        await BackgroundProcessing(stoppingToken);
    }

    private async Task BackgroundProcessing(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            var workItem =
                await TaskQueue.DequeueAsync(stoppingToken);

            try
            {
                await workItem(stoppingToken);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex,
                    "Error occurred executing {WorkItem}.", nameof(workItem));
            }
        }
    }

    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Queued Hosted Service is stopping.");

        await base.StopAsync(stoppingToken);
    }
}
```

A `MonitorLoop` service handles enqueuing tasks for the hosted service whenever the `w` key is selected on an input device:

- The `IBackgroundTaskQueue` is injected into the `MonitorLoop` service.
- `IBackgroundTaskQueue.QueueBackgroundWorkItem` is called to enqueue a work item.
- The work item simulates a long-running background task:
  - Three 5-second delays are executed (`Task.Delay`).
  - A `try-catch` statement traps OperationCanceledException if the task is cancelled.

```
public class MonitorLoop
{
    private readonly IBackgroundTaskQueue _taskQueue;
    private readonly ILogger _logger;
```

```
    private readonly CancellationToken _cancellationToken;

    public MonitorLoop(IBackgroundTaskQueue taskQueue,
        ILogger<MonitorLoop> logger,
        IHostApplicationLifetime applicationLifetime)
    {
        _taskQueue = taskQueue;
        _logger = logger;
        _cancellationToken = applicationLifetime.ApplicationStopping;
    }

    public void StartMonitorLoop()
    {
        _logger.LogInformation("Monitor Loop is starting.");

        // Run a console user input loop in a background thread
        Task.Run(() => Monitor());
    }

    public void Monitor()
    {
        while (!_cancellationToken.IsCancellationRequested)
        {
            var keyStroke = Console.ReadKey();

            if (keyStroke.Key == ConsoleKey.W)
            {
                // Enqueue a background work item
                _taskQueue.QueueBackgroundWorkItem(async token =>
                {
                    // Simulate three 5-second tasks to complete
                    // for each enqueued work item

                    int delayLoop = 0;
                    var guid = Guid.NewGuid().ToString();

                    _logger.LogInformation(
                        "Queued Background Task {Guid} is starting.", guid);

                    while (!token.IsCancellationRequested && delayLoop < 3)
                    {
                        try
                        {
                            await Task.Delay(TimeSpan.FromSeconds(5), token);
                        }
                        catch (OperationCanceledException)
                        {
                            // Prevent throwing if the Delay is cancelled
                        }

                        delayLoop++;

                        _logger.LogInformation(
                            "Queued Background Task {Guid} is running. " +
                            "{DelayLoop}/3", guid, delayLoop);
                    }

                    if (delayLoop == 3)
                    {
                        _logger.LogInformation(
                            "Queued Background Task {Guid} is complete.", guid);
                    }
                    else
                    {
                        _logger.LogInformation(
                            "Queued Background Task {Guid} was cancelled.", guid);
                    }
                });
            }
```

```
        }
    }
}
```

The services are registered in `IHostBuilder.ConfigureServices` (*Program.cs*). The hosted service is registered with the `AddHostedService` extension method:

```
services.AddSingleton<MonitorLoop>();
services.AddHostedService<QueuedHostedService>();
services.AddSingleton<IBackgroundTaskQueue, BackgroundTaskQueue>();
```

`MonitorLoop` is started in `Program.Main`:

```
var monitorLoop = host.Services.GetRequiredService<MonitorLoop>();
monitorLoop.StartMonitorLoop();
```

In ASP.NET Core, background tasks can be implemented as *hosted services*. A hosted service is a class with background task logic that implements the IHostedService interface. This topic provides three hosted service examples:

- Background task that runs on a timer.
- Hosted service that activates a scoped service. The scoped service can use dependency injection (DI)
- Queued background tasks that run sequentially.

View or download sample code (how to download)

Package

Reference the Microsoft.AspNetCore.App metapackage or add a package reference to the Microsoft.Extensions.Hosting package.

IHostedService interface

Hosted services implement the IHostedService interface. The interface defines two methods for objects that are managed by the host:

- StartAsync(CancellationToken): `StartAsync` contains the logic to start the background task. When using the Web Host, `StartAsync` is called after the server has started and IApplicationLifetime.ApplicationStarted is triggered. When using the Generic Host, `StartAsync` is called before `ApplicationStarted` is triggered.

- StopAsync(CancellationToken): Triggered when the host is performing a graceful shutdown. `StopAsync` contains the logic to end the background task. Implement IDisposable and finalizers (destructors) to dispose of any unmanaged resources.

  The cancellation token has a default five second timeout to indicate that the shutdown process should no longer be graceful. When cancellation is requested on the token:

  - Any remaining background operations that the app is performing should be aborted.
  - Any methods called in `StopAsync` should return promptly.

  However, tasks aren't abandoned after cancellation is requested—the caller awaits all tasks to complete.

  If the app shuts down unexpectedly (for example, the app's process fails), `StopAsync` might not be called. Therefore, any methods called or operations conducted in `StopAsync` might not occur.

To extend the default five second shutdown timeout, set:

- ShutdownTimeout when using Generic Host. For more information, see .NET Generic Host.
- Shutdown timeout host configuration setting when using Web Host. For more information, see ASP.NET Core Web Host.

The hosted service is activated once at app startup and gracefully shut down at app shutdown. If an error is thrown during background task execution, `Dispose` should be called even if `StopAsync` isn't called.

Timed background tasks

A timed background task makes use of the System.Threading.Timer class. The timer triggers the task's `DoWork` method. The timer is disabled on `StopAsync` and disposed when the service container is disposed on `Dispose`:

```
internal class TimedHostedService : IHostedService, IDisposable
{
    private readonly ILogger _logger;
    private Timer _timer;

    public TimedHostedService(ILogger<TimedHostedService> logger)
    {
        _logger = logger;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Timed Background Service is starting.");

        _timer = new Timer(DoWork, null, TimeSpan.Zero,
            TimeSpan.FromSeconds(5));

        return Task.CompletedTask;
    }

    private void DoWork(object state)
    {
        _logger.LogInformation("Timed Background Service is working.");
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Timed Background Service is stopping.");

        _timer?.Change(Timeout.Infinite, 0);

        return Task.CompletedTask;
    }

    public void Dispose()
    {
        _timer?.Dispose();
    }
}
```

The Timer doesn't wait for previous executions of `DoWork` to finish, so the approach shown might not be suitable for every scenario.

The service is registered in `Startup.ConfigureServices` with the `AddHostedService` extension method:

```
services.AddHostedService<TimedHostedService>();
```

Consuming a scoped service in a background task

To use scoped services within an `IHostedService`, create a scope. No scope is created for a hosted service by default.

The scoped background task service contains the background task's logic. In the following example, an ILogger is injected into the service:

```
internal interface IScopedProcessingService
{
    void DoWork();
}

internal class ScopedProcessingService : IScopedProcessingService
{
    private readonly ILogger _logger;

    public ScopedProcessingService(ILogger<ScopedProcessingService> logger)
    {
        _logger = logger;
    }

    public void DoWork()
    {
        _logger.LogInformation("Scoped Processing Service is working.");
    }
}
```

The hosted service creates a scope to resolve the scoped background task service to call its `DoWork` method:

```
internal class ConsumeScopedServiceHostedService : IHostedService
{
    private readonly ILogger _logger;

    public ConsumeScopedServiceHostedService(IServiceProvider services,
        ILogger<ConsumeScopedServiceHostedService> logger)
    {
        Services = services;
        _logger = logger;
    }

    public IServiceProvider Services { get; }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is starting.");

        DoWork();

        return Task.CompletedTask;
    }

    private void DoWork()
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is working.");

        using (var scope = Services.CreateScope())
        {
            var scopedProcessingService =
                scope.ServiceProvider
                    .GetRequiredService<IScopedProcessingService>();

            scopedProcessingService.DoWork();
        }
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is stopping.");

        return Task.CompletedTask;
    }
}
```

The services are registered in `Startup.ConfigureServices`. The `IHostedService` implementation is registered with the `AddHostedService` extension method:

```
services.AddHostedService<ConsumeScopedServiceHostedService>();
services.AddScoped<IScopedProcessingService, ScopedProcessingService>();
```

Queued background tasks

A background task queue is based on the .NET Framework 4.x QueueBackgroundWorkItem (tentatively scheduled to be built-in for ASP.NET Core):

```
public interface IBackgroundTaskQueue
{
    void QueueBackgroundWorkItem(Func<CancellationToken, Task> workItem);

    Task<Func<CancellationToken, Task>> DequeueAsync(
        CancellationToken cancellationToken);
}

public class BackgroundTaskQueue : IBackgroundTaskQueue
{
    private ConcurrentQueue<Func<CancellationToken, Task>> _workItems =
        new ConcurrentQueue<Func<CancellationToken, Task>>();
    private SemaphoreSlim _signal = new SemaphoreSlim(0);

    public void QueueBackgroundWorkItem(
        Func<CancellationToken, Task> workItem)
    {
        if (workItem == null)
        {
            throw new ArgumentNullException(nameof(workItem));
        }

        _workItems.Enqueue(workItem);
        _signal.Release();
    }

    public async Task<Func<CancellationToken, Task>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        await _signal.WaitAsync(cancellationToken);
        _workItems.TryDequeue(out var workItem);

        return workItem;
    }
}
```

In `QueueHostedService`, background tasks in the queue are dequeued and executed as a BackgroundService, which is a base class for implementing a long running `IHostedService`:

```
public class QueuedHostedService : BackgroundService
{
    private readonly ILogger _logger;

    public QueuedHostedService(IBackgroundTaskQueue taskQueue,
        ILoggerFactory loggerFactory)
    {
        TaskQueue = taskQueue;
        _logger = loggerFactory.CreateLogger<QueuedHostedService>();
    }

    public IBackgroundTaskQueue TaskQueue { get; }

    protected async override Task ExecuteAsync(
        CancellationToken cancellationToken)
    {
        _logger.LogInformation("Queued Hosted Service is starting.");

        while (!cancellationToken.IsCancellationRequested)
        {
            var workItem = await TaskQueue.DequeueAsync(cancellationToken);

            try
            {
                await workItem(cancellationToken);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex,
                   "Error occurred executing {WorkItem}.", nameof(workItem));
            }
        }

        _logger.LogInformation("Queued Hosted Service is stopping.");
    }
}
```

The services are registered in `Startup.ConfigureServices`. The `IHostedService` implementation is registered with the `AddHostedService` extension method:

```
services.AddHostedService<QueuedHostedService>();
services.AddSingleton<IBackgroundTaskQueue, BackgroundTaskQueue>();
```

In the Index page model class:

- The `IBackgroundTaskQueue` is injected into the constructor and assigned to `Queue`.
- An IServiceScopeFactory is injected and assigned to `_serviceScopeFactory`. The factory is used to create instances of IServiceScope, which is used to create services within a scope. A scope is created in order to use the app's `AppDbContext` (a scoped service) to write database records in the `IBackgroundTaskQueue` (a singleton service).

```
public class IndexModel : PageModel
{
    private readonly AppDbContext _db;
    private readonly ILogger _logger;
    private readonly IServiceScopeFactory _serviceScopeFactory;

    public IndexModel(AppDbContext db, IBackgroundTaskQueue queue,
        ILogger<IndexModel> logger, IServiceScopeFactory serviceScopeFactory)
    {
        _db = db;
        _logger = logger;
        Queue = queue;
        _serviceScopeFactory = serviceScopeFactory;
    }

    public IBackgroundTaskQueue Queue { get; }
```

When the **Add Task** button is selected on the Index page, the `OnPostAddTask` method is executed. `QueueBackgroundWorkItem` is called to enqueue a work item:

```
public IActionResult OnPostAddTaskAsync()
{
    Queue.QueueBackgroundWorkItem(async token =>
    {
        var guid = Guid.NewGuid().ToString();

        using (var scope = _serviceScopeFactory.CreateScope())
        {
            var scopedServices = scope.ServiceProvider;
            var db = scopedServices.GetRequiredService<AppDbContext>();

            for (int delayLoop = 1; delayLoop < 4; delayLoop++)
            {
                try
                {
                    db.Messages.Add(
                        new Message()
                        {
                            Text = $"Queued Background Task {guid} has " +
                                $"written a step. {delayLoop}/3"
                        });
                    await db.SaveChangesAsync();
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex,
                        "An error occurred writing to the " +
                        "database. Error: {Message}", ex.Message);
                }

                await Task.Delay(TimeSpan.FromSeconds(5), token);
            }
        }

        _logger.LogInformation(
            "Queued Background Task {Guid} is complete. 3/3", guid);
    });

    return RedirectToPage();
}
```

Additional resources

- Implement background tasks in microservices with IHostedService and the BackgroundService class
- Run background tasks with WebJobs in Azure App Service
- Timer

Use hosting startup assemblies in ASP.NET Core

9/22/2020 • 29 minutes to read • Edit Online

By Pavel Krymets

An IHostingStartup (hosting startup) implementation adds enhancements to an app at startup from an external assembly. For example, an external library can use a hosting startup implementation to provide additional configuration providers or services to an app.

View or download sample code (how to download)

HostingStartup attribute

A HostingStartup attribute indicates the presence of a hosting startup assembly to activate at runtime.

The entry assembly or the assembly containing the `Startup` class is automatically scanned for the `HostingStartup` attribute. The list of assemblies to search for `HostingStartup` attributes is loaded at runtime from configuration in the WebHostDefaults.HostingStartupAssembliesKey. The list of assemblies to exclude from discovery is loaded from the WebHostDefaults.HostingStartupExcludeAssembliesKey.

In the following example, the namespace of the hosting startup assembly is `StartupEnhancement`. The class containing the hosting startup code is `StartupEnhancementHostingStartup`:

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

The `HostingStartup` attribute is typically located in the hosting startup assembly's `IHostingStartup` implementation class file.

Discover loaded hosting startup assemblies

To discover loaded hosting startup assemblies, enable logging and check the app's logs. Errors that occur when loading assemblies are logged. Loaded hosting startup assemblies are logged at the Debug level, and all errors are logged.

Disable automatic loading of hosting startup assemblies

To disable automatic loading of hosting startup assemblies, use one of the following approaches:

- To prevent all hosting startup assemblies from loading, set one of the following to `true` or `1`:
  - Prevent Hosting Startup host configuration setting:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseSetting(
                    WebHostDefaults.PreventHostingStartupKey, "true")
                .UseStartup<Startup>();
        });
```

  - `ASPNETCORE_PREVENTHOSTINGSTARTUP` environment variable.

- To prevent specific hosting startup assemblies from loading, set one of the following to a semicolon-delimited string of hosting startup assemblies to exclude at startup:
  - Hosting Startup Exclude Assemblies host configuration setting:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseSetting(
                    WebHostDefaults.HostingStartupExcludeAssembliesKey,
                    "{ASSEMBLY1;ASSEMBLY2; ...}")
                .UseStartup<Startup>();
        });
```

  - `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES` environment variable.

If both the host configuration setting and the environment variable are set, the host setting controls the behavior.

Disabling hosting startup assemblies using the host setting or environment variable disables the assembly globally and may disable several characteristics of an app.

Project

Create a hosting startup with either of the following project types:

- Class library
- Console app without an entry point

Class library

A hosting startup enhancement can be provided in a class library. The library contains a `HostingStartup` attribute.

The sample code includes a Razor Pages app, *HostingStartupApp*, and a class library, *HostingStartupLibrary*. The class library:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration using the in-memory configuration provider (AddInMemoryCollection).
- Includes a `HostingStartup` attribute that identifies the hosting startup's namespace and class.

The `ServiceKeyInjection` class's Configure method uses an IWebHostBuilder to add enhancements to an app.

HostingStartupLibrary/ServiceKeyInjection.cs:

```
[assembly: HostingStartup(typeof(HostingStartupLibrary.ServiceKeyInjection))]

namespace HostingStartupLibrary
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromLibrary", "DEV_1111111-1111"},
                    {"ProdAccount_FromLibrary", "PROD_2222222-2222"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

The app's Index page reads and renders the configuration values for the two keys set by the class library's hosting startup assembly:

HostingStartupApp/Pages/Index.cshtml.cs:

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

The sample code also includes a NuGet package project that provides a separate hosting startup, *HostingStartupPackage*. The package has the same characteristics of the class library described earlier. The package:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration.
- Includes a `HostingStartup` attribute.

HostingStartupPackage/ServiceKeyInjection.cs:

```
[assembly: HostingStartup(typeof(HostingStartupPackage.ServiceKeyInjection))]

namespace HostingStartupPackage
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromPackage", "DEV_3333333-3333"},
                    {"ProdAccount_FromPackage", "PROD_4444444-4444"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

The app's Index page reads and renders the configuration values for the two keys set by the package's hosting startup assembly:

HostingStartupApp/Pages/Index.cshtml.cs:

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

Console app without an entry point

This approach is only available for .NET Core apps, not .NET Framework.

A dynamic hosting startup enhancement that doesn't require a compile-time reference for activation can be provided in a console app without an entry point that contains a `HostingStartup` attribute. Publishing the console app produces a hosting startup assembly that can be consumed from the runtime store.

A console app without an entry point is used in this process because:

- A dependencies file is required to consume the hosting startup in the hosting startup assembly. A dependencies file is a runnable app asset that's produced by publishing an app, not a library.
- A library can't be added directly to the runtime package store, which requires a runnable project that targets the shared runtime.

In the creation of a dynamic hosting startup:

- A hosting startup assembly is created from the console app without an entry point that:
  - Includes a class that contains the `IHostingStartup` implementation.
  - Includes a HostingStartup attribute to identify the `IHostingStartup` implementation class.
- The console app is published to obtain the hosting startup's dependencies. A consequence of publishing the console app is that unused dependencies are trimmed from the dependencies file.
- The dependencies file is modified to set the runtime location of the hosting startup assembly.
- The hosting startup assembly and its dependencies file is placed into the runtime package store. To discover the hosting startup assembly and its dependencies file, they're listed in a pair of environment variables.

The console app references the Microsoft.AspNetCore.Hosting.Abstractions package:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Hosting.Abstractions"
                      Version="3.0.0" />
  </ItemGroup>

</Project>
```

A HostingStartup attribute identifies a class as an implementation of `IHostingStartup` for loading and execution when building the IWebHost. In the following example, the namespace is `StartupEnhancement`, and the class is `StartupEnhancementHostingStartup`:

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

A class implements `IHostingStartup`. The class's Configure method uses an IWebHostBuilder to add enhancements to an app. `IHostingStartup.Configure` in the hosting startup assembly is called by the runtime before `Startup.Configure` in user code, which allows user code to overwrite any configuration provided by the hosting startup assembly.

```
namespace StartupEnhancement
{
    public class StartupEnhancementHostingStartup : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            // Use the IWebHostBuilder to add app enhancements.
        }
    }
}
```

When building an `IHostingStartup` project, the dependencies file (*.deps.json*) sets the `runtime` location of the assembly to the *bin* folder:

```
"targets": {
  ".NETCoreApp,Version=v3.0": {
    "StartupEnhancement/1.0.0": {
      "dependencies": {
        "Microsoft.AspNetCore.Hosting.Abstractions": "3.0.0"
      },
      "runtime": {
        "StartupEnhancement.dll": {}
      }
    }
  }
}
```

Only part of the file is shown. The assembly name in the example is `StartupEnhancement`.

Configuration provided by the hosting startup

There are two approaches to handling configuration depending on whether you want the hosting startup's configuration to take precedence or the app's configuration to take precedence:

1. Provide configuration to the app using ConfigureAppConfiguration to load the configuration after the app's ConfigureAppConfiguration delegates execute. Hosting startup configuration takes priority over the app's configuration using this approach.
2. Provide configuration to the app using UseConfiguration to load the configuration before the app's ConfigureAppConfiguration delegates execute. The app's configuration values take priority over those provided by the hosting startup using this approach.

```
public class ConfigurationInjection : IHostingStartup
{
    public void Configure(IWebHostBuilder builder)
    {
        Dictionary<string, string> dict;

        builder.ConfigureAppConfiguration(config =>
        {
            dict = new Dictionary<string, string>
            {
                {"ConfigurationKey1",
                    "From IHostingStartup: Higher priority " +
                    "than the app's configuration."},
            };

            config.AddInMemoryCollection(dict);
        });

        dict = new Dictionary<string, string>
        {
            {"ConfigurationKey2",
                "From IHostingStartup: Lower priority " +
                "than the app's configuration."},
        };

        var builtConfig = new ConfigurationBuilder()
            .AddInMemoryCollection(dict)
            .Build();

        builder.UseConfiguration(builtConfig);
    }
}
```

Specify the hosting startup assembly

For either a class library- or console app-supplied hosting startup, specify the hosting startup assembly's name in the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable. The environment variable is a semicolon-delimited list of assemblies.

Only hosting startup assemblies are scanned for the `HostingStartup` attribute. For the sample app, *HostingStartupApp*, to discover the hosting startups described earlier, the environment variable is set to the following value:

```
HostingStartupLibrary;HostingStartupPackage;StartupDiagnostics
```

A hosting startup assembly can also be set using the Hosting Startup Assemblies host configuration setting:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseSetting(
                    WebHostDefaults.HostingStartupAssembliesKey,
                    "{ASSEMBLY1;ASSEMBLY2; ...}")
                .UseStartup<Startup>();
        });
```

When multiple hosting startup assembles are present, their Configure methods are executed in the order that the assemblies are listed.

Activation

Options for hosting startup activation are:

- Runtime store: Activation doesn't require a compile-time reference for activation. The sample app places the hosting startup assembly and dependencies files into a folder, *deployment*, to facilitate deployment of the hosting startup in a multimachine environment. The *deployment* folder also includes a PowerShell script that creates or modifies environment variables on the deployment system to enable the hosting startup.
- Compile-time reference required for activation
  - NuGet package
  - Project bin folder

Runtime store

The hosting startup implementation is placed in the runtime store. A compile-time reference to the assembly isn't required by the enhanced app.

After the hosting startup is built, a runtime store is generated using the manifest project file and the dotnet store command.

```
dotnet store --manifest {MANIFEST FILE} --runtime {RUNTIME IDENTIFIER} --output {OUTPUT LOCATION} --skip-
optimization
```

In the sample app (*RuntimeStore* project) the following command is used:

```
dotnet store --manifest store.manifest.csproj --runtime win7-x64 --output ./deployment/store --skip-
optimization
```

For the runtime to discover the runtime store, the runtime store's location is added to the `DOTNET_SHARED_STORE` environment variable.

Modify and place the hosting startup's dependencies file

To activate the enhancement without a package reference to the enhancement, specify additional dependencies to the runtime with `additionalDeps`. `additionalDeps` allows you to:

- Extend the app's library graph by providing a set of additional *.deps.json* files to merge with the app's own *.deps.json* file on startup.
- Make the hosting startup assembly discoverable and loadable.

The recommended approach for generating the additional dependencies file is to:

1. Execute `dotnet publish` on the runtime store manifest file referenced in the previous section.
2. Remove the manifest reference from libraries and the `runtime` section of the resulting *.deps.json* file.

In the example project, the `store.manifest/1.0.0` property is removed from the `targets` and `libraries` section:

```
{
  "runtimeTarget": {
    "name": ".NETCoreApp,Version=v3.0",
    "signature": ""
  },
  "compilationOptions": {},
  "targets": {
    ".NETCoreApp,Version=v3.0": {
      "store.manifest/1.0.0": {
        "dependencies": {
          "StartupDiagnostics": "1.0.0"
        },
        "runtime": {
          "store.manifest.dll": {}
        }
      },
      "StartupDiagnostics/1.0.0": {
        "runtime": {
          "lib/netcoreapp3.0/StartupDiagnostics.dll": {
            "assemblyVersion": "1.0.0.0",
            "fileVersion": "1.0.0.0"
          }
        }
      }
    }
  },
  "libraries": {
    "store.manifest/1.0.0": {
      "type": "project",
      "serviceable": false,
      "sha512": ""
    },
    "StartupDiagnostics/1.0.0": {
      "type": "package",
      "serviceable": true,
      "sha512": "sha512-
xrhzuNSyM5/f4ZswhooJ9dmIYLP64wMnqUJSyTKVDKDVj5T+qtzypl8JmM/aFJLLpYrf0FYpVWvGujd7/FfMEw==",
      "path": "startupdiagnostics/1.0.0",
      "hashPath": "startupdiagnostics.1.0.0.nupkg.sha512"
    }
  }
}
```

Place the *.deps.json* file into the following location:

```
{ADDITIONAL DEPENDENCIES PATH}/shared/{SHARED FRAMEWORK NAME}/{SHARED FRAMEWORK VERSION}/{ENHANCEMENT
ASSEMBLY NAME}.deps.json
```

- `{ADDITIONAL DEPENDENCIES PATH}`: Location added to the `DOTNET_ADDITIONAL_DEPS` environment variable.
- `{SHARED FRAMEWORK NAME}`: Shared framework required for this additional dependencies file.
- `{SHARED FRAMEWORK VERSION}`: Minimum shared framework version.
- `{ENHANCEMENT ASSEMBLY NAME}`: The enhancement's assembly name.

In the sample app (*RuntimeStore* project), the additional dependencies file is placed into the following location:

```
deployment/additionalDeps/shared/Microsoft.AspNetCore.App/3.0.0/StartupDiagnostics.deps.json
```

For runtime to discover the runtime store location, the additional dependencies file location is added to the `DOTNET_ADDITIONAL_DEPS` environment variable.

In the sample app (*RuntimeStore* project), building the runtime store and generating the additional dependencies file is accomplished using a PowerShell script.

For examples of how to set environment variables for various operating systems, see Use multiple environments.

Deployment

To facilitate the deployment of a hosting startup in a multimachine environment, the sample app creates a *deployment* folder in published output that contains:

- The hosting startup runtime store.
- The hosting startup dependencies file.
- A PowerShell script that creates or modifies the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`, `DOTNET_SHARED_STORE`, and `DOTNET_ADDITIONAL_DEPS` to support the activation of the hosting startup. Run the script from an administrative PowerShell command prompt on the deployment system.

NuGet package

A hosting startup enhancement can be provided in a NuGet package. The package has a `HostingStartup` attribute. The hosting startup types provided by the package are made available to the app using either of the following approaches:

- The enhanced app's project file makes a package reference for the hosting startup in the app's project file (a compile-time reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (*.deps.json*). This approach applies to a hosting startup assembly package published to nuget.org.
- The hosting startup's dependencies file is made available to the enhanced app as described in the Runtime store section (without a compile-time reference).

For more information on NuGet packages and the runtime store, see the following topics:

- How to Create a NuGet Package with Cross Platform Tools
- Publishing packages
- Runtime package store

Project bin folder

A hosting startup enhancement can be provided by a *bin*-deployed assembly in the enhanced app. The hosting startup types provided by the assembly are made available to the app using one of the following approaches:

- The enhanced app's project file makes an assembly reference to the hosting startup (a compile-time

reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (*.deps.json*). This approach applies when the deployment scenario calls for making a compile-time reference to the hosting startup's assembly (*.dll* file) and moving the assembly to either:
    - The consuming project.
    - A location accessible by the consuming project.
- The hosting startup's dependencies file is made available to the enhanced app as described in the Runtime store section (without a compile-time reference).
- When targeting the .NET Framework, the assembly is loadable in the default load context, which on .NET Framework means that the assembly is located at either of the following locations:
    - Application base path: The *bin* folder where the app's executable (*.exe*) is located.
    - Global Assembly Cache (GAC): The GAC stores assemblies that several .NET Framework apps share. For more information, see How to: Install an assembly into the global assembly cache in the .NET Framework documentation.

Sample code

The sample code (how to download) demonstrates hosting startup implementation scenarios:

- Two hosting startup assemblies (class libraries) set a pair of in-memory configuration key-value pairs each:
    - NuGet package (*HostingStartupPackage*)
    - Class library (*HostingStartupLibrary*)
- A hosting startup is activated from a runtime store-deployed assembly (*StartupDiagnostics*). The assembly adds two middlewares to the app at startup that provide diagnostic information on:
    - Registered services
    - Address (scheme, host, path base, path, query string)
    - Connection (remote IP, remote port, local IP, local port, client certificate)
    - Request headers
    - Environment variables

To run the sample:

Activation from a NuGet package

1. Compile the *HostingStartupPackage* package with the dotnet pack command.
2. Add the package's assembly name of the *HostingStartupPackage* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
3. Compile and run the app. A package reference is present in the enhanced app (a compile-time reference). A `<PropertyGroup>` in the app's project file specifies the package project's output (*../HostingStartupPackage/bin/Debug*) as a package source. This allows the app to use the package without uploading the package to nuget.org. For more information, see the notes in the HostingStartupApp's project file.

    ```
    <PropertyGroup>

    <RestoreSources>$(RestoreSources);https://api.nuget.org/v3/index.json;../HostingStartupPackage/bin/De
    bug</RestoreSources>
    </PropertyGroup>
    ```

4. Observe that the service configuration key values rendered by the Index page match the values set by the package's `ServiceKeyInjection.Configure` method.

If you make changes to the *HostingStartupPackage* project and recompile it, clear the local NuGet package caches to ensure that the *HostingStartupApp* receives the updated package and not a stale package from the local cache. To clear the local NuGet caches, execute the following dotnet nuget locals command:

```
dotnet nuget locals all --clear
```

Activation from a class library

1. Compile the *HostingStartupLibrary* class library with the dotnet build command.
2. Add the class library's assembly name of *HostingStartupLibrary* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
3. *bin*-deploy the class library's assembly to the app by copying the *HostingStartupLibrary.dll* file from the class library's compiled output to the app's *bin/Debug* folder.
4. Compile and run the app. An `<ItemGroup>` in the app's project file references the class library's assembly (*.\bin\Debug\netcoreapp3.0\HostingStartupLibrary.dll*) (a compile-time reference). For more information, see the notes in the HostingStartupApp's project file.

   ```
   <ItemGroup>
     <Reference Include=".\\bin\\Debug\\netcoreapp3.0\\HostingStartupLibrary.dll">
       <HintPath>.\bin\Debug\netcoreapp3.0\HostingStartupLibrary.dll</HintPath>
       <SpecificVersion>False</SpecificVersion>
     </Reference>
   </ItemGroup>
   ```

5. Observe that the service configuration key values rendered by the Index page match the values set by the class library's `ServiceKeyInjection.Configure` method.

Activation from a runtime store-deployed assembly

1. The *StartupDiagnostics* project uses PowerShell to modify its *StartupDiagnostics.deps.json* file. PowerShell is installed by default on Windows starting with Windows 7 SP1 and Windows Server 2008 R2 SP1. To obtain PowerShell on other platforms, see Installing various versions of PowerShell.
2. Execute the *build.ps1* script in the *RuntimeStore* folder. The script:
   - Generates the `StartupDiagnostics` package in the *obj\packages* folder.
   - Generates the runtime store for `StartupDiagnostics` in the *store* folder. The `dotnet store` command in the script uses the `win7-x64` runtime identifier (RID) for a hosting startup deployed to Windows. When providing the hosting startup for a different runtime, substitute the correct RID on line 37 of the script. The runtime store for `StartupDiagnostics` would later be moved to the user's or system's runtime store on the machine where the assembly will be consumed. The user runtime store install location for the `StartupDiagnostics` assembly is *.dotnet/store/x64/netcoreapp3.0/startupdiagnostics/1.0.0/lib/netcoreapp3.0/StartupDiagnostics.dll*.
   - Generates the `additionalDeps` for `StartupDiagnostics` in the *additionalDeps* folder. The additional dependencies would later be moved to the user's or system's additional dependencies. The user `StartupDiagnostics` additional dependencies install location is *.dotnet/x64/additionalDeps/StartupDiagnostics/shared/Microsoft.NETCore.App/3.0.0/StartupDiagnostics.deps.json*.
   - Places the *deploy.ps1* file in the *deployment* folder.
3. Run the *deploy.ps1* script in the *deployment* folder. The script appends:
   - `StartupDiagnostics` to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
   - The hosting startup dependencies path (in the RuntimeStore project's *deployment* folder) to the `DOTNET_ADDITIONAL_DEPS` environment variable.

- The runtime store path (in the RuntimeStore project's *deployment* folder) to the `DOTNET_SHARED_STORE` environment variable.

4. Run the sample app.
5. Request the `/services` endpoint to see the app's registered services. Request the `/diag` endpoint to see the diagnostic information.

An IHostingStartup (hosting startup) implementation adds enhancements to an app at startup from an external assembly. For example, an external library can use a hosting startup implementation to provide additional configuration providers or services to an app.

View or download sample code (how to download)

HostingStartup attribute

A HostingStartup attribute indicates the presence of a hosting startup assembly to activate at runtime.

The entry assembly or the assembly containing the `Startup` class is automatically scanned for the `HostingStartup` attribute. The list of assemblies to search for `HostingStartup` attributes is loaded at runtime from configuration in the WebHostDefaults.HostingStartupAssembliesKey. The list of assemblies to exclude from discovery is loaded from the WebHostDefaults.HostingStartupExcludeAssembliesKey. For more information, see Web Host: Hosting Startup Assemblies and Web Host: Hosting Startup Exclude Assemblies.

In the following example, the namespace of the hosting startup assembly is `StartupEnhancement`. The class containing the hosting startup code is `StartupEnhancementHostingStartup`:

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

The `HostingStartup` attribute is typically located in the hosting startup assembly's `IHostingStartup` implementation class file.

Discover loaded hosting startup assemblies

To discover loaded hosting startup assemblies, enable logging and check the app's logs. Errors that occur when loading assemblies are logged. Loaded hosting startup assemblies are logged at the Debug level, and all errors are logged.

Disable automatic loading of hosting startup assemblies

To disable automatic loading of hosting startup assemblies, use one of the following approaches:

- To prevent all hosting startup assemblies from loading, set one of the following to `true` or `1`:
  - Prevent Hosting Startup host configuration setting.
  - `ASPNETCORE_PREVENTHOSTINGSTARTUP` environment variable.
- To prevent specific hosting startup assemblies from loading, set one of the following to a semicolon-delimited string of hosting startup assemblies to exclude at startup:
  - Hosting Startup Exclude Assemblies host configuration setting.
  - `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES` environment variable.

If both the host configuration setting and the environment variable are set, the host setting controls the behavior.

Disabling hosting startup assemblies using the host setting or environment variable disables the assembly globally and may disable several characteristics of an app.

Project

Create a hosting startup with either of the following project types:

- Class library
- Console app without an entry point

Class library

A hosting startup enhancement can be provided in a class library. The library contains a `HostingStartup` attribute.

The sample code includes a Razor Pages app, *HostingStartupApp*, and a class library, *HostingStartupLibrary*. The class library:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration using the in-memory configuration provider (AddInMemoryCollection).
- Includes a `HostingStartup` attribute that identifies the hosting startup's namespace and class.

The `ServiceKeyInjection` class's Configure method uses an IWebHostBuilder to add enhancements to an app.

HostingStartupLibrary/ServiceKeyInjection.cs:

```
[assembly: HostingStartup(typeof(HostingStartupLibrary.ServiceKeyInjection))]

namespace HostingStartupLibrary
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromLibrary", "DEV_1111111-1111"},
                    {"ProdAccount_FromLibrary", "PROD_2222222-2222"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

The app's Index page reads and renders the configuration values for the two keys set by the class library's hosting startup assembly:

HostingStartupApp/Pages/Index.cshtml.cs:

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

The sample code also includes a NuGet package project that provides a separate hosting startup, *HostingStartupPackage*. The package has the same characteristics of the class library described earlier. The package:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration.
- Includes a `HostingStartup` attribute.

HostingStartupPackage/ServiceKeyInjection.cs:

```
[assembly: HostingStartup(typeof(HostingStartupPackage.ServiceKeyInjection))]

namespace HostingStartupPackage
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromPackage", "DEV_3333333-3333"},
                    {"ProdAccount_FromPackage", "PROD_4444444-4444"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

The app's Index page reads and renders the configuration values for the two keys set by the package's hosting startup assembly:

HostingStartupApp/Pages/Index.cshtml.cs:

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

Console app without an entry point

This approach is only available for .NET Core apps, not .NET Framework.

A dynamic hosting startup enhancement that doesn't require a compile-time reference for activation can be provided in a console app without an entry point that contains a `HostingStartup` attribute. Publishing the console app produces a hosting startup assembly that can be consumed from the runtime store.

A console app without an entry point is used in this process because:

- A dependencies file is required to consume the hosting startup in the hosting startup assembly. A dependencies file is a runnable app asset that's produced by publishing an app, not a library.
- A library can't be added directly to the runtime package store, which requires a runnable project that targets the shared runtime.

In the creation of a dynamic hosting startup:

- A hosting startup assembly is created from the console app without an entry point that:
  - Includes a class that contains the `IHostingStartup` implementation.
  - Includes a HostingStartup attribute to identify the `IHostingStartup` implementation class.
- The console app is published to obtain the hosting startup's dependencies. A consequence of publishing the console app is that unused dependencies are trimmed from the dependencies file.
- The dependencies file is modified to set the runtime location of the hosting startup assembly.
- The hosting startup assembly and its dependencies file is placed into the runtime package store. To discover the hosting startup assembly and its dependencies file, they're listed in a pair of environment variables.

The console app references the Microsoft.AspNetCore.Hosting.Abstractions package:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Hosting.Abstractions"
                      Version="2.1.1" />
  </ItemGroup>

</Project>
```

A HostingStartup attribute identifies a class as an implementation of `IHostingStartup` for loading and execution when building the IWebHost. In the following example, the namespace is `StartupEnhancement`, and the class is `StartupEnhancementHostingStartup`:

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

A class implements `IHostingStartup`. The class's Configure method uses an IWebHostBuilder to add enhancements to an app. `IHostingStartup.Configure` in the hosting startup assembly is called by the runtime before `Startup.Configure` in user code, which allows user code to overwrite any configuration provided by the hosting startup assembly.

```
namespace StartupEnhancement
{
    public class StartupEnhancementHostingStartup : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            // Use the IWebHostBuilder to add app enhancements.
        }
    }
}
```

When building an `IHostingStartup` project, the dependencies file (*.deps.json*) sets the `runtime` location of the assembly to the *bin* folder:

```
"targets": {
  ".NETCoreApp,Version=v2.1": {
    "StartupEnhancement/1.0.0": {
      "dependencies": {
        "Microsoft.AspNetCore.Hosting.Abstractions": "2.1.1"
      },
      "runtime": {
        "StartupEnhancement.dll": {}
      }
    }
  }
}
```

Only part of the file is shown. The assembly name in the example is `StartupEnhancement`.

Configuration provided by the hosting startup

There are two approaches to handling configuration depending on whether you want the hosting startup's configuration to take precedence or the app's configuration to take precedence:

1. Provide configuration to the app using ConfigureAppConfiguration to load the configuration after the app's ConfigureAppConfiguration delegates execute. Hosting startup configuration takes priority over the app's configuration using this approach.
2. Provide configuration to the app using UseConfiguration to load the configuration before the app's ConfigureAppConfiguration delegates execute. The app's configuration values take priority over those provided by the hosting startup using this approach.

```
public class ConfigurationInjection : IHostingStartup
{
    public void Configure(IWebHostBuilder builder)
    {
        Dictionary<string, string> dict;

        builder.ConfigureAppConfiguration(config =>
        {
            dict = new Dictionary<string, string>
            {
                {"ConfigurationKey1",
                    "From IHostingStartup: Higher priority " +
                    "than the app's configuration."},
            };

            config.AddInMemoryCollection(dict);
        });

        dict = new Dictionary<string, string>
        {
            {"ConfigurationKey2",
                "From IHostingStartup: Lower priority " +
                "than the app's configuration."},
        };

        var builtConfig = new ConfigurationBuilder()
            .AddInMemoryCollection(dict)
            .Build();

        builder.UseConfiguration(builtConfig);
    }
}
```

Specify the hosting startup assembly

For either a class library- or console app-supplied hosting startup, specify the hosting startup assembly's name in the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable. The environment variable is a semicolon-delimited list of assemblies.

Only hosting startup assemblies are scanned for the `HostingStartup` attribute. For the sample app, *HostingStartupApp*, to discover the hosting startups described earlier, the environment variable is set to the following value:

```
HostingStartupLibrary;HostingStartupPackage;StartupDiagnostics
```

A hosting startup assembly can also be set using the Hosting Startup Assemblies host configuration setting.

When multiple hosting startup assembles are present, their Configure methods are executed in the order that the assemblies are listed.

Activation

Options for hosting startup activation are:

- Runtime store: Activation doesn't require a compile-time reference for activation. The sample app places the hosting startup assembly and dependencies files into a folder, *deployment*, to facilitate deployment of the hosting startup in a multimachine environment. The *deployment* folder also includes a PowerShell script that creates or modifies environment variables on the deployment system to enable the hosting startup.
- Compile-time reference required for activation

- NuGet package
- Project bin folder

Runtime store

The hosting startup implementation is placed in the runtime store. A compile-time reference to the assembly isn't required by the enhanced app.

After the hosting startup is built, a runtime store is generated using the manifest project file and the dotnet store command.

```
dotnet store --manifest {MANIFEST FILE} --runtime {RUNTIME IDENTIFIER} --output {OUTPUT LOCATION} --skip-optimization
```

In the sample app (*RuntimeStore* project) the following command is used:

```
dotnet store --manifest store.manifest.csproj --runtime win7-x64 --output ./deployment/store --skip-optimization
```

For the runtime to discover the runtime store, the runtime store's location is added to the `DOTNET_SHARED_STORE` environment variable.

Modify and place the hosting startup's dependencies file

To activate the enhancement without a package reference to the enhancement, specify additional dependencies to the runtime with `additionalDeps`. `additionalDeps` allows you to:

- Extend the app's library graph by providing a set of additional *.deps.json* files to merge with the app's own *.deps.json* file on startup.
- Make the hosting startup assembly discoverable and loadable.

The recommended approach for generating the additional dependencies file is to:

1. Execute `dotnet publish` on the runtime store manifest file referenced in the previous section.
2. Remove the manifest reference from libraries and the `runtime` section of the resulting *.deps.json* file.

In the example project, the `store.manifest/1.0.0` property is removed from the `targets` and `libraries` section:

```
{
  "runtimeTarget": {
    "name": ".NETCoreApp,Version=v2.1",
    "signature": "4ea77c7b75ad1895ae1ea65e6ba2399010514f99"
  },
  "compilationOptions": {},
  "targets": {
    ".NETCoreApp,Version=v2.1": {
      "store.manifest/1.0.0": {
        "dependencies": {
          "StartupDiagnostics": "1.0.0"
        },
        "runtime": {
          "store.manifest.dll": {}
        }
      },
      "StartupDiagnostics/1.0.0": {
        "runtime": {
          "lib/netcoreapp2.1/StartupDiagnostics.dll": {
            "assemblyVersion": "1.0.0.0",
            "fileVersion": "1.0.0.0"
          }
        }
      }
    }
  },
  "libraries": {
    "store.manifest/1.0.0": {
      "type": "project",
      "serviceable": false,
      "sha512": ""
    },
    "StartupDiagnostics/1.0.0": {
      "type": "package",
      "serviceable": true,
      "sha512": "sha512-
oiQr60vBQW7+nBTmgKLSldj06WNLRTdhOZpAdEbCuapoZ+M2DJH2uQbRLvFT8EGAAv4TAKzNtcztpx5YOgBXQQ==",
      "path": "startupdiagnostics/1.0.0",
      "hashPath": "startupdiagnostics.1.0.0.nupkg.sha512"
    }
  }
}
```

Place the *.deps.json* file into the following location:

```
{ADDITIONAL DEPENDENCIES PATH}/shared/{SHARED FRAMEWORK NAME}/{SHARED FRAMEWORK VERSION}/{ENHANCEMENT
ASSEMBLY NAME}.deps.json
```

- `{ADDITIONAL DEPENDENCIES PATH}`: Location added to the `DOTNET_ADDITIONAL_DEPS` environment variable.
- `{SHARED FRAMEWORK NAME}`: Shared framework required for this additional dependencies file.
- `{SHARED FRAMEWORK VERSION}`: Minimum shared framework version.
- `{ENHANCEMENT ASSEMBLY NAME}`: The enhancement's assembly name.

In the sample app (*RuntimeStore* project), the additional dependencies file is placed into the following location:

```
deployment/additionalDeps/shared/Microsoft.AspNetCore.App/2.1.0/StartupDiagnostics.deps.json
```

For runtime to discover the runtime store location, the additional dependencies file location is added to the `DOTNET_ADDITIONAL_DEPS` environment variable.

In the sample app (*RuntimeStore* project), building the runtime store and generating the additional

dependencies file is accomplished using a PowerShell script.

For examples of how to set environment variables for various operating systems, see Use multiple environments.

Deployment

To facilitate the deployment of a hosting startup in a multimachine environment, the sample app creates a *deployment* folder in published output that contains:

- The hosting startup runtime store.
- The hosting startup dependencies file.
- A PowerShell script that creates or modifies the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`, `DOTNET_SHARED_STORE`, and `DOTNET_ADDITIONAL_DEPS` to support the activation of the hosting startup. Run the script from an administrative PowerShell command prompt on the deployment system.

NuGet package

A hosting startup enhancement can be provided in a NuGet package. The package has a `HostingStartup` attribute. The hosting startup types provided by the package are made available to the app using either of the following approaches:

- The enhanced app's project file makes a package reference for the hosting startup in the app's project file (a compile-time reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (*.deps.json*). This approach applies to a hosting startup assembly package published to nuget.org.
- The hosting startup's dependencies file is made available to the enhanced app as described in the Runtime store section (without a compile-time reference).

For more information on NuGet packages and the runtime store, see the following topics:

- How to Create a NuGet Package with Cross Platform Tools
- Publishing packages
- Runtime package store

Project bin folder

A hosting startup enhancement can be provided by a *bin*-deployed assembly in the enhanced app. The hosting startup types provided by the assembly are made available to the app using one of the following approaches:

- The enhanced app's project file makes an assembly reference to the hosting startup (a compile-time reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (*.deps.json*). This approach applies when the deployment scenario calls for making a compile-time reference to the hosting startup's assembly (*.dll* file) and moving the assembly to either:
  - The consuming project.
  - A location accessible by the consuming project.
- The hosting startup's dependencies file is made available to the enhanced app as described in the Runtime store section (without a compile-time reference).
- When targeting the .NET Framework, the assembly is loadable in the default load context, which on .NET Framework means that the assembly is located at either of the following locations:
  - Application base path: The *bin* folder where the app's executable (*.exe*) is located.
  - Global Assembly Cache (GAC): The GAC stores assemblies that several .NET Framework apps share. For more information, see How to: Install an assembly into the global assembly cache in the .NET Framework documentation.

Sample code

The sample code (how to download) demonstrates hosting startup implementation scenarios:

- Two hosting startup assemblies (class libraries) set a pair of in-memory configuration key-value pairs each:
  - NuGet package (*HostingStartupPackage*)
  - Class library (*HostingStartupLibrary*)
- A hosting startup is activated from a runtime store-deployed assembly (*StartupDiagnostics*). The assembly adds two middlewares to the app at startup that provide diagnostic information on:
  - Registered services
  - Address (scheme, host, path base, path, query string)
  - Connection (remote IP, remote port, local IP, local port, client certificate)
  - Request headers
  - Environment variables

To run the sample:

Activation from a NuGet package

1. Compile the *HostingStartupPackage* package with the dotnet pack command.
2. Add the package's assembly name of the *HostingStartupPackage* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
3. Compile and run the app. A package reference is present in the enhanced app (a compile-time reference). A `<PropertyGroup>` in the app's project file specifies the package project's output (*../HostingStartupPackage/bin/Debug*) as a package source. This allows the app to use the package without uploading the package to nuget.org. For more information, see the notes in the HostingStartupApp's project file.

   ```
   <PropertyGroup>

   <RestoreSources>$(RestoreSources);https://api.nuget.org/v3/index.json;../HostingStartupPackage/bin/De
   bug</RestoreSources>
   </PropertyGroup>
   ```

4. Observe that the service configuration key values rendered by the Index page match the values set by the package's `ServiceKeyInjection.Configure` method.

If you make changes to the *HostingStartupPackage* project and recompile it, clear the local NuGet package caches to ensure that the *HostingStartupApp* receives the updated package and not a stale package from the local cache. To clear the local NuGet caches, execute the following dotnet nuget locals command:

```
dotnet nuget locals all --clear
```

Activation from a class library

1. Compile the *HostingStartupLibrary* class library with the dotnet build command.
2. Add the class library's assembly name of *HostingStartupLibrary* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
3. *bin*-deploy the class library's assembly to the app by copying the *HostingStartupLibrary.dll* file from the class library's compiled output to the app's *bin/Debug* folder.
4. Compile and run the app. An `<ItemGroup>` in the app's project file references the class library's assembly

(*.\bin\Debug\netcoreapp2.1\HostingStartupLibrary.dll*) (a compile-time reference). For more information, see the notes in the HostingStartupApp's project file.

```
<ItemGroup>
  <Reference Include=".\\bin\\Debug\\netcoreapp2.1\\HostingStartupLibrary.dll">
    <HintPath>.\bin\Debug\netcoreapp2.1\HostingStartupLibrary.dll</HintPath>
    <SpecificVersion>False</SpecificVersion>
  </Reference>
</ItemGroup>
```

5. Observe that the service configuration key values rendered by the Index page match the values set by the class library's `ServiceKeyInjection.Configure` method.

Activation from a runtime store-deployed assembly

1. The *StartupDiagnostics* project uses PowerShell to modify its *StartupDiagnostics.deps.json* file. PowerShell is installed by default on Windows starting with Windows 7 SP1 and Windows Server 2008 R2 SP1. To obtain PowerShell on other platforms, see Installing various versions of PowerShell.
2. Execute the *build.ps1* script in the *RuntimeStore* folder. The script:
   - Generates the `StartupDiagnostics` package in the *obj\packages* folder.
   - Generates the runtime store for `StartupDiagnostics` in the *store* folder. The `dotnet store` command in the script uses the `win7-x64` runtime identifier (RID) for a hosting startup deployed to Windows. When providing the hosting startup for a different runtime, substitute the correct RID on line 37 of the script. The runtime store for `StartupDiagnostics` would later be moved to the user's or system's runtime store on the machine where the assembly will be consumed. The user runtime store install location for the `StartupDiagnostics` assembly is *.dotnet/store/x64/netcoreapp2.2/startupdiagnostics/1.0.0/lib/netcoreapp2.2/StartupDiagnostics.dll*.
   - Generates the `additionalDeps` for `StartupDiagnostics` in the *additionalDeps* folder. The additional dependencies would later be moved to the user's or system's additional dependencies. The user `StartupDiagnostics` additional dependencies install location is *.dotnet/x64/additionalDeps/StartupDiagnostics/shared/Microsoft.NETCore.App/2.2.0/StartupDiagnostics.deps.json*.
   - Places the *deploy.ps1* file in the *deployment* folder.
3. Run the *deploy.ps1* script in the *deployment* folder. The script appends:
   - `StartupDiagnostics` to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
   - The hosting startup dependencies path (in the RuntimeStore project's *deployment* folder) to the `DOTNET_ADDITIONAL_DEPS` environment variable.
   - The runtime store path (in the RuntimeStore project's *deployment* folder) to the `DOTNET_SHARED_STORE` environment variable.
4. Run the sample app.
5. Request the `/services` endpoint to see the app's registered services. Request the `/diag` endpoint to see the diagnostic information.

Use ASP.NET Core APIs in a class library

9/22/2020 • 9 minutes to read • Edit Online

By Scott Addie

This document provides guidance for using ASP.NET Core APIs in a class library. For all other library guidance, see Open-source library guidance.

Determine which ASP.NET Core versions to support

ASP.NET Core adheres to the .NET Core support policy. Consult the support policy when determining which ASP.NET Core versions to support in a library. A library should:

- Make an effort to support all ASP.NET Core versions classified as *Long-Term Support* (LTS).
- Not feel obligated to support ASP.NET Core versions classified as *End of Life* (EOL).

As preview releases of ASP.NET Core are made available, breaking changes are posted in the aspnet/Announcements GitHub repository. Compatibility testing of libraries can be conducted as framework features are being developed.

Use the ASP.NET Core shared framework

With the release of .NET Core 3.0, many ASP.NET Core assemblies are no longer published to NuGet as packages. Instead, the assemblies are included in the `Microsoft.AspNetCore.App` shared framework, which is installed with the .NET Core SDK and runtime installers. For a list of packages no longer being published, see Remove obsolete package references.

As of .NET Core 3.0, projects using the `Microsoft.NET.Sdk.Web` MSBuild SDK implicitly reference the shared framework. Projects using the `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Razor` SDK must reference ASP.NET Core to use ASP.NET Core APIs in the shared framework.

To reference ASP.NET Core, add the following `<FrameworkReference>` element to your project file:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

Referencing ASP.NET Core in this manner is only supported for projects targeting .NET Core 3.x.

Include Blazor extensibility

Blazor supports WebAssembly (WASM) and Server hosting models. Unless there's a specific reason not to, a Razor components library should support both hosting models. A Razor components library must use the Microsoft.NET.Sdk.Razor SDK.

Support both hosting models

To support Razor component consumption from both Blazor Server and Blazor WASM projects, use the following instructions for your editor.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Use the **Razor Class Library** project template. The template's **Support pages and views** checkbox should be deselected.

The project generated from the template does the following things:

- Targets .NET Standard 2.0.
- Sets the `RazorLangVersion` property to `3.0`. `3.0` is the default value for .NET Core 3.x.
- Adds the following package references:
  - Microsoft.AspNetCore.Components
  - Microsoft.AspNetCore.Components.Web

For example:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFrameworks>netstandard2.0</TargetFrameworks>
    <RazorLangVersion>3.0</RazorLangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Components" Version="3.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Components.Web" Version="3.0.0" />
  </ItemGroup>

</Project>
```

Support a specific hosting model

It's far less common to support a single Blazor hosting model. As an example, to support Razor component consumption from Blazor Server projects only:

- Target .NET Core 3.x.
- Add a `<FrameworkReference>` element for the shared framework.

For example:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

For more information on libraries containing Razor components, see ASP.NET Core Razor components class libraries.

Include MVC extensibility

This section outlines recommendations for libraries that include:

- Razor views or Razor Pages
- Tag Helpers
- View components

This section doesn't discuss multi-targeting to support multiple versions of MVC. For guidance on supporting multiple ASP.NET Core versions, see Support multiple ASP.NET Core versions.

Razor views or Razor Pages

A project that includes Razor views or Razor Pages must use the Microsoft.NET.Sdk.Razor SDK.

If the project targets .NET Core 3.x, it requires:

- An `AddRazorSupportForMvc` MSBuild property set to `true`.
- A `<FrameworkReference>` element for the shared framework.

The **Razor Class Library** project template satisfies the preceding requirements for projects targeting .NET Core 3.x. Use the following instructions for your editor.

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

Use the **Razor Class Library** project template. The template's **Support pages and views** checkbox should be selected.

For example:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

If the project targets .NET Standard instead, a Microsoft.AspNetCore.Mvc package reference is required. The `Microsoft.AspNetCore.Mvc` package moved into the shared framework in ASP.NET Core 3.0 and is therefore no longer published. For example:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  </ItemGroup>

</Project>
```

Tag Helpers

A project that includes Tag Helpers should use the `Microsoft.NET.Sdk` SDK. If targeting .NET Core 3.x, add a `<FrameworkReference>` element for the shared framework. For example:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

If targeting .NET Standard (to support versions earlier than ASP.NET Core 3.x), add a package reference to Microsoft.AspNetCore.Mvc.Razor. The `Microsoft.AspNetCore.Mvc.Razor` package moved into the shared framework and is therefore no longer published. For example:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  </ItemGroup>

</Project>
```

View components

A project that includes View components should use the `Microsoft.NET.Sdk` SDK. If targeting .NET Core 3.x, add a `<FrameworkReference>` element for the shared framework. For example:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

If targeting .NET Standard (to support versions earlier than ASP.NET Core 3.x), add a package reference to Microsoft.AspNetCore.Mvc.ViewFeatures. The `Microsoft.AspNetCore.Mvc.ViewFeatures` package moved into the shared framework and is therefore no longer published. For example:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.ViewFeatures" Version="2.2.0" />
  </ItemGroup>

</Project>
```

Support multiple ASP.NET Core versions

Multi-targeting is required to author a library that supports multiple variants of ASP.NET Core. Consider a scenario in which a Tag Helpers library must support the following ASP.NET Core variants:

- ASP.NET Core 2.1 targeting .NET Framework 4.6.1
- ASP.NET Core 2.x targeting .NET Core 2.x
- ASP.NET Core 3.x targeting .NET Core 3.x

The following project file supports these variants via the `TargetFrameworks` property:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netcoreapp2.1;netcoreapp3.0;net461</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.0'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor" Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.0'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```

With the preceding project file:

- The `Markdig` package is added for all consumers.
- A reference to Microsoft.AspNetCore.Mvc.Razor is added for consumers targeting .NET Framework 4.6.1 or later or .NET Core 2.x. Version 2.1.0 of the package works with ASP.NET Core 2.2 because of backwards compatibility.
- The shared framework is referenced for consumers targeting .NET Core 3.x. The `Microsoft.AspNetCore.Mvc.Razor` package is included in the shared framework.

Alternatively, .NET Standard 2.0 could be targeted instead of targeting both .NET Core 2.1 and .NET Framework 4.6.1:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netstandard2.0;netcoreapp3.0</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.0'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor" Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.0'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```

With the preceding project file, the following caveats exist:

- Since the library only contains Tag Helpers, it's more straightforward to target the specific platforms on which ASP.NET Core runs: .NET Core and .NET Framework. Tag Helpers can't be used by other .NET Standard 2.0-compliant target frameworks such as Unity, UWP, and Xamarin.
- Using .NET Standard 2.0 from .NET Framework has some issues that were addressed in .NET Framework 4.7.2. You can improve the experience for consumers using .NET Framework 4.6.1 through 4.7.1 by targeting .NET Framework 4.6.1.

If your library needs to call platform-specific APIs, target specific .NET implementations instead of .NET Standard. For more information, see Multi-targeting.

Use an API that hasn't changed

Imagine a scenario in which you're upgrading a middleware library from .NET Core 2.2 to 3.0. The ASP.NET Core middleware APIs being used in the library haven't changed between ASP.NET Core 2.2 and 3.0. To continue supporting the middleware library in .NET Core 3.0, take the following steps:

- Follow the standard library guidance.
- Add a package reference for each API's NuGet package if the corresponding assembly doesn't exist in the shared framework.

Use an API that changed

Imagine a scenario in which you're upgrading a library from .NET Core 2.2 to .NET Core 3.0. An ASP.NET Core API being used in the library has a breaking change in ASP.NET Core 3.0. Consider whether the library can be rewritten to not use the broken API in all versions.

If you can rewrite the library, do so and continue to target an earlier target framework (for example, .NET Standard 2.0 or .NET Framework 4.6.1) with package references.

If you can't rewrite the library, take the following steps:

- Add a target for .NET Core 3.0.
- Add a `<FrameworkReference>` element for the shared framework.
- Use the #if preprocessor directive with the appropriate target framework symbol to conditionally compile code.

For example, synchronous reads and writes on HTTP request and response streams are disabled by default as of ASP.NET Core 3.0. ASP.NET Core 2.2 supports the synchronous behavior by default. Consider a middleware library

in which synchronous reads and writes should be enabled where I/O is occurring. The library should enclose the code to enable synchronous features in the appropriate preprocessor directive. For example:

```
public async Task Invoke(HttpContext httpContext)
{
    if (httpContext.Request.Path.StartsWithSegments(_path, StringComparison.Ordinal))
    {
        httpContext.Response.StatusCode = 200;
        httpContext.Response.ContentType = "application/json";
        httpContext.Response.ContentLength = _bufferSize;

#if !NETCOREAPP3_0 && !NETCOREAPP5_0
        var syncIOFeature = httpContext.Features.Get<IHttpBodyControlFeature>();
        if (syncIOFeature != null)
        {
            syncIOFeature.AllowSynchronousIO = true;
        }

        using (var sw = new StreamWriter(
            httpContext.Response.Body, _encoding, bufferSize: _bufferSize))
        {
            _json.Serialize(sw, new JsonMessage { message = "Hello, World!" });
        }
#else
        await JsonSerializer.SerializeAsync<JsonMessage>(
            httpContext.Response.Body, new JsonMessage { message = "Hello, World!" });
#endif
        return;
    }

    await _next(httpContext);
}
```

Use an API introduced in 3.0

Imagine that you want to use an ASP.NET Core API that was introduced in ASP.NET Core 3.0. Consider the following questions:

1. Does the library functionally require the new API?
2. Can the library implement this feature in a different way?

If the library functionally requires the API and there's no way to implement it down-level:

- Target .NET Core 3.x only.
- Add a `<FrameworkReference>` element for the shared framework.

If the library can implement the feature in a different way:

- Add .NET Core 3.x as a target framework.
- Add a `<FrameworkReference>` element for the shared framework.
- Use the #if preprocessor directive with the appropriate target framework symbol to conditionally compile code.

For example, the following Tag Helper uses the IWebHostEnvironment interface introduced in ASP.NET Core 3.0. Consumers targeting .NET Core 3.0 execute the code path defined by the `NETCOREAPP3_0` target framework symbol. The Tag Helper's constructor parameter type changes to IHostingEnvironment for .NET Core 2.1 and .NET Framework 4.6.1 consumers. This change was necessary because ASP.NET Core 3.0 marked `IHostingEnvironment` as obsolete and recommended `IWebHostEnvironment` as the replacement.

```
[HtmlTargetElement("script", Attributes = "asp-inline")]
public class ScriptInliningTagHelper : TagHelper
{
    private readonly IFileProvider _wwwroot;

#if NETCOREAPP3_0
    public ScriptInliningTagHelper(IWebHostEnvironment env)
#else
    public ScriptInliningTagHelper(IHostingEnvironment env)
#endif
    {
        _wwwroot = env.WebRootFileProvider;
    }

    // code omitted for brevity
}
```

The following multi-targeted project file supports this Tag Helper scenario:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netcoreapp2.1;netcoreapp3.0;net461</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.0'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor" Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.0'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```

Use an API removed from the shared framework

To use an ASP.NET Core assembly that was removed from the shared framework, add the appropriate package reference. For a list of packages removed from the shared framework in ASP.NET Core 3.0, see Remove obsolete package references.

For example, to add the web API client:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNet.WebApi.Client" Version="5.2.7" />
  </ItemGroup>

</Project>
```

Additional resources

- Reusable Razor UI in class libraries with ASP.NET Core
- ASP.NET Core Razor components class libraries
- .NET implementation support
- .NET support policies

Microsoft.AspNetCore.App for ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

The ASP.NET Core shared framework (`Microsoft.AspNetCore.App`) contains assemblies that are developed and supported by Microsoft. `Microsoft.AspNetCore.App` is installed when the .NET Core 3.0 or later SDK is installed. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and includes a runtime component and a targeting pack. For more information, see The shared framework.

- Projects that target the `Microsoft.NET.Sdk.Web` SDK implicitly reference the `Microsoft.AspNetCore.App` framework.

No additional references are required for these projects:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>
    ...
</Project>
```

The ASP.NET Core shared framework:

- Doesn't include third-party dependencies.
- Includes all supported packages by the ASP.NET Core team.

This feature requires ASP.NET Core 2.x targeting .NET Core 2.x.

The Microsoft.AspNetCore.App metapackage for ASP.NET Core:

- Does not include third-party dependencies except for Json.NET, Remotion.Linq, and IX-Async. These 3rd-party dependencies are deemed necessary to ensure the major frameworks features function.
- Includes all supported packages by the ASP.NET Core team except those that contain third-party dependencies (other than those previously mentioned).
- Includes all supported packages by the Entity Framework Core team except those that contain third-party dependencies (other than those previously mentioned).

All the features of ASP.NET Core 2.x and Entity Framework Core 2.x are included in the `Microsoft.AspNetCore.App` package. The default project templates targeting ASP.NET Core 2.x use this package. We recommend applications targeting ASP.NET Core 2.x and Entity Framework Core 2.x use the `Microsoft.AspNetCore.App` package.

The version number of the `Microsoft.AspNetCore.App` metapackage represents the minimum ASP.NET Core version and Entity Framework Core version.

Using the `Microsoft.AspNetCore.App` metapackage provides version restrictions that protect your app:

- If a package is included that has a transitive (not direct) dependency on a package in `Microsoft.AspNetCore.App` , and those version numbers differ, NuGet will generate an error.
- Other packages added to your app cannot change the version of packages included in

`Microsoft.AspNetCore.App`.

- Version consistency ensures a reliable experience. `Microsoft.AspNetCore.App` was designed to prevent untested version combinations of related bits being used together in the same app.

Applications that use the `Microsoft.AspNetCore.App` metapackage automatically take advantage of the ASP.NET Core shared framework. When you use the `Microsoft.AspNetCore.App` metapackage, **no** assets from the referenced ASP.NET Core NuGet packages are deployed with the application—the ASP.NET Core shared framework contains these assets. The assets in the shared framework are precompiled to improve application startup time. For more information, see The shared framework.

The following project file references the `Microsoft.AspNetCore.App` metapackage for ASP.NET Core and represents a typical ASP.NET Core 2.2 template:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

The preceding markup represents a typical ASP.NET Core 2.x template. It doesn't specify a version number for the `Microsoft.AspNetCore.App` package reference. When the version is not specified, an implicit version is specified by the SDK, that is, `Microsoft.NET.Sdk.Web`. We recommend relying on the implicit version specified by the SDK and not explicitly setting the version number on the package reference. If you have questions on this approach, leave a GitHub comment at the Discussion for the Microsoft.AspNetCore.App implicit version.

The implicit version is set to `major.minor.0` for portable apps. The shared framework roll-forward mechanism will run the app on the latest compatible version among the installed shared frameworks. To guarantee the same version is used in development, test, and production, ensure the same version of the shared framework is installed in all environments. For self contained apps, the implicit version number is set to the `major.minor.patch` of the shared framework bundled in the installed SDK.

Specifying a version number on the `Microsoft.AspNetCore.App` reference does **not** guarantee that version of the shared framework will be chosen. For example, suppose version "2.2.1" is specified, but "2.2.3" is installed. In that case, the app will use "2.2.3". Although not recommended, you can disable roll forward (patch and/or minor). For more information regarding dotnet host roll-forward and how to configure its behavior, see dotnet host roll forward.

`<Project Sdk` must be set to `Microsoft.NET.Sdk.Web` to use the implicit version `Microsoft.AspNetCore.App`. When `<Project Sdk="Microsoft.NET.Sdk">` (without the trailing `.Web`) is used:

- The following warning is generated:

  Warning NU1604: Project dependency Microsoft.AspNetCore.App does not contain an inclusive lower bound. Include a lower bound in the dependency version to ensure consistent restore results.

- This is a known issue with the .NET Core 2.1 SDK.

Update ASP.NET Core

The `Microsoft.AspNetCore.App` metapackage isn't a traditional package that's updated from NuGet. Similar to `Microsoft.NETCore.App`, `Microsoft.AspNetCore.App` represents a shared runtime, which has special versioning semantics handled outside of NuGet. For more information, see Packages, metapackages and frameworks.

To update ASP.NET Core:

- On development machines and build servers: Download and install the .NET Core SDK.
- On deployment servers: Download and install the .NET Core runtime.

Applications will roll forward to the latest installed version on application restart. It's not necessary to update the `Microsoft.AspNetCore.App` version number in the project file. For more information, see Framework-dependent apps roll forward.

If your application previously used `Microsoft.AspNetCore.All`, see Migrating from Microsoft.AspNetCore.All to Microsoft.AspNetCore.App.

Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0

9/22/2020 • 3 minutes to read • Edit Online

The `Microsoft.AspNetCore.All` metapackage isn't included in ASP.NET Core 3.0 and later. For more information, see this GitHub issue.

> **NOTE**
> We recommend applications targeting ASP.NET Core 2.1 and later use the Microsoft.AspNetCore.App metapackage rather than this package. See Migrating from Microsoft.AspNetCore.All to Microsoft.AspNetCore.App in this article.

This feature requires ASP.NET Core 2.x targeting .NET Core 2.x.

Microsoft.AspNetCore.All is a metapackage that refers to a shared framework. A *shared framework* is a set of assemblies (*.dll* files) that are not in the app's folders. The shared framework must be installed on the machine to run the app. For more information, see The shared framework.

The shared framework that `Microsoft.AspNetCore.All` refers to includes:

- All supported packages by the ASP.NET Core team.
- All supported packages by the Entity Framework Core.
- Internal and 3rd-party dependencies used by ASP.NET Core and Entity Framework Core.

All the features of ASP.NET Core 2.x and Entity Framework Core 2.x are included in the `Microsoft.AspNetCore.All` package. The default project templates targeting ASP.NET Core 2.0 use this package.

The version number of the `Microsoft.AspNetCore.All` metapackage represents the minimum ASP.NET Core version and Entity Framework Core version.

The following *.csproj* file references the `Microsoft.AspNetCore.All` metapackage for ASP.NET Core:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
  </ItemGroup>

</Project>
```

Implicit versioning

In ASP.NET Core 2.1 or later, you can specify the `Microsoft.AspNetCore.All` package reference without a version. When the version isn't specified, an implicit version is specified by the SDK (`Microsoft.NET.Sdk.Web`). We recommend relying on the implicit version specified by the SDK and not explicitly setting the version number on the package reference. If you have questions about this approach, leave a GitHub comment at the Discussion for the Microsoft.AspNetCore.App implicit version.

The implicit version is set to `major.minor.0` for portable apps. The shared framework roll-forward mechanism runs the app on the latest compatible version among the installed shared frameworks. To guarantee the same version is used in development, test, and production, ensure the same version of the shared framework is installed in all environments. For self-contained apps, the implicit version number is set to the `major.minor.patch` of the shared framework bundled in the installed SDK.

Specifying a version number on the `Microsoft.AspNetCore.All` package reference does **not** guarantee that version of the shared framework is chosen. For example, suppose version "2.1.1" is specified, but "2.1.3" is installed. In that case, the app will use "2.1.3". Although not recommended, you can disable roll forward (patch and/or minor). For more information regarding dotnet host roll-forward and how to configure its behavior, see dotnet host roll forward.

The project's SDK must be set to `Microsoft.NET.Sdk.Web` in the project file to use the implicit version of `Microsoft.AspNetCore.All`. When the `Microsoft.NET.Sdk` SDK is specified (`<Project Sdk="Microsoft.NET.Sdk">` at the top of the project file), the following warning is generated:

Warning NU1604: Project dependency Microsoft.AspNetCore.All does not contain an inclusive lower bound. Include a lower bound in the dependency version to ensure consistent restore results.

This is a known issue with the .NET Core 2.1 SDK and will be fixed in the .NET Core 2.2 SDK.

Migrating from Microsoft.AspNetCore.All to Microsoft.AspNetCore.App

The following packages are included in `Microsoft.AspNetCore.All` but not the `Microsoft.AspNetCore.App` package.

- `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServices.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServicesIntegration`
- `Microsoft.AspNetCore.DataProtection.AzureKeyVault`
- `Microsoft.AspNetCore.DataProtection.AzureStorage`
- `Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv`
- `Microsoft.AspNetCore.SignalR.Redis`
- `Microsoft.Data.Sqlite`
- `Microsoft.Data.Sqlite.Core`
- `Microsoft.EntityFrameworkCore.Sqlite`
- `Microsoft.EntityFrameworkCore.Sqlite.Core`
- `Microsoft.Extensions.Caching.Redis`
- `Microsoft.Extensions.Configuration.AzureKeyVault`
- `Microsoft.Extensions.Logging.AzureAppServices`
- `Microsoft.VisualStudio.Web.BrowserLink`

To move from `Microsoft.AspNetCore.All` to `Microsoft.AspNetCore.App`, if your app uses any APIs from the above packages, or packages brought in by those packages, add references to those packages in your project.

Any dependencies of the preceding packages that otherwise aren't dependencies of `Microsoft.AspNetCore.App` are not included implicitly. For example:

- `StackExchange.Redis` as a dependency of `Microsoft.Extensions.Caching.Redis`
- `Microsoft.ApplicationInsights` as a dependency of `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`

Update ASP.NET Core 2.1

We recommend migrating to the `Microsoft.AspNetCore.App` metapackage for 2.1 and later. To keep using the

`Microsoft.AspNetCore.All` metapackage and ensure the latest patch version is deployed:

- On development machines and build servers: Install the latest .NET Core SDK.
- On deployment servers: Install the latest .NET Core runtime. Your app will roll forward to the latest installed version on an application restart.

High-performance logging with LoggerMessage in ASP.NET Core

9/22/2020 • 14 minutes to read • Edit Online

LoggerMessage features create cacheable delegates that require fewer object allocations and reduced computational overhead compared to logger extension methods, such as LogInformation and LogDebug. For high-performance logging scenarios, use the LoggerMessage pattern.

LoggerMessage provides the following performance advantages over Logger extension methods:

- Logger extension methods require "boxing" (converting) value types, such as `int`, into `object`. The LoggerMessage pattern avoids boxing by using static Action fields and extension methods with strongly-typed parameters.
- Logger extension methods must parse the message template (named format string) every time a log message is written. LoggerMessage only requires parsing a template once when the message is defined.

View or download sample code (how to download)

The sample app demonstrates LoggerMessage features with a basic quote tracking system. The app adds and deletes quotes using an in-memory database. As these operations occur, log messages are generated using the LoggerMessage pattern.

LoggerMessage.Define

Define(LogLevel, EventId, String) creates an Action delegate for logging a message. Define overloads permit passing up to six type parameters to a named format string (template).

The string provided to the Define method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend Pascal casing for placeholder names. For example, `{Count}`, `{FirstName}`.

Each log message is an Action held in a static field created by LoggerMessage.Define. For example, the sample app creates a field to describe a log message for a GET request for the Index page (*Internal/LoggerExtensions.cs*):

```
private static readonly Action<ILogger, Exception> _indexPageRequested;
```

For the Action, specify:

- The log level.
- A unique event identifier (EventId) with the name of the static extension method.
- The message template (named format string).

A request for the Index page of the sample app sets the:

- Log level to `Information`.
- Event id to `1` with the name of the `IndexPageRequested` method.
- Message template (named format string) to a string.

```
_indexPageRequested = LoggerMessage.Define(
    LogLevel.Information,
    new EventId(1, nameof(IndexPageRequested)),
    "GET request for Index page");
```

Structured logging stores may use the event name when it's supplied with the event id to enrich logging. For example, Serilog uses the event name.

The Action is invoked through a strongly-typed extension method. The `IndexPageRequested` method logs a message for an Index page GET request in the sample app:

```
public static void IndexPageRequested(this ILogger logger)
{
    _indexPageRequested(logger, null);
}
```

`IndexPageRequested` is called on the logger in the `OnGetAsync` method in *Pages/Index.cshtml.cs*:

```
public async Task OnGetAsync()
{
    _logger.IndexPageRequested();

    Quotes = await _db.Quotes.AsNoTracking().ToListAsync();
}
```

Inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[1]
      => RequestId:0HL90M6E7PHK4:00000001 RequestPath:/ => /Index
      GET request for Index page
```

To pass parameters to a log message, define up to six types when creating the static field. The sample app logs a string when adding a quote by defining a `string` type for the Action field:

```
private static readonly Action<ILogger, string, Exception> _quoteAdded;
```

The delegate's log message template receives its placeholder values from the types provided. The sample app defines a delegate for adding a quote where the quote parameter is a `string`:

```
_quoteAdded = LoggerMessage.Define<string>(
    LogLevel.Information,
    new EventId(2, nameof(QuoteAdded)),
    "Quote added (Quote = '{Quote}')");
```

The static extension method for adding a quote, `QuoteAdded`, receives the quote argument value and passes it to the Action delegate:

```
public static void QuoteAdded(this ILogger logger, string quote)
{
    _quoteAdded(logger, quote, null);
}
```

In the Index page's page model (*Pages/Index.cshtml.cs*), `QuoteAdded` is called to log the message:

```
public async Task<IActionResult> OnPostAddQuoteAsync()
{
    _db.Quotes.Add(Quote);
    await _db.SaveChangesAsync();

    _logger.QuoteAdded(Quote.Text);

    return RedirectToPage();
}
```

Inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[2]
      => RequestId:0HL90M6E7PHK5:0000000A RequestPath:/ => /Index
      Quote added (Quote = 'You can avoid reality, but you cannot avoid the
          consequences of avoiding reality. - Ayn Rand')
```

The sample app implements a try-catch pattern for quote deletion. An informational message is logged for a successful delete operation. An error message is logged for a delete operation when an exception is thrown. The log message for the unsuccessful delete operation includes the exception stack trace (*Internal/LoggerExtensions.cs*):

```
private static readonly Action<ILogger, string, int, Exception> _quoteDeleted;
private static readonly Action<ILogger, int, Exception> _quoteDeleteFailed;
```

```
_quoteDeleted = LoggerMessage.Define<string, int>(
    LogLevel.Information,
    new EventId(4, nameof(QuoteDeleted)),
    "Quote deleted (Quote = '{Quote}' Id = {Id})");

_quoteDeleteFailed = LoggerMessage.Define<int>(
    LogLevel.Error,
    new EventId(5, nameof(QuoteDeleteFailed)),
    "Quote delete failed (Id = {Id})");
```

Note how the exception is passed to the delegate in `QuoteDeleteFailed`:

```
public static void QuoteDeleted(this ILogger logger, string quote, int id)
{
    _quoteDeleted(logger, quote, id, null);
}

public static void QuoteDeleteFailed(this ILogger logger, int id, Exception ex)
{
    _quoteDeleteFailed(logger, id, ex);
}
```

In the page model for the Index page, a successful quote deletion calls the `QuoteDeleted` method on the logger. When a quote isn't found for deletion, an ArgumentNullException is thrown. The exception is trapped by the try-catch statement and logged by calling the `QuoteDeleteFailed` method on the logger in the catch block (*Pages/Index.cshtml.cs*):

```
public async Task<IActionResult> OnPostDeleteQuoteAsync(int id)
{
    try
    {
        var quote = await _db.Quotes.FindAsync(id);
        _db.Quotes.Remove(quote);
        await _db.SaveChangesAsync();

        _logger.QuoteDeleted(quote.Text, id);
    }
    catch (NullReferenceException ex)
    {
        _logger.QuoteDeleteFailed(id, ex);
    }

    return RedirectToPage();
}
```

When a quote is successfully deleted, inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:00000016 RequestPath:/ => /Index
      Quote deleted (Quote = 'You can avoid reality, but you cannot avoid the
          consequences of avoiding reality. - Ayn Rand' Id = 1)
```

When quote deletion fails, inspect the app's console output. Note that the exception is included in the log message:

```
LoggerMessageSample.Pages.IndexModel: Error: Quote delete failed (Id = 999)

System.NullReferenceException: Object reference not set to an instance of an object.
   at lambda_method(Closure , ValueBuffer )
   at System.Linq.Enumerable.SelectEnumerableIterator`2.MoveNext()
   at
Microsoft.EntityFrameworkCore.InMemory.Query.Internal.InMemoryShapedQueryCompilingExpressionVisitor.AsyncQuery
ingEnumerable`1.AsyncEnumerator.MoveNextAsync()
   at Microsoft.EntityFrameworkCore.Query.ShapedQueryCompilingExpressionVisitor.SingleOrDefaultAsync[TSource]
(IAsyncEnumerable`1 asyncEnumerable, CancellationToken cancellationToken)
   at Microsoft.EntityFrameworkCore.Query.ShapedQueryCompilingExpressionVisitor.SingleOrDefaultAsync[TSource]
(IAsyncEnumerable`1 asyncEnumerable, CancellationToken cancellationToken)
   at LoggerMessageSample.Pages.IndexModel.OnPostDeleteQuoteAsync(Int32 id) in
c:\Users\guard\Documents\GitHub\Docs\aspnetcore\fundamentals\logging\loggermessage\samples\3.x\LoggerMessageSa
mple\Pages\Index.cshtml.cs:line 77
```

LoggerMessage.DefineScope

DefineScope(String) creates a Func<TResult> delegate for defining a log scope. DefineScope overloads permit passing up to three type parameters to a named format string (template).

As is the case with the Define method, the string provided to the DefineScope method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend Pascal casing for placeholder names. For example, `{Count}`, `{FirstName}`.

Define a log scope to apply to a series of log messages using the DefineScope method.

The sample app has a **Clear All** button for deleting all of the quotes in the database. The quotes are deleted by removing them one at a time. Each time a quote is deleted, the `QuoteDeleted` method is called on the logger. A log scope is added to these log messages.

Enable `IncludeScopes` in the console logger section of *appsettings.json*:

```
{
  "Logging": {
    "Console": {
      "IncludeScopes": true
    },
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

To create a log scope, add a field to hold a Func<TResult> delegate for the scope. The sample app creates a field called `_allQuotesDeletedScope` (*Internal/LoggerExtensions.cs*):

```
private static Func<ILogger, int, IDisposable> _allQuotesDeletedScope;
```

Use DefineScope to create the delegate. Up to three types can be specified for use as template arguments when the delegate is invoked. The sample app uses a message template that includes the number of deleted quotes (an `int` type):

```
_allQuotesDeletedScope =
    LoggerMessage.DefineScope<int>("All quotes deleted (Count = {Count})");
```

Provide a static extension method for the log message. Include any type parameters for named properties that appear in the message template. The sample app takes in a `count` of quotes to delete and returns `_allQuotesDeletedScope`:

```
public static IDisposable AllQuotesDeletedScope(
    this ILogger logger, int count)
{
    return _allQuotesDeletedScope(logger, count);
}
```

The scope wraps the logging extension calls in a using block:

```
public async Task<IActionResult> OnPostDeleteAllQuotesAsync()
{
    var quoteCount = await _db.Quotes.CountAsync();

    using (_logger.AllQuotesDeletedScope(quoteCount))
    {
        foreach (Quote quote in _db.Quotes)
        {
            _db.Quotes.Remove(quote);

            _logger.QuoteDeleted(quote.Text, quote.Id);
        }
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

Inspect the log messages in the app's console output. The following result shows three quotes deleted with the log scope message included:

```
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index =>
          All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 1' Id = 2)
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index =>
          All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 2' Id = 3)
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index =>
          All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 3' Id = 4)
```

LoggerMessage features create cacheable delegates that require fewer object allocations and reduced computational overhead compared to logger extension methods, such as LogInformation and LogDebug. For high-performance logging scenarios, use the LoggerMessage pattern.

LoggerMessage provides the following performance advantages over Logger extension methods:

- Logger extension methods require "boxing" (converting) value types, such as `int`, into `object`. The LoggerMessage pattern avoids boxing by using static Action fields and extension methods with strongly-typed parameters.
- Logger extension methods must parse the message template (named format string) every time a log message is written. LoggerMessage only requires parsing a template once when the message is defined.

View or download sample code (how to download)

The sample app demonstrates LoggerMessage features with a basic quote tracking system. The app adds and deletes quotes using an in-memory database. As these operations occur, log messages are generated using the LoggerMessage pattern.

LoggerMessage.Define

Define(LogLevel, EventId, String) creates an Action delegate for logging a message. Define overloads permit passing up to six type parameters to a named format string (template).

The string provided to the Define method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend Pascal casing for placeholder names. For example, `{Count}`, `{FirstName}`.

Each log message is an Action held in a static field created by LoggerMessage.Define. For example, the sample app creates a field to describe a log message for a GET request for the Index page (*Internal/LoggerExtensions.cs*):

```
private static readonly Action<ILogger, Exception> _indexPageRequested;
```

For the Action, specify:

- The log level.
- A unique event identifier (EventId) with the name of the static extension method.
- The message template (named format string).

A request for the Index page of the sample app sets the:

- Log level to `Information`.
- Event id to `1` with the name of the `IndexPageRequested` method.
- Message template (named format string) to a string.

```
_indexPageRequested = LoggerMessage.Define(
    LogLevel.Information,
    new EventId(1, nameof(IndexPageRequested)),
    "GET request for Index page");
```

Structured logging stores may use the event name when it's supplied with the event id to enrich logging. For example, Serilog uses the event name.

The Action is invoked through a strongly-typed extension method. The `IndexPageRequested` method logs a message for an Index page GET request in the sample app:

```
public static void IndexPageRequested(this ILogger logger)
{
    _indexPageRequested(logger, null);
}
```

`IndexPageRequested` is called on the logger in the `OnGetAsync` method in *Pages/Index.cshtml.cs*:

```
public async Task OnGetAsync()
{
    _logger.IndexPageRequested();

    Quotes = await _db.Quotes.AsNoTracking().ToListAsync();
}
```

Inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[1]
      => RequestId:0HL90M6E7PHK4:00000001 RequestPath:/ => /Index
      GET request for Index page
```

To pass parameters to a log message, define up to six types when creating the static field. The sample app logs a string when adding a quote by defining a `string` type for the Action field:

```
private static readonly Action<ILogger, string, Exception> _quoteAdded;
```

The delegate's log message template receives its placeholder values from the types provided. The sample app defines a delegate for adding a quote where the quote parameter is a `string`:

```
_quoteAdded = LoggerMessage.Define<string>(
    LogLevel.Information,
    new EventId(2, nameof(QuoteAdded)),
    "Quote added (Quote = '{Quote}')");
```

The static extension method for adding a quote, `QuoteAdded`, receives the quote argument value and passes it to the Action delegate:

```
public static void QuoteAdded(this ILogger logger, string quote)
{
    _quoteAdded(logger, quote, null);
}
```

In the Index page's page model (*Pages/Index.cshtml.cs*), `QuoteAdded` is called to log the message:

```
public async Task<IActionResult> OnPostAddQuoteAsync()
{
    _db.Quotes.Add(Quote);
    await _db.SaveChangesAsync();

    _logger.QuoteAdded(Quote.Text);

    return RedirectToPage();
}
```

Inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[2]
      => RequestId:0HL90M6E7PHK5:0000000A RequestPath:/ => /Index
      Quote added (Quote = 'You can avoid reality, but you cannot avoid the
          consequences of avoiding reality. - Ayn Rand')
```

The sample app implements a try-catch pattern for quote deletion. An informational message is logged for a successful delete operation. An error message is logged for a delete operation when an exception is thrown. The log message for the unsuccessful delete operation includes the exception stack trace (*Internal/LoggerExtensions.cs*):

```
private static readonly Action<ILogger, string, int, Exception> _quoteDeleted;
private static readonly Action<ILogger, int, Exception> _quoteDeleteFailed;
```

```
_quoteDeleted = LoggerMessage.Define<string, int>(
    LogLevel.Information,
    new EventId(4, nameof(QuoteDeleted)),
    "Quote deleted (Quote = '{Quote}' Id = {Id})");

_quoteDeleteFailed = LoggerMessage.Define<int>(
    LogLevel.Error,
    new EventId(5, nameof(QuoteDeleteFailed)),
    "Quote delete failed (Id = {Id})");
```

Note how the exception is passed to the delegate in `QuoteDeleteFailed`:

```
public static void QuoteDeleted(this ILogger logger, string quote, int id)
{
    _quoteDeleted(logger, quote, id, null);
}

public static void QuoteDeleteFailed(this ILogger logger, int id, Exception ex)
{
    _quoteDeleteFailed(logger, id, ex);
}
```

In the page model for the Index page, a successful quote deletion calls the `QuoteDeleted` method on the logger. When a quote isn't found for deletion, an ArgumentNullException is thrown. The exception is trapped by the try-

catch statement and logged by calling the `QuoteDeleteFailed` method on the logger in the catch block (*Pages/Index.cshtml.cs*):

```
public async Task<IActionResult> OnPostDeleteQuoteAsync(int id)
{
    var quote = await _db.Quotes.FindAsync(id);

    // DO NOT use this approach in production code!
    // You should check quote to see if it's null before removing
    // it and saving changes to the database. A try-catch is used
    // here for demonstration purposes of LoggerMessage features.
    try
    {
        _db.Quotes.Remove(quote);
        await _db.SaveChangesAsync();

        _logger.QuoteDeleted(quote.Text, id);
    }
    catch (ArgumentNullException ex)
    {
        _logger.QuoteDeleteFailed(id, ex);
    }

    return RedirectToPage();
}
```

When a quote is successfully deleted, inspect the app's console output:

```
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:00000016 RequestPath:/ => /Index
      Quote deleted (Quote = 'You can avoid reality, but you cannot avoid the
          consequences of avoiding reality. - Ayn Rand' Id = 1)
```

When quote deletion fails, inspect the app's console output. Note that the exception is included in the log message:

```
fail: LoggerMessageSample.Pages.IndexModel[5]
      => RequestId:0HL90M6E7PHK5:00000010 RequestPath:/ => /Index
      Quote delete failed (Id = 999)
System.ArgumentNullException: Value cannot be null.
Parameter name: entity
   at Microsoft.EntityFrameworkCore.Utilities.Check.NotNull[T]
       (T value, String parameterName)
   at Microsoft.EntityFrameworkCore.DbContext.Remove[TEntity](TEntity entity)
   at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.Remove(TEntity entity)
   at LoggerMessageSample.Pages.IndexModel.<OnPostDeleteQuoteAsync>d__14.MoveNext()
      in <PATH>\sample\Pages\Index.cshtml.cs:line 87
```

LoggerMessage.DefineScope

DefineScope(String) creates a Func<TResult> delegate for defining a log scope. DefineScope overloads permit passing up to three type parameters to a named format string (template).

As is the case with the Define method, the string provided to the DefineScope method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend Pascal casing for placeholder names. For example, `{Count}`, `{FirstName}`.

Define a log scope to apply to a series of log messages using the DefineScope method.

The sample app has a **Clear All** button for deleting all of the quotes in the database. The quotes are deleted by

removing them one at a time. Each time a quote is deleted, the `QuoteDeleted` method is called on the logger. A log scope is added to these log messages.

Enable `IncludeScopes` in the console logger section of *appsettings.json*:

```
{
  "Logging": {
    "Console": {
      "IncludeScopes": true
    },
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

To create a log scope, add a field to hold a Func<TResult> delegate for the scope. The sample app creates a field called `_allQuotesDeletedScope` (*Internal/LoggerExtensions.cs*):

```
private static Func<ILogger, int, IDisposable> _allQuotesDeletedScope;
```

Use DefineScope to create the delegate. Up to three types can be specified for use as template arguments when the delegate is invoked. The sample app uses a message template that includes the number of deleted quotes (an `int` type):

```
_allQuotesDeletedScope =
    LoggerMessage.DefineScope<int>("All quotes deleted (Count = {Count})");
```

Provide a static extension method for the log message. Include any type parameters for named properties that appear in the message template. The sample app takes in a `count` of quotes to delete and returns `_allQuotesDeletedScope`:

```
public static IDisposable AllQuotesDeletedScope(
    this ILogger logger, int count)
{
    return _allQuotesDeletedScope(logger, count);
}
```

The scope wraps the logging extension calls in a using block:

```
public async Task<IActionResult> OnPostDeleteAllQuotesAsync()
{
    var quoteCount = await _db.Quotes.CountAsync();

    using (_logger.AllQuotesDeletedScope(quoteCount))
    {
        foreach (Quote quote in _db.Quotes)
        {
            _db.Quotes.Remove(quote);

            _logger.QuoteDeleted(quote.Text, quote.Id);
        }
        await _db.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

Inspect the log messages in the app's console output. The following result shows three quotes deleted with the log scope message included:

```
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index =>
          All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 1' Id = 2)
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index =>
          All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 2' Id = 3)
info: LoggerMessageSample.Pages.IndexModel[4]
      => RequestId:0HL90M6E7PHK5:0000002E RequestPath:/ => /Index =>
          All quotes deleted (Count = 3)
      Quote deleted (Quote = 'Quote 3' Id = 4)
```

Additional resources

- Logging

Develop ASP.NET Core apps using a file watcher

9/22/2020 • 5 minutes to read • Edit Online

By Rick Anderson and Victor Hurdugaci

`dotnet watch` is a tool that runs a .NET Core CLI command when source files change. For example, a file change can trigger compilation, test execution, or deployment.

This tutorial uses an existing web API with two endpoints: one that returns a sum and one that returns a product. The product method has a bug, which is fixed in this tutorial.

Download the sample app. It consists of two projects: *WebApp* (an ASP.NET Core web API) and *WebAppTests* (unit tests for the web API).

In a command shell, navigate to the *WebApp* folder. Run the following command:

```
dotnet run
```

NOTE

You can use `dotnet run --project <PROJECT>` to specify a project to run. For example, running `dotnet run --project WebApp` from the root of the sample app will also run the *WebApp* project.

The console output shows messages similar to the following (indicating that the app is running and awaiting requests):

```
$ dotnet run
Hosting environment: Development
Content root path: C:/Docs/aspnetcore/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

In a web browser, navigate to `http://localhost:<port number>/api/math/sum?a=4&b=5`. You should see the result of `9`.

Navigate to the product API (`http://localhost:<port number>/api/math/product?a=4&b=5`). It returns `9`, not `20` as you'd expect. That problem is fixed later in the tutorial.

Add `dotnet watch` to a project

The `dotnet watch` file watcher tool is included with version 2.1.300 of the .NET Core SDK. The following steps are required when using an earlier version of the .NET Core SDK.

1. Add a `Microsoft.DotNet.Watcher.Tools` package reference to the *.csproj* file:

   ```
   <ItemGroup>
       <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools" Version="2.0.0" />
   </ItemGroup>
   ```

2. Install the `Microsoft.DotNet.Watcher.Tools` package by running the following command:

```
dotnet restore
```

Run .NET Core CLI commands using `dotnet watch`

Any .NET Core CLI command can be run with `dotnet watch`. For example:

COMMAND	COMMAND WITH WATCH
dotnet run	dotnet watch run
dotnet run -f netcoreapp3.1	dotnet watch run -f netcoreapp3.1
dotnet run -f netcoreapp3.1 -- --arg1	dotnet watch run -f netcoreapp3.1 -- --arg1
dotnet test	dotnet watch test

Run `dotnet watch run` in the *WebApp* folder. The console output indicates `watch` has started.

Running `dotnet watch run` on a web app launches a browser that navigates to the app's URL once ready. `dotnet watch` does this by reading the app's console output and waiting for the the ready message displayed by WebHost.

`dotnet watch` refreshes the browser when it detects changes to watched files. To do this, the watch command injects a middleware to the app that modifies HTML responses created by the app. The middleware adds a JavaScript script block to the page that allows `dotnet watch` to instruct the browser to refresh. Currently, changes to all watched files, including static content such as *.html* and *.css* files cause the app to be rebuilt.

`dotnet watch`:

- Only watches files that impact builds by default.
- Any additionally watched files (via configuration) still results in a build taking place.

For more information on configuration, see dotnet-watch configuration in this document

> **NOTE**
> You can use `dotnet watch --project <PROJECT>` to specify a project to watch. For example, running `dotnet watch --project WebApp run` from the root of the sample app will also run and watch the *WebApp* project.

Make changes with `dotnet watch`

Make sure `dotnet watch` is running.

Fix the bug in the `Product` method of *MathController.cs* so it returns the product and not the sum:

```
public static int Product(int a, int b)
{
    return a * b;
}
```

Save the file. The console output indicates that `dotnet watch` detected a file change and restarted the app.

Verify `http://localhost:<port number>/api/math/product?a=4&b=5` returns the correct result.

Run tests using `dotnet watch`

1. Change the `Product` method of *MathController.cs* back to returning the sum. Save the file.
2. In a command shell, navigate to the *WebAppTests* folder.
3. Run dotnet restore.
4. Run `dotnet watch test`. Its output indicates that a test failed and that the watcher is awaiting file changes:

```
Total tests: 2. Passed: 1. Failed: 1. Skipped: 0.
Test Run Failed.
```

5. Fix the `Product` method code so it returns the product. Save the file.

`dotnet watch` detects the file change and reruns the tests. The console output indicates the tests passed.

Customize files list to watch

By default, `dotnet-watch` tracks all files matching the following glob patterns:

- `**/*.cs`
- `*.csproj`
- `**/*.resx`

More items can be added to the watch list by editing the *.csproj* file. Items can be specified individually or by using glob patterns.

```
<ItemGroup>
    <!-- extends watching group to include *.js files -->
    <Watch Include="**\*.js" Exclude="node_modules\**\*;**\*.js.map;obj\**\*;bin\**\*" />
</ItemGroup>
```

Opt-out of files to be watched

`dotnet-watch` can be configured to ignore its default settings. To ignore specific files, add the `Watch="false"` attribute to an item's definition in the *.csproj* file:

```
<ItemGroup>
    <!-- exclude Generated.cs from dotnet-watch -->
    <Compile Include="Generated.cs" Watch="false" />

    <!-- exclude Strings.resx from dotnet-watch -->
    <EmbeddedResource Include="Strings.resx" Watch="false" />

    <!-- exclude changes in this referenced project -->
    <ProjectReference Include="..\ClassLibrary1\ClassLibrary1.csproj" Watch="false" />
</ItemGroup>
```

Custom watch projects

`dotnet-watch` isn't restricted to C# projects. Custom watch projects can be created to handle different scenarios. Consider the following project layout:

- **test/**
  - *UnitTests/UnitTests.csproj*

- *IntegrationTests/IntegrationTests.csproj*

If the goal is to watch both projects, create a custom project file configured to watch both projects:

```
<Project>
    <ItemGroup>
        <TestProjects Include="**\*.csproj" />
        <Watch Include="**\*.cs" />
    </ItemGroup>

    <Target Name="Test">
        <MSBuild Targets="VSTest" Projects="@(TestProjects)" />
    </Target>

    <Import Project="$(MSBuildExtensionsPath)\Microsoft.Common.targets" />
</Project>
```

To start file watching on both projects, change to the *test* folder. Execute the following command:

```
dotnet watch msbuild /t:Test
```

VSTest executes when any file changes in either test project.

dotnet-watch configuration

Some configuration options can be passed to `dotnet watch` through environment variables. The available variables are:

SETTING	DESCRIPTION
`DOTNET_USE_POLLING_FILE_WATCHER`	If set to "1" or "true", `dotnet watch` uses a polling file watcher instead of CoreFx's `FileSystemWatcher`. Used when watching files on network shares or Docker mounted volumes.
`DOTNET_WATCH_SUPPRESS_MSBUILD_INCREMENTALISM`	By default, `dotnet watch` optimizes the build by avoiding certain operations such as running restore or re-evaluating the set of watched files on every file change. If set to "1" or "true", these optimizations are disabled.
`DOTNET_WATCH_SUPPRESS_LAUNCH_BROWSER`	`dotnet watch run` attempts to launch browsers for web apps with `launchBrowser` configured in *launchSettings.json*. If set to "1" or "true", this behavior is suppressed.
`DOTNET_WATCH_SUPPRESS_BROWSER_REFRESH`	`dotnet watch run` attempts to refresh browsers when it detects file changes. If set to "1" or "true", this behavior is suppressed. This behavior is also suppressed if `DOTNET_WATCH_SUPPRESS_LAUNCH_BROWSER` is set.

`dotnet-watch` in GitHub

`dotnet-watch` is part of the GitHub dotnet/AspNetCore repository.

Factory-based middleware activation in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

IMiddlewareFactory/IMiddleware is an extensibility point for middleware activation.

UseMiddleware extension methods check if a middleware's registered type implements IMiddleware. If it does, the IMiddlewareFactory instance registered in the container is used to resolve the IMiddleware implementation instead of using the convention-based middleware activation logic. The middleware is registered as a scoped or transient service in the app's service container.

Benefits:

- Activation per client request (injection of scoped services)
- Strong typing of middleware

IMiddleware is activated per client request (connection), so scoped services can be injected into the middleware's constructor.

View or download sample code (how to download)

IMiddleware

IMiddleware defines middleware for the app's request pipeline. The InvokeAsync(HttpContext, RequestDelegate) method handles requests and returns a Task that represents the execution of the middleware.

Middleware activated by convention:

```
public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context, AppDbContext db)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "ConventionalMiddleware",
                    Value = keyValue
                });

            await db.SaveChangesAsync();
        }

        await _next(context);
    }
}
```

Middleware activated by MiddlewareFactory:

```
public class FactoryActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public FactoryActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "FactoryActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }

        await next(context);
    }
}
```

Extensions are created for the middlewares:

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseConventionalMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<ConventionalMiddleware>();
    }

    public static IApplicationBuilder UseFactoryActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<FactoryActivatedMiddleware>();
    }
}
```

It isn't possible to pass objects to the factory-activated middleware with UseMiddleware:

```
public static IApplicationBuilder UseFactoryActivatedMiddleware(
    this IApplicationBuilder builder, bool option)
{
    // Passing 'option' as an argument throws a NotSupportedException at runtime.
    return builder.UseMiddleware<FactoryActivatedMiddleware>(option);
}
```

The factory-activated middleware is added to the built-in container in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options =>
        options.UseInMemoryDatabase("InMemoryDb"));

    services.AddTransient<FactoryActivatedMiddleware>();

    services.AddRazorPages();
}
```

Both middlewares are registered in the request processing pipeline in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseConventionalMiddleware();
    app.UseFactoryActivatedMiddleware();

    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware. The middleware factory implementation is registered in the container as a scoped service.

The default IMiddlewareFactory implementation, MiddlewareFactory, is found in the Microsoft.AspNetCore.Http package.

IMiddlewareFactory/IMiddleware is an extensibility point for middleware activation.

UseMiddleware extension methods check if a middleware's registered type implements IMiddleware. If it does, the IMiddlewareFactory instance registered in the container is used to resolve the IMiddleware implementation instead of using the convention-based middleware activation logic. The middleware is registered as a scoped or transient service in the app's service container.

Benefits:

- Activation per client request (injection of scoped services)
- Strong typing of middleware

IMiddleware is activated per client request (connection), so scoped services can be injected into the middleware's constructor.

View or download sample code (how to download)

IMiddleware

IMiddleware defines middleware for the app's request pipeline. The InvokeAsync(HttpContext, RequestDelegate) method handles requests and returns a Task that represents the execution of the middleware.

Middleware activated by convention:

```
public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context, AppDbContext db)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "ConventionalMiddleware",
                    Value = keyValue
                });

            await db.SaveChangesAsync();
        }

        await _next(context);
    }
}
```

Middleware activated by MiddlewareFactory:

```
public class FactoryActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public FactoryActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "FactoryActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }

        await next(context);
    }
}
```

Extensions are created for the middlewares:

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseConventionalMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<ConventionalMiddleware>();
    }

    public static IApplicationBuilder UseFactoryActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<FactoryActivatedMiddleware>();
    }
}
```

It isn't possible to pass objects to the factory-activated middleware with UseMiddleware:

```
public static IApplicationBuilder UseFactoryActivatedMiddleware(
    this IApplicationBuilder builder, bool option)
{
    // Passing 'option' as an argument throws a NotSupportedException at runtime.
    return builder.UseMiddleware<FactoryActivatedMiddleware>(option);
}
```

The factory-activated middleware is added to the built-in container in `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options =>
        options.UseInMemoryDatabase("InMemoryDb"));

    services.AddTransient<FactoryActivatedMiddleware>();

    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Both middlewares are registered in the request processing pipeline in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseConventionalMiddleware();
    app.UseFactoryActivatedMiddleware();

    app.UseStaticFiles();
    app.UseMvc();
}
```

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware. The middleware factory implementation is registered in the container as a scoped service.

The default IMiddlewareFactory implementation, MiddlewareFactory, is found in the Microsoft.AspNetCore.Http package.

Additional resources

- ASP.NET Core Middleware
- Middleware activation with a third-party container in ASP.NET Core

Middleware activation with a third-party container in ASP.NET Core

9/22/2020 • 4 minutes to read • Edit Online

This article demonstrates how to use IMiddlewareFactory and IMiddleware as an extensibility point for middleware activation with a third-party container. For introductory information on `IMiddlewareFactory` and `IMiddleware`, see Factory-based middleware activation in ASP.NET Core.

View or download sample code (how to download)

The sample app demonstrates middleware activation by an `IMiddlewareFactory` implementation, `SimpleInjectorMiddlewareFactory`. The sample uses the Simple Injector dependency injection (DI) container.

The sample's middleware implementation records the value provided by a query string parameter (`key`). The middleware uses an injected database context (a scoped service) to record the query string value in an in-memory database.

> **NOTE**
> The sample app uses Simple Injector purely for demonstration purposes. Use of Simple Injector isn't an endorsement. Middleware activation approaches described in the Simple Injector documentation and GitHub issues are recommended by the maintainers of Simple Injector. For more information, see the Simple Injector documentation and Simple Injector GitHub repository.

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware.

In the sample app, a middleware factory is implemented to create an `SimpleInjectorActivatedMiddleware` instance. The middleware factory uses the Simple Injector container to resolve the middleware:

```
public class SimpleInjectorMiddlewareFactory : IMiddlewareFactory
{
    private readonly Container _container;

    public SimpleInjectorMiddlewareFactory(Container container)
    {
        _container = container;
    }

    public IMiddleware Create(Type middlewareType)
    {
        return _container.GetInstance(middlewareType) as IMiddleware;
    }

    public void Release(IMiddleware middleware)
    {
        // The container is responsible for releasing resources.
    }
}
```

IMiddleware

IMiddleware defines middleware for the app's request pipeline.

Middleware activated by an `IMiddlewareFactory` implementation (*Middleware/SimpleInjectorActivatedMiddleware.cs*):

```
public class SimpleInjectorActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public SimpleInjectorActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "SimpleInjectorActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }

        await next(context);
    }
}
```

An extension is created for the middleware (*Middleware/MiddlewareExtensions.cs*):

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleInjectorActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleInjectorActivatedMiddleware>();
    }
}
```

`Startup.ConfigureServices` must perform several tasks:

- Set up the Simple Injector container.
- Register the factory and middleware.
- Make the app's database context available from the Simple Injector container.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    // Replace the default middleware factory with the
    // SimpleInjectorMiddlewareFactory.
    services.AddTransient<IMiddlewareFactory>(_ =>
    {
        return new SimpleInjectorMiddlewareFactory(_container);
    });

    // Wrap ASP.NET Core requests in a Simple Injector execution
    // context.
    services.UseSimpleInjectorAspNetRequestScoping(_container);

    // Provide the database context from the Simple
    // Injector container whenever it's requested from
    // the default service container.
    services.AddScoped<AppDbContext>(provider =>
        _container.GetInstance<AppDbContext>());

    _container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

    _container.Register<AppDbContext>(() =>
    {
        var optionsBuilder = new DbContextOptionsBuilder<DbContext>();
        optionsBuilder.UseInMemoryDatabase("InMemoryDb");
        return new AppDbContext(optionsBuilder.Options);
    }, Lifestyle.Scoped);

    _container.Register<SimpleInjectorActivatedMiddleware>();

    _container.Verify();
}
```

The middleware is registered in the request processing pipeline in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseSimpleInjectorActivatedMiddleware();

    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

This article demonstrates how to use IMiddlewareFactory and IMiddleware as an extensibility point for middleware activation with a third-party container. For introductory information on `IMiddlewareFactory` and `IMiddleware`, see Factory-based middleware activation in ASP.NET Core.

View or download sample code (how to download)

The sample app demonstrates middleware activation by an `IMiddlewareFactory` implementation, `SimpleInjectorMiddlewareFactory`. The sample uses the Simple Injector dependency injection (DI) container.

The sample's middleware implementation records the value provided by a query string parameter (`key`). The middleware uses an injected database context (a scoped service) to record the query string value in an in-memory database.

> **NOTE**
>
> The sample app uses Simple Injector purely for demonstration purposes. Use of Simple Injector isn't an endorsement. Middleware activation approaches described in the Simple Injector documentation and GitHub issues are recommended by the maintainers of Simple Injector. For more information, see the Simple Injector documentation and Simple Injector GitHub repository.

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware.

In the sample app, a middleware factory is implemented to create an `SimpleInjectorActivatedMiddleware` instance. The middleware factory uses the Simple Injector container to resolve the middleware:

```
public class SimpleInjectorMiddlewareFactory : IMiddlewareFactory
{
    private readonly Container _container;

    public SimpleInjectorMiddlewareFactory(Container container)
    {
        _container = container;
    }

    public IMiddleware Create(Type middlewareType)
    {
        return _container.GetInstance(middlewareType) as IMiddleware;
    }

    public void Release(IMiddleware middleware)
    {
        // The container is responsible for releasing resources.
    }
}
```

IMiddleware

IMiddleware defines middleware for the app's request pipeline.

Middleware activated by an `IMiddlewareFactory` implementation (*Middleware/SimpleInjectorActivatedMiddleware.cs*):

```
public class SimpleInjectorActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public SimpleInjectorActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation = "SimpleInjectorActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }

        await next(context);
    }
}
```

An extension is created for the middleware (*Middleware/MiddlewareExtensions.cs*):

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleInjectorActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleInjectorActivatedMiddleware>();
    }
}
```

`Startup.ConfigureServices` must perform several tasks:

- Set up the Simple Injector container.
- Register the factory and middleware.
- Make the app's database context available from the Simple Injector container.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

    // Replace the default middleware factory with the
    // SimpleInjectorMiddlewareFactory.
    services.AddTransient<IMiddlewareFactory>(_ =>
    {
        return new SimpleInjectorMiddlewareFactory(_container);
    });

    // Wrap ASP.NET Core requests in a Simple Injector execution
    // context.
    services.UseSimpleInjectorAspNetRequestScoping(_container);

    // Provide the database context from the Simple
    // Injector container whenever it's requested from
    // the default service container.
    services.AddScoped<AppDbContext>(provider =>
        _container.GetInstance<AppDbContext>());

    _container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

    _container.Register<AppDbContext>(() =>
    {
        var optionsBuilder = new DbContextOptionsBuilder<DbContext>();
        optionsBuilder.UseInMemoryDatabase("InMemoryDb");
        return new AppDbContext(optionsBuilder.Options);
    }, Lifestyle.Scoped);

    _container.Register<SimpleInjectorActivatedMiddleware>();

    _container.Verify();
}
```

The middleware is registered in the request processing pipeline in `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseSimpleInjectorActivatedMiddleware();

    app.UseStaticFiles();
    app.UseMvc();
}
```

Additional resources

- Middleware
- Factory-based middleware activation
- Simple Injector GitHub repository
- Simple Injector documentation

Migrate from ASP.NET Core 3.1 to 5.0

9/22/2020 • 2 minutes to read • Edit Online

By Scott Addie

This article explains how to update an existing ASP.NET Core 3.1 project to ASP.NET Core 5.0.

> **IMPORTANT**
> ASP.NET Core 5.0 is currently in preview.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.8 or later with the **ASP.NET and web development** workload
- .NET 5.0 SDK or later

Update .NET Core SDK version in global.json

If you rely upon a global.json file to target a specific .NET Core SDK version, update the `version` property to the .NET 5.0 SDK version that's installed. For example:

```
{
  "sdk": {
-    "version": "3.1.200"
+    "version": "5.0.100-rc.1.20452.10"
  }
}
```

Update the target framework

If updating a Blazor WebAssembly project, skip to the Update Blazor WebAssembly projects section. For any other ASP.NET Core project type, update the project file's Target Framework Moniker (TFM) to `net5.0`:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.1</TargetFramework>
+    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Update Blazor WebAssembly projects

For Blazor WebAssembly projects, apply the following changes in the project file:

1. Update the SDK from `Microsoft.NET.Sdk.Web` to `Microsoft.NET.Sdk.BlazorWebAssembly`:

```
- <Project Sdk="Microsoft.NET.Sdk.Web">
+ <Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
```

2. Update the following properties:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-     <TargetFramework>netstandard2.1</TargetFramework>
-     <RazorLangVersion>3.0</RazorLangVersion>
+     <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
```

3. Remove the package reference to Microsoft.AspNetCore.Components.WebAssembly.Build:

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.Build" Version="3.2.1"
PrivateAssets="all" />
```

Update package references

In the project file, update each Microsoft.AspNetCore.*, Microsoft.Extensions.*, and System.Net.Http.Json package reference's `Version` attribute to 5.0.0 or later. For example:

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.JsonPatch" Version="3.1.6" />
-    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions" Version="3.1.6" />
-    <PackageReference Include="System.Net.Http.Json" Version="3.2.1" />
+    <PackageReference Include="Microsoft.AspNetCore.JsonPatch" Version="5.0.0-rc.1.*" />
+    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions" Version="5.0.0-rc.1.*" />
+    <PackageReference Include="System.Net.Http.Json" Version="5.0.0-rc.1.*" />
</ItemGroup>
```

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 5.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 3.1 and 5.0:

```
- docker pull mcr.microsoft.com/dotnet/core/aspnet:3.1
+ docker pull mcr.microsoft.com/dotnet/aspnet:5.0
```

As part of the move to ".NET" as the product name, the Docker images moved from the `mcr.microsoft.com/dotnet/core` repositories to `mcr.microsoft.com/dotnet`. For more information, see dotnet/dotnet-docker#1939.

Review breaking changes

For breaking changes from .NET Core 3.1 to .NET 5.0, see Breaking changes for migration from version 3.1 to 5.0. ASP.NET Core and Entity Framework Core are also included in the list.

Migrate from ASP.NET Core 3.0 to 3.1

9/22/2020 • 2 minutes to read • Edit Online

By Scott Addie

This article explains how to update an existing ASP.NET Core 3.0 project to ASP.NET Core 3.1.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Update .NET Core SDK version in global.json

If you rely upon a global.json file to target a specific .NET Core SDK version, update the `version` property to the 3.1 SDK version that's installed. For example:

```
{
  "sdk": {
-    "version": "3.0.101"
+    "version": "3.1.101"
  }
}
```

Update the target framework

In the project file, update the Target Framework Moniker (TFM) to `netcoreapp3.1`:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.0</TargetFramework>
+    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

Update package references

In the project file, update each `Microsoft.AspNetCore.*` package reference's `Version` attribute to 3.1.0 or later. For example:

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson" Version="3.0.0" />
-    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.0.0"
Condition="'$(Configuration)' == 'Debug'" />
+    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson" Version="3.1.1" />
+    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.1.1"
Condition="'$(Configuration)' == 'Debug'" />
</ItemGroup>
```

Update Docker images

For apps using Docker, use a base image that includes ASP.NET Core 3.1. For example:

```
docker pull mcr.microsoft.com/dotnet/core/aspnet:3.1
```

React to SameSite cookie changes

The `SameSite` attribute implementations for HTTP cookies changed between ASP.NET Core 3.0 and 3.1. For actions to be taken, see the following resources:

- Work with SameSite cookies in ASP.NET Core
- aspnet/Announcements#390
- Upcoming SameSite cookie changes in ASP.NET and ASP.NET Core

Review breaking changes

Review 3.0-to-3.1 breaking changes across .NET Core, ASP.NET Core, and Entity Framework Core at Breaking changes for migration from version 3.0 to 3.1.

Optional changes

The following changes are optional.

Use the Component Tag Helper

ASP.NET Core 3.1 introduces a `Component` Tag Helper. The Tag Helper can replace the `RenderComponentAsync<TComponent>` HTML helper method in a Blazor project. For example:

```
- @(await Html.RenderComponentAsync<Counter>(RenderMode.ServerPrerendered, new { IncrementAmount = 10 }))
+ <component type="typeof(Counter)" render-mode="ServerPrerendered" param-IncrementAmount="10" />
```

For more information, see Integrate ASP.NET Core Razor components into Razor Pages and MVC apps.

Migrate from ASP.NET Core 2.2 to 3.0

9/22/2020 • 27 minutes to read • Edit Online

By Scott Addie and Rick Anderson

This article explains how to update an existing ASP.NET Core 2.2 project to ASP.NET Core 3.0. It might be helpful to create a new ASP.NET Core 3.0 project to:

- Compare with the ASP.NET Core 2.2 code.
- Copy the relevant changes to your ASP.NET Core 3.0 project.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK or later

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to the 3.0 version installed on your machine:

```
{
  "sdk": {
    "version": "3.0.100"
  }
}
```

Update the project file

Update the Target Framework

ASP.NET Core 3.0 and later only run on .NET Core. Set the Target Framework Moniker (TFM) to `netcoreapp3.0`:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Remove obsolete package references

A large number of NuGet packages aren't produced for ASP.NET Core 3.0. Such package references should be removed from your project file. Consider the following project file for an ASP.NET Core 2.2 web app:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App"/>
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

The updated project file for ASP.NET Core 3.0:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

The updated ASP.NET Core 3.0 project file:

- In the `<PropertyGroup>`:
  - Updates the TFM to `netcoreapp3.0`
  - Removes the `<AspNetCoreHostingModel>` element. For more information, see In-process hosting model in this document.
- In the `<ItemGroup>`:
  - `Microsoft.AspNetCore.App` is removed. For more information, see Framework reference in this document.
  - `Microsoft.AspNetCore.Razor.Design` is removed and in the following list of packages no longer being produced.

To see the full list of packages that are no longer produced, select the following expand list:

▶ Click to expand the list of packages no longer being produced

Review breaking changes

Review breaking changes

Framework reference

Features of ASP.NET Core that were available through one of the packages listed above are available as part of the `Microsoft.AspNetCore.App` shared framework. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and includes a runtime component and a targeting pack. For more information, see The shared framework.

- Projects that target the `Microsoft.NET.Sdk.Web` SDK implicitly reference the `Microsoft.AspNetCore.App` framework.

  No additional references are required for these projects:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>
    ...
</Project>
```

- Projects that target `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Razor` SDK, should add an explicit `FrameworkReference` to `Microsoft.AspNetCore.App`:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
    ...
</Project>
```

Framework-dependent builds using Docker

Framework-dependent builds of console apps that use a package that depends on the ASP.NET Core shared framework may give the following runtime error:

```
It was not possible to find any compatible framework version
The specified framework 'Microsoft.AspNetCore.App', version '3.0.0' was not found.
  - No frameworks were found.
```

`Microsoft.AspNetCore.App` is the shared framework containing the ASP.NET Core runtime and is only present on the dotnet/core/aspnet Docker image. The 3.0 SDK reduces the size of framework-dependent builds using ASP.NET Core by not including duplicate copies of libraries that are available in the shared framework. This is a potential savings of up to 18 MB, but it requires that the ASP.NET Core runtime be present / installed to run the app.

To determine if the app has a dependency (either direct or indirect) on the ASP.NET Core shared framework, examine the *runtimeconfig.json* file generated during a build/publish of your app. The following JSON file shows a dependency on the ASP.NET Core shared framework:

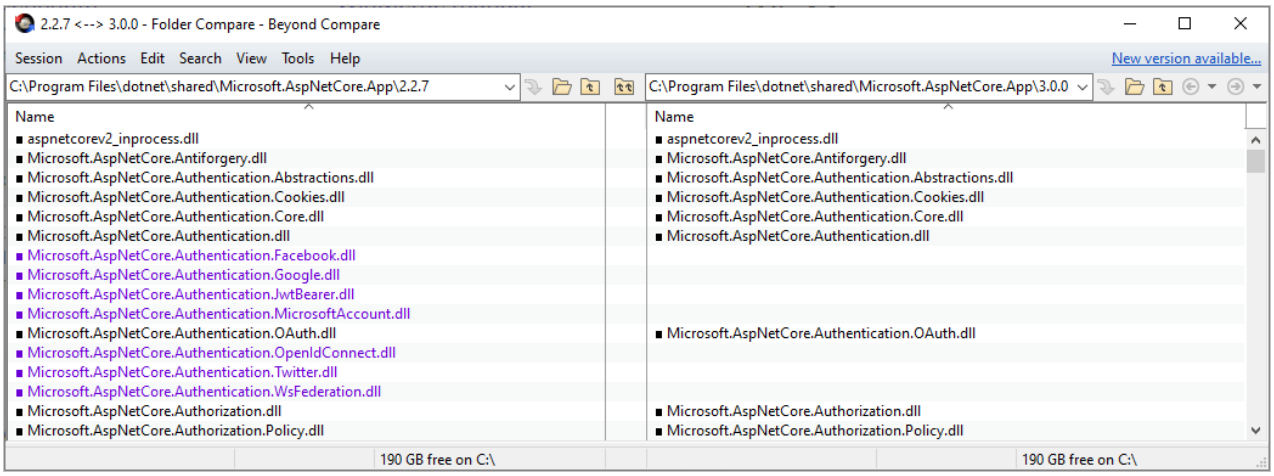
```
{
  "runtimeOptions": {
    "tfm": "netcoreapp3.0",
    "framework": {
      "name": "Microsoft.AspNetCore.App",
      "version": "3.0.0"
    },
    "configProperties": {
      "System.GC.Server": true
    }
  }
}
```

If your app is using Docker, use a base image that includes ASP.NET Core 3.0. For example, `docker pull mcr.microsoft.com/dotnet/core/aspnet:3.0`.

Add package references for removed assemblies

ASP.NET Core 3.0 removes some assemblies that were previously part of the `Microsoft.AspNetCore.App` package reference. To visualize which assemblies were removed, compare the two shared framework folders. For example,

a comparison of versions 2.2.7 and 3.0.0:



To continue using features provided by the removed assemblies, reference the 3.0 versions of the corresponding packages:

- A template-generated web app with **Individual User Accounts** requires adding the following packages:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UserSecretsId>My-secret</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="3.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="3.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.UI" Version="3.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="3.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="3.0.0" />
  </ItemGroup>

</Project>
```

- Microsoft.EntityFrameworkCore

  For more information on referencing the database provider-specific package, see Database Providers.

- Identity UI

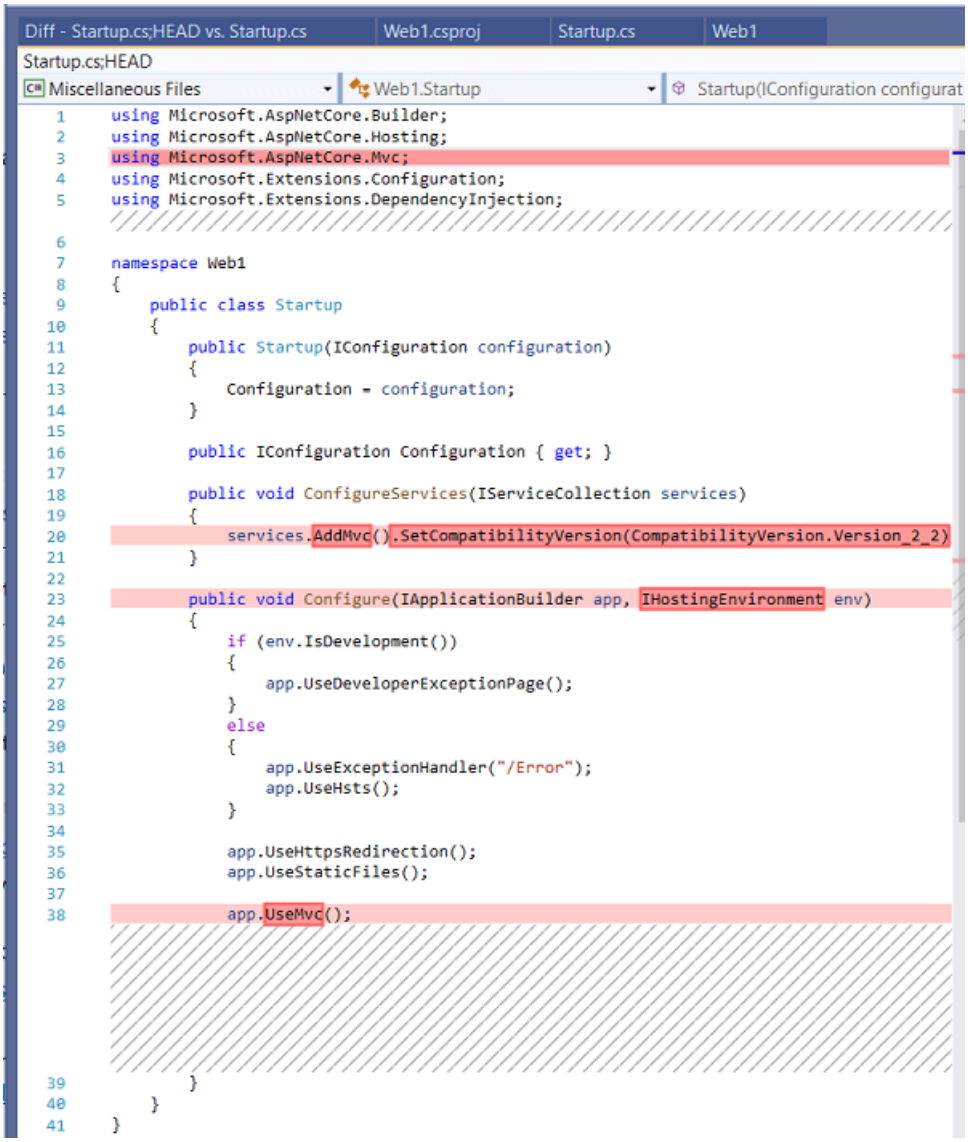
  Support for Identity UI can be added by referencing the Microsoft.AspNetCore.Identity.UI package.

- SPA Services
  - Microsoft.AspNetCore.SpaServices
  - Microsoft.AspNetCore.SpaServices.Extensions
- Authentication: Support for third-party authentication flows are available as NuGet packages:
  - Facebook OAuth (Microsoft.AspNetCore.Authentication.Facebook)
  - Google OAuth (Microsoft.AspNetCore.Authentication.Google)
  - Microsoft Account authentication (Microsoft.AspNetCore.Authentication.MicrosoftAccount)
  - OpenID Connect authentication (Microsoft.AspNetCore.Authentication.OpenIdConnect)
  - OpenID Connect bearer token (Microsoft.AspNetCore.Authentication.JwtBearer)
  - Twitter OAuth (Microsoft.AspNetCore.Authentication.Twitter)
  - WsFederation authentication (Microsoft.AspNetCore.Authentication.WsFederation)

- Formatting and content negotiation support for `System.Net.HttpClient`: The Microsoft.AspNet.WebApi.Client NuGet package provides useful extensibility to `System.Net.HttpClient` with APIs such as `ReadAsAsync` and `PostJsonAsync`.
- Razor runtime compilation: Support for runtime compilation of Razor views and pages is now part of Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation.
- MVC `Newtonsoft.Json` (Json.NET) support: Support for using MVC with `Newtonsoft.Json` is now part of `Microsoft.AspNetCore.Mvc.NewtonsoftJson`.

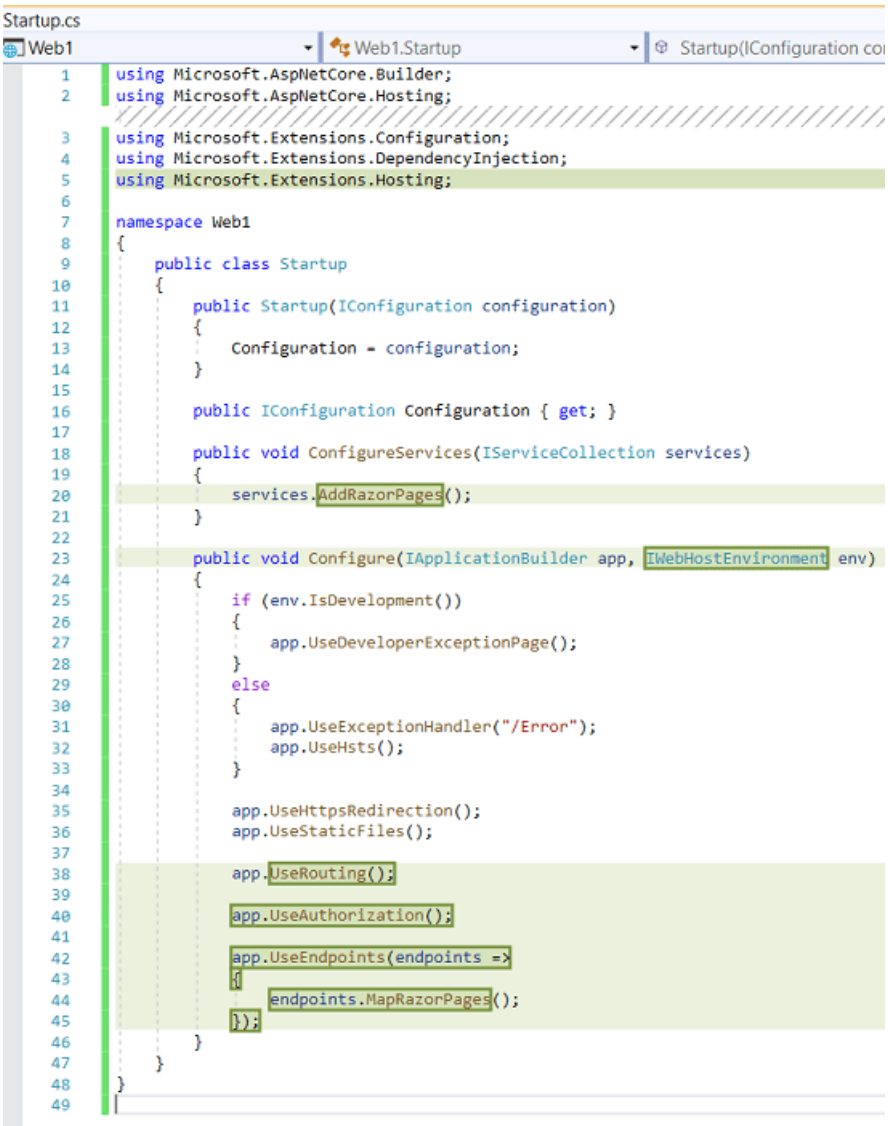
Startup changes

The following image shows the deleted and changed lines in an ASP.NET Core 2.2 Razor Pages Web app:



In the preceding image, deleted code is shown in red. The deleted code doesn't show cookie options code, which was deleted prior to comparing the files.

The following image shows the added and changed lines in an ASP.NET Core 3.0 Razor Pages Web app:



In the preceding image, added code is shown in green. For information on the following changes:

- `services.AddMvc` to `services.AddRazorPages`, see MVC service registration in this document.
- `CompatibilityVersion`, see Compatibility version for ASP.NET Core MVC.
- `IHostingEnvironment` to `IWebHostEnvironment`, see this GitHub announcement.
- `app.UseAuthorization` was added to the templates to show the order authorization middleware must be added. If the app doesn't use authorization, you can safely remove the call to `app.UseAuthorization`.
- `app.UseEndpoints`, see Razor Pages or Migrate Startup.Configure in this document.

Analyzer support

Projects that target `Microsoft.NET.Sdk.Web` implicitly reference analyzers previously shipped as part of the Microsoft.AspNetCore.Mvc.Analyzers package. No additional references are required to enable these.

If your app uses API analyzers previously shipped using the Microsoft.AspNetCore.Mvc.Api.Analyzers package, edit your project file to reference the analyzers shipped as part of the .NET Core Web SDK:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
    <PropertyGroup>
        <TargetFramework>netcoreapp3.0</TargetFramework>
        <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
    </PropertyGroup>

    ...
</Project>
```

Razor Class Library

Razor Class Library projects that provide UI components for MVC must set the `AddRazorSupportForMvc` property in the project file:

```
<PropertyGroup>
  <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
</PropertyGroup>
```

In-process hosting model

Projects default to the in-process hosting model in ASP.NET Core 3.0 or later. You may optionally remove the `<AspNetCoreHostingModel>` property in the project file if its value is `InProcess`.

Kestrel

Configuration

Migrate Kestrel configuration to the web host builder provided by `ConfigureWebHostDefaults` (*Program.cs*):

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(serverOptions =>
            {
                // Set properties and call methods on options
            })
            .UseStartup<Startup>();
        });
```

If the app creates the host manually with `HostBuilder`, call `UseKestrel` on the web host builder in `ConfigureWebHostDefaults`:

```
public static void Main(string[] args)
{
    var host = new HostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseKestrel(serverOptions =>
            {
                // Set properties and call methods on options
            })
            .UseIISIntegration()
            .UseStartup<Startup>();
        })
        .Build();

    host.Run();
}
```

Connection Middleware replaces Connection Adapters

Connection Adapters (`Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal.IConnectionAdapter`) have been removed from Kestrel. Replace Connection Adapters with Connection Middleware. Connection Middleware is similar to HTTP Middleware in the ASP.NET Core pipeline but for lower-level connections. HTTPS and connection logging:

- Have been moved from Connection Adapters to Connection Middleware.
- These extension methods work as in previous versions of ASP.NET Core.

For more information, see the TlsFilterConnectionHandler example in the ListenOptions.Protocols section of the Kestrel article.

Transport abstractions moved and made public

The Kestrel transport layer has been exposed as a public interface in `Connections.Abstractions`. As part of these updates:

- `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions` and associated types have been removed.
- NoDelay was moved from ListenOptions to the transport options.
- `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions.Internal.SchedulingMode` was removed from KestrelServerOptions.

For more information, see the following GitHub resources:

- Client/server networking abstractions (dotnet/AspNetCore #10308)
- Implement new bedrock listener abstraction and re-plat Kestrel on top (dotnet/AspNetCore #10321)

Kestrel Request trailer headers

For apps that target earlier versions of ASP.NET Core:

- Kestrel adds HTTP/1.1 chunked trailer headers into the request headers collection.
- Trailers are available after the request body is read to the end.

This causes some concerns about ambiguity between headers and trailers, so the trailers have been moved to a new collection (`RequestTrailerExtensions`) in 3.0.

HTTP/2 request trailers are:

- Not available in ASP.NET Core 2.2.
- Available in 3.0 as `RequestTrailerExtensions`.

New request extension methods are present to access these trailers. As with HTTP/1.1, trailers are available after the request body is read to the end.

For the 3.0 release, the following `RequestTrailerExtensions` methods are available:

- `GetDeclaredTrailers`: Gets the request `Trailer` header that lists which trailers to expect after the body.
- `SupportsTrailers`: Indicates if the request supports receiving trailer headers.
- `CheckTrailersAvailable`: Checks if the request supports trailers and if they're available to be read. This check doesn't assume that there are trailers to read. There might be no trailers to read even if `true` is returned by this method.
- `GetTrailer`: Gets the requested trailing header from the response. Check `SupportsTrailers` before calling `GetTrailer`, or a NotSupportedException may occur if the request doesn't support trailing headers.

For more information, see Put request trailers in a separate collection (dotnet/AspNetCore #10410).

AllowSynchronousIO disabled

`AllowSynchronousIO` enables or disables synchronous I/O APIs, such as `HttpRequest.Body.Read`, `HttpResponse.Body.Write`, and `Stream.Flush`. These APIs are a source of thread starvation leading to app crashes. In 3.0, `AllowSynchronousIO` is disabled by default. For more information, see the Synchronous I/O section in the Kestrel article.

If synchronous I/O is needed, it can be enabled by configuring the `AllowSynchronousIO` option on the server being used (when calling `ConfigureKestrel`, for example, if using Kestrel). Note that servers (Kestrel, HttpSys, TestServer, etc.) all have their own `AllowSynchronousIO` option that won't affect other servers. Synchronous I/O can be enabled

for all servers on a per-request basis using the `IHttpBodyControlFeature.AllowSynchronousIO` option:

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();

if (syncIOFeature != null)
{
    syncIOFeature.AllowSynchronousIO = true;
}
```

If you have trouble with TextWriter implementations or other streams that call synchronous APIs in Dispose, call the new DisposeAsync API instead.

For more information, see [Announcement] AllowSynchronousIO disabled in all servers (dotnet/AspNetCore #7644).

Output formatter buffering

Newtonsoft.Json, XmlSerializer, and DataContractSerializer based output formatters only support synchronous serialization. To allow these formatters to work with the AllowSynchronousIO restrictions of the server, MVC buffers the output of these formatters before writing to disk. As a result of buffering, MVC will include the Content-Length header when responding using these formatters.

System.Text.Json supports asynchronous serialization and consequently the `System.Text.Json` based formatter does not buffer. Consider using this formatter for improved performance.

To disable buffering, applications can configure SuppressOutputFormatterBuffering in their startup:

```
services.AddControllers(options => options.SuppressOutputFormatterBuffering = true)
```

Note that this may result in the application throwing a runtime exception if `AllowSynchronousIO` isn't also configured.

Microsoft.AspNetCore.Server.Kestrel.Https assembly removed

In ASP.NET Core 2.1, the contents of *Microsoft.AspNetCore.Server.Kestrel.Https.dll* were moved to *Microsoft.AspNetCore.Server.Kestrel.Core.dll*. This was a non-breaking update using `TypeForwardedTo` attributes. For 3.0, the empty *Microsoft.AspNetCore.Server.Kestrel.Https.dll* assembly and the NuGet package have been removed.

Libraries referencing Microsoft.AspNetCore.Server.Kestrel.Https should update ASP.NET Core dependencies to 2.1 or later.

Apps and libraries targeting ASP.NET Core 2.1 or later should remove any direct references to the Microsoft.AspNetCore.Server.Kestrel.Https package.

Newtonsoft.Json (Json.NET) support

As part of the work to improve the ASP.NET Core shared framework, Newtonsoft.Json (Json.NET) has been removed from the ASP.NET Core shared framework.

The default JSON serializer for ASP.NET Core is now System.Text.Json, which is new in .NET Core 3.0. Consider using `System.Text.Json` when possible. It's high-performance and doesn't require an additional library dependency. However, since `System.Text.Json` is new, it might currently be missing features that your app needs. For more information, see How to migrate from Newtonsoft.Json to System.Text.Json.

Use Newtonsoft.Json in an ASP.NET Core 3.0 SignalR project

- Install the Microsoft.AspNetCore.SignalR.Protocols.NewtonsoftJson NuGet package.

- On the client, chain an `AddNewtonsoftJsonProtocol` method call to the `HubConnectionBuilder` instance:

```
new HubConnectionBuilder()
    .WithUrl("/chathub")
    .AddNewtonsoftJsonProtocol(...)
    .Build();
```

- On the server, chain an `AddNewtonsoftJsonProtocol` method call to the `AddSignalR` method call in `Startup.ConfigureServices`:

```
services.AddSignalR()
    .AddNewtonsoftJsonProtocol(...);
```

Use Newtonsoft.Json in an ASP.NET Core 3.0 MVC project

- Install the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` package.
- Update `Startup.ConfigureServices` to call `AddNewtonsoftJson`.

```
services.AddMvc()
    .AddNewtonsoftJson();
```

`AddNewtonsoftJson` is compatible with the new MVC service registration methods:

  - `AddRazorPages`
  - `AddControllersWithViews`
  - `AddControllers`

```
services.AddControllers()
    .AddNewtonsoftJson();
```

`Newtonsoft.Json` settings can be set in the call to `AddNewtonsoftJson`:

```
services.AddMvc()
    .AddNewtonsoftJson(options =>
           options.SerializerSettings.ContractResolver =
              new CamelCasePropertyNamesContractResolver());
```

Note: If the `AddNewtonsoftJson` method isn't available, make sure that you installed the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` package. A common error is to install the Newtonsoft.Json package instead of the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` package.

MVC service registration

ASP.NET Core 3.0 adds new options for registering MVC scenarios inside `Startup.ConfigureServices`.

Three new top-level extension methods related to MVC scenarios on `IServiceCollection` are available. Templates use these new methods instead of `AddMvc`. However, `AddMvc` continues to behave as it has in previous releases.

The following example adds support for controllers and API-related features, but not views or pages. The API template uses this code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
}
```

The following example adds support for controllers, API-related features, and views, but not pages. The Web Application (MVC) template uses this code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
}
```

The following example adds support for Razor Pages and minimal controller support. The Web Application template uses this code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
}
```

The new methods can also be combined. The following example is equivalent to calling `AddMvc` in ASP.NET Core 2.2:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

Routing startup code

If an app calls `UseMvc` or `UseSignalR`, migrate the app to Endpoint Routing if possible. To improve Endpoint Routing compatibility with previous versions of MVC, we've reverted some of the changes in URL generation introduced in ASP.NET Core 2.2. If you experienced problems using Endpoint Routing in 2.2, expect improvements in ASP.NET Core 3.0 with the following exceptions:

- If the app implements `IRouter` or inherits from `Route`, use DynamicRouteValuesTransformer as the replacement.
- If the app directly accesses `RouteData.Routers` inside MVC to parse URLs, you can replace this with use of LinkParser.ParsePathByEndpointName.
  - Define the route with a route name.
  - Use `LinkParser.ParsePathByEndpointName` and pass in the desired route name.

Endpoint Routing supports the same route pattern syntax and route pattern authoring features as `IRouter`. Endpoint Routing supports `IRouteConstraint`. Endpoint routing supports `[Route]`, `[HttpGet]`, and the other MVC routing attributes.

For most applications, only `Startup` requires changes.

Migrate Startup.Configure

General advice:

- Add `UseRouting`.

- If the app calls `UseStaticFiles`, place `UseStaticFiles` **before** `UseRouting`.
- If the app uses authentication/authorization features such as `AuthorizePage` or `[Authorize]`, place the call to `UseAuthentication` and `UseAuthorization`: **after**, `UseRouting` and `UseCors`, but before `UseEndpoints`:

```
public void Configure(IApplicationBuilder app)
{
  ...

  app.UseStaticFiles();

  app.UseRouting();
  app.UseCors();

  app.UseAuthentication();
  app.UseAuthorization();

  app.UseEndpoints(endpoints => {
     endpoints.MapControllers();
  });
```

- Replace `UseMvc` or `UseSignalR` with `UseEndpoints`.
- If the app uses CORS scenarios, such as `[EnableCors]`, place the call to `UseCors` before any other middleware that use CORS (for example, place `UseCors` before `UseAuthentication`, `UseAuthorization`, and `UseEndpoints`).
- Replace `IHostingEnvironment` with `IWebHostEnvironment` and add a `using` statement for the Microsoft.Extensions.Hosting namespace.
- Replace `IApplicationLifetime` with IHostApplicationLifetime (Microsoft.Extensions.Hosting namespace).
- Replace `EnvironmentName` with Environments (Microsoft.Extensions.Hosting namespace).

The following code is an example of `Startup.Configure` in a typical ASP.NET Core 2.2 app:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    app.UseAuthentication();

    app.UseSignalR(hubs =>
    {
        hubs.MapHub<ChatHub>("/chat");
    });

    app.UseMvc(routes =>
    {
        routes.MapRoute("default", "{controller=Home}/{action=Index}/{id?}");
    });
}
```

After updating the previous `Startup.Configure` code:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    app.UseRouting();

    app.UseCors();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chat");
        endpoints.MapControllerRoute("default", "{controller=Home}/{action=Index}/{id?}");
    });
}
```

WARNING

For most apps, calls to `UseAuthentication`, `UseAuthorization`, and `UseCors` must appear between the calls to `UseRouting` and `UseEndpoints` to be effective.

Health Checks

Health Checks use endpoint routing with the Generic Host. In `Startup.Configure`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
});
```

Health Checks endpoints can:

- Specify one or more permitted hosts/ports.
- Require authorization.
- Require CORS.

For more information, see Health checks in ASP.NET Core.

Security middleware guidance

Support for authorization and CORS is unified around the middleware approach. This allows use of the same middleware and functionality across these scenarios. An updated authorization middleware is provided in this release, and CORS Middleware is enhanced so that it can understand the attributes used by MVC controllers.

CORS

Previously, CORS could be difficult to configure. Middleware was provided for use in some use cases, but MVC filters were intended to be used **without** the middleware in other use cases. With ASP.NET Core 3.0, we recommend that all apps that require CORS use the CORS Middleware in tandem with Endpoint Routing. `UseCors` can be provided with a default policy, and `[EnableCors]` and `[DisableCors]` attributes can be used to override the default policy where required.

In the following example:

- CORS is enabled for all endpoints with the `default` named policy.

- The `MyController` class disables CORS with the `[DisableCors]` attribute.

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseCors("default");

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

[DisableCors]
public class MyController : ControllerBase
{
    ...
}
```

Authorization

In earlier versions of ASP.NET Core, authorization support was provided via the `[Authorize]` attribute. Authorization middleware wasn't available. In ASP.NET Core 3.0, authorization middleware is required. We recommend placing the ASP.NET Core Authorization Middleware (`UseAuthorization`) immediately after `UseAuthentication` . The Authorization Middleware can also be configured with a default policy, which can be overridden.

In ASP.NET Core 3.0 or later, `UseAuthorization` is called in `Startup.Configure` , and the following `HomeController` requires a signed in user:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

public class HomeController : Controller
{
    [Authorize]
    public IActionResult BuyWidgets()
    {
        ...
    }
}
```

When using endpoint routing, we recommend against configuring `<xref:Microsoft.AspNetCore.Mvc.Authorization.AuthorizeFilter>` and instead relying on the Authorization middleware. If the app uses an `AuthorizeFilter` as a global filter in MVC, we recommend refactoring the code to provide a policy in the call to `AddAuthorization` .

The `DefaultPolicy` is initially configured to require authentication, so no additional configuration is required. In the following example, MVC endpoints are marked as `RequireAuthorization` so that all requests must be authorized based on the `DefaultPolicy`. However, the `HomeController` allows access without the user signing into the app due to `[AllowAnonymous]`:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute().RequireAuthorization();
    });
}

[AllowAnonymous]
public class HomeController : Controller
{
    ...
}
```

Authorization for specific endpoints

Authorization can also be configured for specific classes of endpoints. The following code is an example of converting an MVC app that configured a global `AuthorizeFilter` to an app with a specific policy requiring authorization:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    static readonly string _RequireAuthenticatedUserPolicy =
                           "RequireAuthenticatedUserPolicy";
    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(
                 options => options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();

        // Pre 3.0:
        // services.AddMvc(options => options.Filters.Add(new AuthorizeFilter(...));

        services.AddControllersWithViews();
        services.AddRazorPages();
        services.AddAuthorization(o => o.AddPolicy(_RequireAuthenticatedUserPolicy,
                        builder => builder.RequireAuthenticatedUser()));

    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute()
                .RequireAuthorization(_RequireAuthenticatedUserPolicy);
            endpoints.MapRazorPages();
        });
    }
}
```

Policies can also be customized. The `DefaultPolicy` is configured to require authentication:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(
                 options => options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();

        services.AddControllersWithViews();
        services.AddRazorPages();
        services.AddAuthorization(options =>
        {
            options.DefaultPolicy = new AuthorizationPolicyBuilder()
              .RequireAuthenticatedUser()
              .Build();
        });

    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute().RequireAuthorization();
            endpoints.MapRazorPages();
        });
    }
}
```

```
[AllowAnonymous]
public class HomeController : Controller
{
```

Alternatively, all endpoints can be configured to require authorization without `[Authorize]` or `RequireAuthorization` by configuring a `FallbackPolicy`. The `FallbackPolicy` is different from the `DefaultPolicy`. The `DefaultPolicy` is triggered by `[Authorize]` or `RequireAuthorization`, while the `FallbackPolicy` is triggered

when no other policy is set. `FallbackPolicy` is initially configured to allow requests without authorization.

The following example is the same as the preceding `DefaultPolicy` example but uses the `FallbackPolicy` to always require authentication on all endpoints except when `[AllowAnonymous]` is specified:

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddAuthorization(options =>
    {
        options.FallbackPolicy = new AuthorizationPolicyBuilder()
          .RequireAuthenticatedUser()
          .Build();
    });
}

public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

[AllowAnonymous]
public class HomeController : Controller
{
    ...
}
```

Authorization by middleware works without the framework having any specific knowledge of authorization. For instance, health checks has no specific knowledge of authorization, but health checks can have a configurable authorization policy applied by the middleware.

Additionally, each endpoint can customize its authorization requirements. In the following example, `UseAuthorization` processes authorization with the `DefaultPolicy`, but the `/healthz` health check endpoint requires an `admin` user:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints
            .MapHealthChecks("/healthz")
            .RequireAuthorization(new AuthorizeAttribute(){ Roles = "admin", });
    });
}
```

Protection is implemented for some scenarios. Endpoints Middleware throws an exception if an authorization or CORS policy is skipped due to missing middleware. Analyzer support to provide additional feedback about misconfiguration is in progress.

Custom authorization handlers

If the app uses custom authorization handlers, endpoint routing passes a different resource type to handlers than MVC. Handlers that expect the authorization handler context resource to be of type AuthorizationFilterContext (the resource type provided by MVC filters) will need to be updated to handle resources of type RouteEndpoint (the resource type given to authorization handlers by endpoint routing).

MVC still uses `AuthorizationFilterContext` resources, so if the app uses MVC authorization filters along with endpoint routing authorization, it may be necessary to handle both types of resources.

SignalR

Mapping of SignalR hubs now takes place inside `UseEndpoints`.

Map each hub with `MapHub`. As in previous versions, each hub is explicitly listed.

In the following example, support for the `ChatHub` SignalR hub is added:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>();
    });
}
```

There is a new option for controlling message size limits from clients. For example, in `Startup.ConfigureServices`:

```
services.AddSignalR(hubOptions =>
{
    hubOptions.MaximumReceiveMessageSize = 32768;
});
```

In ASP.NET Core 2.2, you could set the `TransportMaxBufferSize` and that would effectively control the maximum message size. In ASP.NET Core 3.0, that option now only controls the maximum size before backpressure is observed.

MVC controllers

Mapping of controllers now takes place inside `UseEndpoints`.

Add `MapControllers` if the app uses attribute routing. Since routing includes support for many frameworks in ASP.NET Core 3.0 or later, adding attribute-routed controllers is opt-in.

Replace the following:

- `MapRoute` with `MapControllerRoute`
- `MapAreaRoute` with `MapAreaControllerRoute`

Since routing now includes support for more than just MVC, the terminology has changed to make these methods clearly state what they do. Conventional routes such as `MapControllerRoute` / `MapAreaControllerRoute` / `MapDefaultControllerRoute` are applied in the order that they're added. Place more specific routes (such as routes for an area) first.

In the following example:

- `MapControllers` adds support for attribute-routed controllers.
- `MapAreaControllerRoute` adds a conventional route for controllers in an area.
- `MapControllerRoute` adds a conventional route for controllers.

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapAreaControllerRoute(
            "admin",
            "admin",
            "Admin/{controller=Home}/{action=Index}/{id?}");
        endpoints.MapControllerRoute(
            "default", "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Async suffix removal from controller action names

In ASP.NET Core 3.0, ASP.NET Core MVC removes the `Async` suffix from controller action names. Both routing and link generation are impacted by this new default. For example:

```
public class ProductsController : Controller
{
    public async Task<IActionResult> ListAsync()
    {
        var model = await _dbContext.Products.ToListAsync();
        return View(model);
    }
}
```

Prior to ASP.NET Core 3.0:

- The preceding action could be accessed at the *Products/ListAsync* route.
- Link generation required specifying the `Async` suffix. For example:

  ```
  <a asp-controller="Products" asp-action="ListAsync">List</a>
  ```

In ASP.NET Core 3.0:

- The preceding action can be accessed at the *Products/List* route.
- Link generation doesn't require specifying the `Async` suffix. For example:

  ```
  <a asp-controller="Products" asp-action="List">List</a>
  ```

This change doesn't affect names specified using the `[ActionName]` attribute. The default behavior can be disabled with the following code in `Startup.ConfigureServices`:

```
services.AddMvc(options =>
    options.SuppressAsyncSuffixInActionNames = false);
```

Changes to link generation

As explained in documentation on differences from earlier versions of routing, there are some differences in link generation (using `Url.Link` and similar APIs, for example). These include:

- By default, when using endpoint routing, casing of route parameters in generated URIs is not necessarily preserved. This behavior can be controlled with the `IOutboundParameterTransformer` interface.
- Generating a URI for an invalid route (a controller/action or page that doesn't exist) will produce an empty string under endpoint routing instead of producing an invalid URI.
- Ambient values (route parameters from the current context) are not automatically used in link generation with endpoint routing. Previously, when generating a link to another action (or page), unspecified route values would be inferred from the *current* routes ambient values. When using endpoint routing, all route parameters must be specified explicitly during link generation.

Razor Pages

Mapping Razor Pages now takes place inside `UseEndpoints`.

Add `MapRazorPages` if the app uses Razor Pages. Since Endpoint Routing includes support for many frameworks, adding Razor Pages is now opt-in.

In the following `Startup.Configure` method, `MapRazorPages` adds support for Razor Pages:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Use MVC without Endpoint Routing

Using MVC via `UseMvc` or `UseMvcWithDefaultRoute` in ASP.NET Core 3.0 requires an explicit opt-in inside `Startup.ConfigureServices`. This is required because MVC must know whether it can rely on the authorization and CORS Middleware during initialization. An analyzer is provided that warns if the app attempts to use an unsupported configuration.

If the app requires legacy `IRouter` support, disable `EnableEndpointRouting` using any of the following approaches in `Startup.ConfigureServices`:

```
services.AddMvc(options => options.EnableEndpointRouting = false);
```

```
services.AddControllers(options => options.EnableEndpointRouting = false);
```

```
services.AddControllersWithViews(options => options.EnableEndpointRouting = false);
```

```
services.AddRazorPages().AddMvcOptions(options => options.EnableEndpointRouting = false);
```

Health checks

Health checks can be used as a *router-ware* with Endpoint Routing.

Add `MapHealthChecks` to use health checks with Endpoint Routing. The `MapHealthChecks` method accepts arguments similar to `UseHealthChecks`. The advantage of using `MapHealthChecks` over `UseHealthChecks` is the ability to apply authorization and to have greater fine-grained control over the matching policy.

In the following example, `MapHealthChecks` is called for a health check endpoint at `/healthz`:

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHealthChecks("/healthz", new HealthCheckOptions() { });
    });
}
```

HostBuilder replaces WebHostBuilder

The ASP.NET Core 3.0 templates use Generic Host. Previous versions used Web Host. The following code shows the ASP.NET Core 3.0 template generated `Program` class:

```
// requires using Microsoft.AspNetCore.Hosting;
// requires using Microsoft.Extensions.Hosting;

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code shows the ASP.NET Core 2.2 template-generated `Program` class:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

IWebHostBuilder remains in 3.0 and is the type of the `webBuilder` seen in the preceding code sample. WebHostBuilder will be deprecated in a future release and replaced by `HostBuilder`.

The most significant change from `WebHostBuilder` to `HostBuilder` is in dependency injection (DI). When using `HostBuilder`, you can only inject the following into `Startup`'s constructor:

- IConfiguration
- IHostEnvironment
- IWebHostEnvironment

The `HostBuilder` DI constraints:

- Enable the DI container to be built only one time.
- Avoids the resulting object lifetime issues like resolving multiple instances of singletons.

For more information, see Avoiding Startup service injection in ASP.NET Core 3.

AddAuthorization moved to a different assembly

The ASP.NET Core 2.2 and lower `AddAuthorization` methods in *Microsoft.AspNetCore.Authorization.dll*:

- Have been renamed `AddAuthorizationCore`.
- Have been moved to *Microsoft.AspNetCore.Authorization.Policy.dll*.

Apps that are using both *Microsoft.AspNetCore.Authorization.dll* and *Microsoft.AspNetCore.Authorization.Policy.dll* aren't impacted.

Apps that are not using *Microsoft.AspNetCore.Authorization.Policy.dll* should do one of the following:

- Add a reference to *Microsoft.AspNetCore.Authorization.Policy.dll*. This approach works for most apps and is all that is required.
- Switch to using `AddAuthorizationCore`

For more information, see Breaking change in `AddAuthorization(o =>`) overload lives in a different assembly #386.

Identity UI

Identity UI updates for ASP.NET Core 3.0:

- Add a package reference to Microsoft.AspNetCore.Identity.UI.
- Apps that don't use Razor Pages must call `MapRazorPages`. See Razor Pages in this document.
- Bootstrap 4 is the default UI framework. Set an `IdentityUIFrameworkVersion` project property to change the default. For more information, see this GitHub announcement.

SignalR

The SignalR JavaScript client has changed from `@aspnet/signalr` to `@microsoft/signalr`. To react to this change, change the references in *package.json* files, `require` statements, and ECMAScript `import` statements.

System.Text.Json is the default protocol

`System.Text.Json` is now the default Hub protocol used by both the client and server.

In `Startup.ConfigureServices`, call `AddJsonProtocol` to set serializer options.

Server:

```
services.AddSignalR(...)
        .AddJsonProtocol(options =>
        {
            options.PayloadSerializerOptions.WriteIndented = false;
        })
```

Client:

```
new HubConnectionBuilder()
    .WithUrl("/chathub")
    .AddJsonProtocol(options =>
    {
        options.PayloadSerializerOptions.WriteIndented = false;
    })
    .Build();
```

Switch to Newtonsoft.Json

If you're using features of Newtonsoft.Json that aren't supported in System.Text.Json, you can switch back to `Newtonsoft.Json`. See Use Newtonsoft.Json in an ASP.NET Core 3.0 SignalR project earlier in this article.

Redis distributed caches

The Microsoft.Extensions.Caching.Redis package isn't available for ASP.NET Core 3.0 or later apps. Replace the package reference with Microsoft.Extensions.Caching.StackExchangeRedis. For more information, see Distributed caching in ASP.NET Core.

Opt in to runtime compilation

Prior to ASP.NET Core 3.0, runtime compilation of views was an implicit feature of the framework. Runtime compilation supplements build-time compilation of views. It allows the framework to compile Razor views and pages (*.cshtml* files) when the files are modified, without having to rebuild the entire app. This feature supports the scenario of making a quick edit in the IDE and refreshing the browser to view the changes.

In ASP.NET Core 3.0, runtime compilation is an opt-in scenario. Build-time compilation is the only mechanism for view compilation that's enabled by default. The runtime relies on Visual Studio or dotnet-watch in Visual Studio Code to rebuild the project when it detects changes to *.cshtml* files. In Visual Studio, changes to *.cs*, *.cshtml*, or *.razor* files in the project being run (`Ctrl+F5`), but not debugged (`F5`), trigger recompilation of the project.

To enable runtime compilation in your ASP.NET Core 3.0 project:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.
2. Update `Startup.ConfigureServices` to call `AddRazorRuntimeCompilation`:

   For ASP.NET Core MVC, use the following code:

```
services.AddControllersWithViews()
    .AddRazorRuntimeCompilation(...);
```

For ASP.NET Core Razor Pages, use the following code:

```
services.AddRazorPages()
    .AddRazorRuntimeCompilation(...);
```

The sample at https://github.com/aspnet/samples/tree/master/samples/aspnetcore/mvc/runtimecompilation shows an example of enabling runtime compilation conditionally in Development environments.

For more information on Razor file compilation, see Razor file compilation in ASP.NET Core.

Migrate libraries via multi-targeting

Libraries often need to support multiple versions of ASP.NET Core. Most libraries that were compiled against previous versions of ASP.NET Core should continue working without issues. The following conditions require the app to be cross-compiled:

- The library relies on a feature that has a binary breaking change.
- The library wants to take advantage of new features in ASP.NET Core 3.0.

For example:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netcoreapp3.0;netstandard2.0</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.0'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netstandard2.0'">
    <PackageReference Include="Microsoft.AspNetCore" Version="2.1.0" />
  </ItemGroup>
</Project>
```

Use `#ifdefs` to enable ASP.NET Core 3.0-specific APIs:

```
var webRootFileProvider =
#if NETCOREAPP3_0
    GetRequiredService<IWebHostEnvironment>().WebRootFileProvider;
#elif NETSTANDARD2_0
    GetRequiredService<IHostingEnvironment>().WebRootFileProvider;
#else
#error unknown target framework
#endif
```

For more information on using ASP.NET Core APIs in a class library, see Use ASP.NET Core APIs in a class library.

Miscellaneous changes

The validation system in .NET Core 3.0 and later treats non-nullable parameters or bound properties as if they had a `[Required]` attribute. For more information, see [Required] attribute.

Publish

Delete the *bin* and *obj* folders in the project directory.

TestServer

For apps that use TestServer directly with the Generic Host, create the `TestServer` on an IWebHostBuilder in ConfigureWebHost:

```
[Fact]
public async Task GenericCreateAndStartHost_GetTestServer()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .Configure(app => { });
        })
    .StartAsync();

    var response = await host.GetTestServer().CreateClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Breaking API changes

Review breaking changes:

- Complete list of breaking changes in the ASP.NET Core 3.0 release
- Breaking API changes in Antiforgery, CORS, Diagnostics, MVC, and Routing. This list includes breaking changes for compatibility switches.
- For a summary of 2.2-to-3.0 breaking changes across .NET Core, ASP.NET Core, and Entity Framework Core, see Breaking changes for migration from version 2.2 to 3.0.

Endpoint routing with catch-all parameter

WARNING

A **catch-all** parameter may match routes incorrectly due to a bug in routing. Apps impacted by this bug have the following characteristics:

- A catch-all route, for example, `{**slug}"`
- The catch-all route fails to match requests it should match.
- Removing other routes makes catch-all route start working.

See GitHub bugs 18677 and 16579 for example cases that hit this bug.

An opt-in fix for this bug is contained in .NET Core 3.1.301 SDK and later. The following code sets an internal switch that fixes this bug:

```
public static void Main(string[] args)
{
   AppContext.SetSwitch("Microsoft.AspNetCore.Routing.UseCorrectCatchAllBehavior",
                        true);
   CreateHostBuilder(args).Build().Run();
}
// Remaining code removed for brevity.
```

.NET Core 3.0 on Azure App Service

The rollout of .NET Core to Azure App Service is finished. .NET Core 3.0 is available in all Azure App Service datacenters.

Migrate from ASP.NET Core 2.1 to 2.2

9/22/2020 • 5 minutes to read • Edit Online

By Scott Addie

This article explains how to update an existing ASP.NET Core 2.1 project to ASP.NET Core 2.2.

Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

Update Target Framework Moniker (TFM)

Projects targeting .NET Core should use the TFM of a version greater than or equal to .NET Core 2.2. In the project file, update the `<TargetFramework>` node's inner text with `netcoreapp2.2`:

```
<TargetFramework>netcoreapp2.2</TargetFramework>
```

Projects targeting .NET Framework may continue to use the TFM of a version greater than or equal to .NET Framework 4.6.1:

```
<TargetFramework>net461</TargetFramework>
```

Adopt the IIS in-process hosting model

To adopt the in-process hosting model for IIS, add the `<AspNetCoreHostingModel>` property with a value of `InProcess` to a `<PropertyGroup>` in the project file:

```
<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
```

The in-process hosting model isn't supported for ASP.NET Core apps targeting .NET Framework.

For more information, see ASP.NET Core Module.

Update a custom web.config file

For projects that use a custom *web.config* file in the project root to generate their published *web.config* file:

- In the `<handlers>` entry that adds the ASP.NET Core Module (`name="aspNetCore"`), change the `modules` attribute value from `AspNetCoreModule` to `AspNetCoreModuleV2`.
- In the `<aspNetCore>` element, add the hosting model attribute (`hostingModel="InProcess"`).

For more information and example *web.config* files, see ASP.NET Core Module.

Update package references

If targeting .NET Core, remove the metapackage reference's `Version` attribute in the project file. Inclusion of a `Version` attribute results in the following warning:

```
A PackageReference to 'Microsoft.AspNetCore.App' specified a Version of `2.2.0`. Specifying the version of this
package is not recommended. For more information, see https://aka.ms/sdkimplicitrefs
```

For more information, see Microsoft.AspNetCore.App metapackage for ASP.NET Core.

The metapackage reference should resemble the following `<PackageReference />` node:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

If targeting .NET Framework, update each package reference's `Version` attribute to 2.2.0 or later. Here are the package references in a typical ASP.NET Core 2.2 project targeting .NET Framework:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.CookiePolicy" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.HttpsPolicy" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.2.0" />
</ItemGroup>
```

If referencing the Microsoft.AspNetCore.Razor.Design package, update its `Version` attribute to 2.2.0 or later. Failure to do so results in the following error:

```
Detected package downgrade: Microsoft.AspNetCore.Razor.Design from 2.2.0 to 2.1.2. Reference the package
directly from the project to select a different version.
```

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to the 2.2 version installed on your machine:

```
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

Update launch settings

If using Visual Studio Code, update the project's launch settings file (*.vscode/launch.json*). The `program` path should

reference the new TFM:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/bin/Debug/netcoreapp2.2/test-app.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "internalConsoleOptions": "openOnSessionStart",
            "launchBrowser": {
                "enabled": true,
                "args": "${auto-detect-url}",
                "windows": {
                    "command": "cmd.exe",
                    "args": "/C start ${auto-detect-url}"
                },
                "osx": {
                    "command": "open"
                },
                "linux": {
                    "command": "xdg-open"
                }
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "sourceFileMap": {
                "/Views": "${workspaceFolder}/Views"
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach",
            "processId": "${command:pickProcess}"
        }
    ]
}
```

Update Kestrel configuration

If the app calls UseKestrel by calling `CreateDefaultBuilder` in the CreateWebHostBuilder method of the `Program` class, call `ConfigureKestrel` to configure Kestrel server instead of `UseKestrel` in order to avoid conflicts with the IIS in-process hosting model:

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        });
```

If the app doesn't call `CreateDefaultBuilder` and builds the host manually in the `Program` class, call UseKestrel **before** calling `ConfigureKestrel`:

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseKestrel()
        .UseIISIntegration()
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        })
        .Build();

    host.Run();
}
```

For more information, see Kestrel web server implementation in ASP.NET Core.

Update compatibility version

Update the compatibility version in `Startup.ConfigureServices` to `Version_2_2`:

```
services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Update CORS policy

In ASP.NET Core 2.2, the CORS middleware responds with a wildcard origin (`*`) if a policy allows any origin and allows credentials. Credentials aren't supported when a wildcard origin (`*`) is specified, and browsers will disallow the CORS request. For more information, including options for correcting the problem on the client, see the MDN web docs.

To correct this problem on the server, take one of the following actions:

- Modify the CORS policy to no longer allow credentials. That is, remove the call to AllowCredentials when configuring the policy.
- If credentials are required for the CORS request to succeed, modify the policy to specify allowed hosts. For example, use `builder.WithOrigins("https://api.example1.com", "https://example2.com")` instead of using AllowAnyOrigin.

Update Docker images

The following table shows the Docker image tag changes:

2.1	2.2
`microsoft/dotnet:2.1-aspnetcore-runtime`	`mcr.microsoft.com/dotnet/core/aspnet:2.2`
`microsoft/dotnet:2.1-sdk`	`mcr.microsoft.com/dotnet/core/sdk:2.2`

Change the `FROM` lines in your *Dockerfile* to use the new image tags in the preceding table's 2.2 column.

Build manually in Visual Studio when using IIS in-process hosting

Visual Studio's **Auto build on browser request** experience doesn't function with the IIS in-process hosting

model. You must manually rebuild the project when using in-process hosting. Improvements to this experience are planned for a future release of Visual Studio.

Update logging code

Recommended logging configuration code didn't change from 2.1 to 2.2, but some 1.x coding patterns that still worked in 2.1 no longer work in 2.2.

If your app does logging provider initialization, filtering, and configuration loading in the `Startup` class, move that code to `Program.Main`:

- Provider initialization:

  1.x example:

  ```
  public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
  {
      loggerFactory.AddConsole();
  }
  ```

  2.2 example:

  ```
  public static void Main(string[] args)
  {
      var webHost = new WebHostBuilder()
          // ...
          .ConfigureLogging((hostingContext, logging) =>
          {
              logging.AddConsole();
          })
          // ...
  }
  ```

- Filtering:

  1.x example:

  ```
  public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
  {
      loggerFactory.AddConsole(LogLevel.Information);
      // or
      loggerFactory.AddConsole((category, level) =>
          category == "A" || level == LogLevel.Critical);
  }
  ```

  2.2 example:

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        // ...
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConsole()
                   .AddFilter<ConsoleLoggerProvider>
                       (category: null, level: LogLevel.Information)
                   // or
                   .AddFilter<ConsoleLoggerProvider>
                       ((category, level) => category == "A" ||
                           level == LogLevel.Critical)
            );
        })
        // ...
}
```

- Configuration loading:

  1.x example:

  ```
  public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)
  {
      loggerFactory.AddConsole(Configuration);
  }
  ```

  2.2 example:

  ```
  public static void Main(string[] args)
  {
      var webHost = new WebHostBuilder()
          // ...
          .ConfigureLogging((hostingContext, logging) =>
          {
              logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
              logging.AddConsole();
          })
          // ...
  }
  ```

For more information, see Logging in .NET Core and ASP.NET Core

Additional resources

- Compatibility version for ASP.NET Core MVC
- Microsoft.AspNetCore.App metapackage for ASP.NET Core
- Implicit package references

Migrate from ASP.NET Core 2.0 to 2.1

9/22/2020 • 10 minutes to read • Edit Online

By Rick Anderson

See What's new in ASP.NET Core 2.1 for an overview of the new features in ASP.NET Core 2.1.

This article:

- Covers the basics of migrating an ASP.NET Core 2.0 app to 2.1.
- Provides an overview of the changes to the ASP.NET Core web application templates.

A quick way to get an overview of the changes in 2.1 is to:

- Create an ASP.NET Core 2.0 web app named WebApp1.
- Commit the WebApp1 in a source control system.
- Delete WebApp1 and create an ASP.NET Core 2.1 web app named WebApp1 in the same place.
- Review the changes in the 2.1 version.

This article provides an overview on migration to ASP.NET Core 2.1. It doesn't contain a complete list of all changes needed to migrate to version 2.1. Some projects might require more steps depending on the options selected when the project was created and modifications made to the project.

Update the project file to use 2.1 versions

Update the project file:

- Change the target framework to .NET Core 2.1 by updating the project file to `<TargetFramework>netcoreapp2.1</TargetFramework>`.
- Replace the package reference for `Microsoft.AspNetCore.All` with a package reference for `Microsoft.AspNetCore.App`. You may need to add dependencies that were removed from `Microsoft.AspNetCore.All`. For more information, see Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0 and Microsoft.AspNetCore.App metapackage for ASP.NET Core.
- Remove the "Version" attribute on the package reference to `Microsoft.AspNetCore.App`. Projects that use `<Project Sdk="Microsoft.NET.Sdk.Web">` don't need to set the version. The version is implied by the target framework and selected to best match the way ASP.NET Core 2.1 works. For more information, see the Rules for projects targeting the shared framework section.
- For apps that target the .NET Framework, update each package reference to 2.1.
- Remove references to **<DotNetCliToolReference>** elements for the following packages. These tools are bundled by default in the .NET Core CLI and don't need to be installed separately.
  - Microsoft.DotNet.Watcher.Tools (`dotnet watch`)
  - Microsoft.EntityFrameworkCore.Tools.DotNet (`dotnet ef`)
  - Microsoft.Extensions.Caching.SqlConfig.Tools (`dotnet sql-cache`)
  - Microsoft.Extensions.SecretManager.Tools (`dotnet user-secrets`)
- Optional: you can remove the **<DotNetCliToolReference>** element for `Microsoft.VisualStudio.Web.CodeGeneration.Tools`. You can replace this tool with a globally installed version by running `dotnet tool install -g dotnet-aspnet-codegenerator`.
- For 2.1, a Razor Class Library is the recommended solution to distribute Razor files. If your app uses embedded views, or otherwise relies on runtime compilation of Razor files, add

`<CopyRefAssembliesToPublishDirectory>true</CopyRefAssembliesToPublishDirectory>` to a `<PropertyGroup>` in your project file.

The following markup shows the template-generated 2.0 project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.3" PrivateAssets="All" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.4"
PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.3" />
    <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.2" />
    <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.4" />
  </ItemGroup>
</Project>
```

The following markup shows the template-generated 2.1 project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1"
PrivateAssets="All" />
  </ItemGroup>

</Project>
```

Rules for projects targeting the shared framework

A *shared framework* is a set of assemblies (*.dll* files) that are not in the app's folders. The shared framework must be installed on the machine to run the app. For more information, see The shared framework.

ASP.NET Core 2.1 includes the following shared frameworks:

- Microsoft.AspNetCore.App
- Microsoft.AspNetCore.All

The version specified by the package reference is the *minimum required* version. For example, a project referencing the 2.1.1 versions of these packages won't run on a machine with only the 2.1.0 runtime installed.

Known issues for projects targeting a shared framework:

- The .NET Core 2.1.300 SDK (first included in Visual Studio 15.6) set the implicit version of `Microsoft.AspNetCore.App` to 2.1.0 which caused conflicts with Entity Framework Core 2.1.1. The recommended solution is to upgrade the .NET Core SDK to 2.1.301 or later. For more information, see Packages that share dependencies with Microsoft.AspNetCore.App cannot reference patch versions.

- All projects that must use `Microsoft.AspNetCore.All` or `Microsoft.AspNetCore.App` should add a package reference for the package in the project file, even if they contain a project reference to another project using `Microsoft.AspNetCore.All` or `Microsoft.AspNetCore.App`.

  Example:

  - `MyApp` has a package reference to `Microsoft.AspNetCore.App`.
  - `MyApp.Tests` has a project reference to `MyApp.csproj`.

  Add a package reference for `Microsoft.AspNetCore.App` to `MyApp.Tests`. For more information, see Integration testing is hard to set up and may break on shared framework servicing.

Update to the 2.1 Docker images

In ASP.NET Core 2.1, the Docker images migrated to the dotnet/dotnet-docker GitHub repository. The following table shows the Docker image and tag changes:

2.0	2.1
microsoft/aspnetcore:2.0	microsoft/dotnet:2.1-aspnetcore-runtime
microsoft/aspnetcore-build:2.0	microsoft/dotnet:2.1-sdk

Change the `FROM` lines in your *Dockerfile* to use the new image names and tags in the preceding table's 2.1 column. For more information, see Migrating from aspnetcore docker repos to dotnet.

Changes to take advantage of the new code-based idioms that are recommended in ASP.NET Core 2.1

Changes to Main

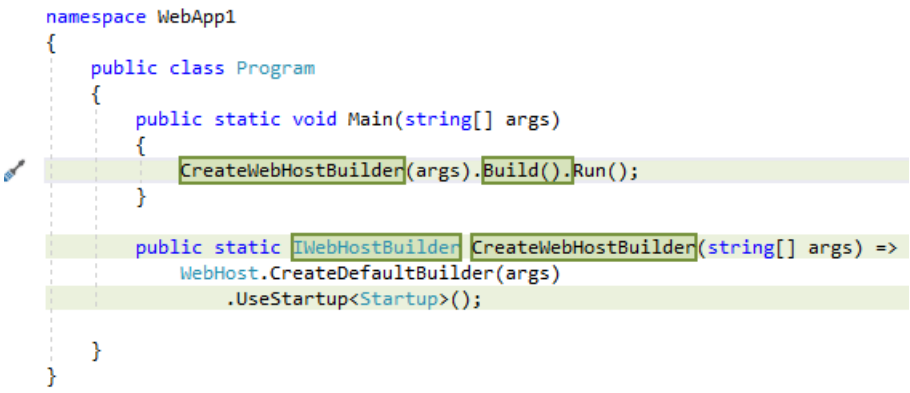
The following images show the changes made to the templated generated *Program.cs* file.

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The preceding image shows the 2.0 version with the deletions in red.

The following image shows the 2.1 code. The code in green replaced the 2.0 version:



```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();

    }
}
```

The following code shows the 2.1 version of *Program.cs*:

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

The new `Main` replaces the call to `BuildWebHost` with CreateWebHostBuilder. IWebHostBuilder was added to support a new integration test infrastructure.

Changes to Startup

The following code shows the changes to 2.1 template generated code. All changes are newly added code, except that `UseBrowserLink` has been removed:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace WebApp1
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                // This lambda determines whether user consent for non-essential cookies is needed for a given
request.
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });


            services.AddMvc()
                .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            // If the app uses Session or TempData based on Session:
            // app.UseSession();

            app.UseMvc();
        }
    }
}
```

The preceding code changes are detailed in:

- GDPR support in ASP.NET Core for `CookiePolicyOptions` and `UseCookiePolicy`.
- HTTP Strict Transport Security Protocol (HSTS) for `UseHsts`.
- Require HTTPS for `UseHttpsRedirection`.
- SetCompatibilityVersion for `SetCompatibilityVersion(CompatibilityVersion.Version_2_1)`.

Changes to authentication code

ASP.NET Core 2.1 provides ASP.NET Core Identity as a Razor Class Library (RCL).

The default 2.1 Identity UI doesn't currently provide significant new features over the 2.0 version. Replacing Identity with the RCL package is optional. The advantages to replacing the template generated Identity code with the RCL version include:

- Many files are moved out of your source tree.
- Any bug fixes or new features to Identity are included in the Microsoft.AspNetCore.App metapackage. You automatically get the updated Identity when `Microsoft.AspNetCore.App` is updated.

If you've made non-trivial changes to the template generated Identity code:

- The preceding advantages probably do **not** justify converting to the RCL version.
- You can keep your ASP.NET Core 2.0 Identity code, it's fully supported.

Identity 2.1 exposes endpoints with the `Identity` area. For example, the follow table shows examples of Identity endpoints that change from 2.0 to 2.1:

2.0 URL	2.1 URL
/Account/Login	/Identity/Account/Login
/Account/Logout	/Identity/Account/Logout
/Account/Manage	/Identity/Account/Manage

Applications that have code using Identity and replace 2.0 Identity UI with the 2.1 Identity Library need to take into account Identity URLs have `/Identity` segment prepended to the URIs. One way to handle the new Identity endpoints is to set up redirects, for example from `/Account/Login` to `/Identity/Account/Login`.

Update Identity to version 2.1

The following options are available to update Identity to 2.1.

- Use the Identity UI 2.0 code with no changes. Using Identity UI 2.0 code is fully supported. This is a good approach when significant changes have been made to the generated Identity code.
- Delete your existing Identity 2.0 code and Scaffold Identity into your project. Your project will use the ASP.NET Core Identity Razor Class Library. You can generate code and UI for any of the Identity UI code that you modified. Apply your code changes to the newly scaffolded UI code.
- Delete your existing Identity 2.0 code and Scaffold Identity into your project with the option to **Override all files**.

Replace Identity 2.0 UI with the Identity 2.1 Razor Class Library

This section outlines the steps to replace the ASP.NET Core 2.0 template generated Identity code with the ASP.NET Core Identity Razor Class Library. The following steps are for a Razor Pages project, but the approach for an MVC project is similar.

- Verify the project file is updated to use 2.1 versions
- Delete the following folders and all the files in them:
  - *Controllers*
  - *Pages/Account/*
  - *Extensions*
- Build the project.

- Scaffold Identity into your project:
  - Select the projects exiting *_Layout.cshtml* file.
  - Select the **+** icon on the right side of the **Data context class**. Accept the default name.
  - Select **Add** to create a new Data context class. Creating a new data context is required for to scaffold. You remove the new data context in the next section.

Update after scaffolding Identity

- Delete the Identity scaffolder generated `IdentityDbContext` derived class in the *Areas/Identity/Data/* folder.
- Delete *Areas/Identity/IdentityHostingStartup.cs*.
- Update the *_LoginPartial.cshtml* file:
  - Move *Pages/_LoginPartial.cshtml* to *Pages/Shared/_LoginPartial.cshtml*.
  - Add `asp-area="Identity"` to the form and anchor links.
  - Update the `<form />` element to
    `<form asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new { area = "" })" method="post" id="logoutForm" class="navbar-right">`
    .

  The following code shows the updated *_LoginPartial.cshtml* file:

```
@using Microsoft.AspNetCore.Identity

@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new {
area = "" })" method="post" id="logoutForm" class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="Identity" asp-page="/Account/Manage/Index" title="Manage">Hello
@UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="Identity" asp-page="/Account/Register">Register</a></li>
        <li><a asp-area="Identity" asp-page="/Account/Login">Log in</a></li>
    </ul>
}
```

Update `ConfigureServices` with the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Register no-op EmailSender used by account confirmation and password reset
    // during development
    services.AddSingleton<IEmailSender, EmailSender>();
}
```

Changes to Razor Pages projects Razor files

The layout file

- Move *Pages/_Layout.cshtml* to *Pages/Shared/_Layout.cshtml*
- In *Areas/Identity/Pages/_ViewStart.cshtml*, change `Layout = "/Pages/_Layout.cshtml"` to `Layout = "/Pages/Shared/_Layout.cshtml"`.
- The *_Layout.cshtml* file has the following changes:
  - `<partial name="_CookieConsentPartial" />` is added. For more information, see GDPR support in ASP.NET Core.
  - jQuery changes from 2.2.0 to 3.3.1.

_ValidationScriptsPartial.cshtml

- *Pages/_ValidationScriptsPartial.cshtml* moves to *Pages/Shared/_ValidationScriptsPartial.cshtml*.
- *jquery.validate/1.14.0* changes to *jquery.validate/1.17.0*.

New files

The following files are added:

- *Privacy.cshtml*
- *Privacy.cshtml.cs*

See GDPR support in ASP.NET Core for information on the preceding files.

Changes to MVC projects Razor files

The layout file

The *Layout.cshtml* file has the following changes:

- `<partial name="_CookieConsentPartial" />` is added.
- jQuery changes from 2.2.0 to 3.3.1

_ValidationScriptsPartial.cshtml

jquery.validate/1.14.0 changes to *jquery.validate/1.17.0*

New files and action methods

The following are added:

- *Views/Home/Privacy.cshtml*

- The `Privacy` action method is added to the Home controller.

See GDPR support in ASP.NET Core for information on the preceding files.

Changes to the launchSettings.json file

As ASP.NET Core apps now use HTTPS by default, the *Properties/launchSettings.json* file has changed.

The following JSON shows the earlier 2.0 template-generated *launchSettings.json* file:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:1799/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:1798/"
    }
  }
}
```

The following JSON shows the new 2.1 template-generated *launchSettings.json* file:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:39191",
      "sslPort": 44390
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

For more information, see Enforce HTTPS in ASP.NET Core.

Breaking changes

FileResult Range header

FileResult no longer processes the Accept-Ranges header by default. To enable the `Accept-Ranges` header, set EnableRangeProcessing to `true`.

ControllerBase.File and PhysicalFile Range header

The following ControllerBase methods no longer processes the Accept-Ranges header by default:

- Overloads of ControllerBase.File
- ControllerBase.PhysicalFile

To enable the `Accept-Ranges` header, set the `EnableRangeProcessing` parameter to `true`.

Additional changes

- If hosting the app on Windows with IIS, install the latest .NET Core Hosting Bundle.
- SetCompatibilityVersion
- Transport configuration

Migrate from ASP.NET Core 1.x to 2.0

9/22/2020 • 7 minutes to read • Edit Online

By Scott Addie

In this article, we walk you through updating an existing ASP.NET Core 1.x project to ASP.NET Core 2.0. Migrating your application to ASP.NET Core 2.0 enables you to take advantage of many new features and performance improvements.

Existing ASP.NET Core 1.x applications are based off of version-specific project templates. As the ASP.NET Core framework evolves, so do the project templates and the starter code contained within them. In addition to updating the ASP.NET Core framework, you need to update the code for your application.

Prerequisites

See Get Started with ASP.NET Core.

Update Target Framework Moniker (TFM)

Projects targeting .NET Core should use the TFM of a version greater than or equal to .NET Core 2.0. Search for the `<TargetFramework>` node in the *.csproj* file, and replace its inner text with `netcoreapp2.0`:

```
<TargetFramework>netcoreapp2.0</TargetFramework>
```

Projects targeting .NET Framework should use the TFM of a version greater than or equal to .NET Framework 4.6.1. Search for the `<TargetFramework>` node in the *.csproj* file, and replace its inner text with `net461`:

```
<TargetFramework>net461</TargetFramework>
```

> **NOTE**
> .NET Core 2.0 offers a much larger surface area than .NET Core 1.x. If you're targeting .NET Framework solely because of missing APIs in .NET Core 1.x, targeting .NET Core 2.0 is likely to work.

If the project file contains `<RuntimeFrameworkVersion>1.{sub-version}</RuntimeFrameworkVersion>`, see this GitHub issue.

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to use the 2.0 version installed on your machine:

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

Update package references

The *.csproj* file in a 1.x project lists each NuGet package used by the project.

In an ASP.NET Core 2.0 project targeting .NET Core 2.0, a single metapackage reference in the *.csproj* file replaces the collection of packages:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
</ItemGroup>
```

All the features of ASP.NET Core 2.0 and Entity Framework Core 2.0 are included in the metapackage.

ASP.NET Core 2.0 projects targeting .NET Framework should continue to reference individual NuGet packages. Update the `Version` attribute of each `<PackageReference />` node to 2.0.0.

For example, here's the list of `<PackageReference />` nodes used in a typical ASP.NET Core 2.0 project targeting .NET Framework:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.Cookies" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0"
PrivateAssets="All" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink" Version="2.0.0" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0"
PrivateAssets="All" />
</ItemGroup>
```

Update .NET Core CLI tools

In the *.csproj* file, update the `Version` attribute of each `<DotNetCliToolReference />` node to 2.0.0.

For example, here's the list of CLI tools used in a typical ASP.NET Core 2.0 project targeting .NET Core 2.0:

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

Rename Package Target Fallback property

The *.csproj* file of a 1.x project used a `PackageTargetFallback` node and variable:

```
<PackageTargetFallback>$(PackageTargetFallback);portable-net45+win8+wp8+wpa81;</PackageTargetFallback>
```

Rename both the node and variable to `AssetTargetFallback`:

```
<AssetTargetFallback>$(AssetTargetFallback);portable-net45+win8+wp8+wpa81;</AssetTargetFallback>
```

Update Main method in Program.cs

In 1.x projects, the `Main` method of *Program.cs* looked like this:

```
using System.IO;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore1App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseApplicationInsights()
                .Build();

            host.Run();
        }
    }
}
```

In 2.0 projects, the `Main` method of *Program.cs* has been simplified:

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The adoption of this new 2.0 pattern is highly recommended and is required for product features like Entity Framework (EF) Core Migrations to work. For example, running `Update-Database` from the Package Manager Console window or `dotnet ef database update` from the command line (on projects converted to ASP.NET Core 2.0) generates the following error:

```
Unable to create an object of type '<Context>'. Add an implementation of
'IDesignTimeDbContextFactory<Context>' to the project, or see https://go.microsoft.com/fwlink/?linkid=851728
for additional patterns supported at design time.
```

Add configuration providers

In 1.x projects, adding configuration providers to an app was accomplished via the `Startup` constructor. The steps involved creating an instance of `ConfigurationBuilder`, loading applicable providers (environment variables, app settings, etc.), and initializing a member of `IConfigurationRoot`.

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; }
```

The preceding example loads the `Configuration` member with configuration settings from *appsettings.json* as well as any *appsettings.<EnvironmentName>.json* file matching the `IHostingEnvironment.EnvironmentName` property. The location of these files is at the same path as *Startup.cs*.

In 2.0 projects, the boilerplate configuration code inherent to 1.x projects runs behind-the-scenes. For example, environment variables and app settings are loaded at startup. The equivalent *Startup.cs* code is reduced to `IConfiguration` initialization with the injected instance:

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

To remove the default providers added by `WebHostBuilder.CreateDefaultBuilder`, invoke the `Clear` method on the `IConfigurationBuilder.Sources` property inside of `ConfigureAppConfiguration`. To add providers back, utilize the `ConfigureAppConfiguration` method in *Program.cs*:

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureAppConfiguration((hostContext, config) =>
        {
            // delete all default configuration providers
            config.Sources.Clear();
            config.AddJsonFile("myconfig.json", optional: true);
        })
        .Build();
```

The configuration used by the `CreateDefaultBuilder` method in the preceding code snippet can be seen here.

For more information, see Configuration in ASP.NET Core.

Move database initialization code

In 1.x projects using EF Core 1.x, a command such as `dotnet ef migrations add` does the following:

1. Instantiates a `Startup` instance
2. Invokes the `ConfigureServices` method to register all services with dependency injection (including `DbContext` types)
3. Performs its requisite tasks

In 2.0 projects using EF Core 2.0, `Program.BuildWebHost` is invoked to obtain the application services. Unlike 1.x, this has the additional side effect of invoking `Startup.Configure`. If your 1.x app invoked database initialization code in its `Configure` method, unexpected problems can occur. For example, if the database doesn't yet exist, the seeding code runs before the EF Core Migrations command execution. This problem causes a `dotnet ef migrations list` command to fail if the database doesn't yet exist.

Consider the following 1.x seed initialization code in the `Configure` method of *Startup.cs*:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

SeedData.Initialize(app.ApplicationServices);
```

In 2.0 projects, move the `SeedData.Initialize` call to the `Main` method of *Program.cs*:

```
var host = BuildWebHost(args);

using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    try
    {
        // Requires using RazorPagesMovie.Models;
        SeedData.Initialize(services);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}

host.Run();
```

As of 2.0, it's bad practice to do anything in `BuildWebHost` except build and configure the web host. Anything that's about running the application should be handled outside of `BuildWebHost` — typically in the `Main` method of *Program.cs*.

Review Razor view compilation setting

Faster application startup time and smaller published bundles are of utmost importance to you. For these reasons, Razor view compilation is enabled by default in ASP.NET Core 2.0.

Setting the `MvcRazorCompileOnPublish` property to true is no longer required. Unless you're disabling view compilation, the property may be removed from the *.csproj* file.

When targeting .NET Framework, you still need to explicitly reference the Microsoft.AspNetCore.Mvc.Razor.ViewCompilation NuGet package in your *.csproj* file:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation" Version="2.0.0" PrivateAssets="All"
/>
```

Rely on Application Insights "light-up" features

Effortless setup of application performance instrumentation is important. You can now rely on the new Application Insights "light-up" features available in the Visual Studio 2017 tooling.

ASP.NET Core 1.1 projects created in Visual Studio 2017 added Application Insights by default. If you're not using the Application Insights SDK directly, outside of *Program.cs* and *Startup.cs*, follow these steps:

1. If targeting .NET Core, remove the following `<PackageReference />` node from the *.csproj* file:

   ```
   <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore" Version="2.0.0" />
   ```

2. If targeting .NET Core, remove the `UseApplicationInsights` extension method invocation from *Program.cs*:

   ```
   public static void Main(string[] args)
   {
       var host = new WebHostBuilder()
           .UseKestrel()
           .UseContentRoot(Directory.GetCurrentDirectory())
           .UseIISIntegration()
           .UseStartup<Startup>()
           .UseApplicationInsights()
           .Build();

       host.Run();
   }
   ```

3. Remove the Application Insights client-side API call from *_Layout.cshtml*. It comprises the following two lines of code:

   ```
   @inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
   @Html.Raw(JavaScriptSnippet.FullScript)
   ```

If you are using the Application Insights SDK directly, continue to do so. The 2.0 metapackage includes the latest version of Application Insights, so a package downgrade error appears if you're referencing an older version.

Adopt authentication/Identity improvements

ASP.NET Core 2.0 has a new authentication model and a number of significant changes to ASP.NET Core Identity. If you created your project with Individual User Accounts enabled, or if you have manually added authentication or Identity, see Migrate Authentication and Identity to ASP.NET Core 2.0.

Additional resources

- Breaking Changes in ASP.NET Core 2.0

Migrate authentication and Identity to ASP.NET Core 2.0

9/22/2020 • 9 minutes to read • Edit Online

By Scott Addie and Hao Kung

ASP.NET Core 2.0 has a new model for authentication and Identity that simplifies configuration by using services. ASP.NET Core 1.x applications that use authentication or Identity can be updated to use the new model as outlined below.

Update namespaces

In 1.x, classes such `IdentityRole` and `IdentityUser` were found in the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` namespace.

In 2.0, the Microsoft.AspNetCore.Identity namespace became the new home for several of such classes. With the default Identity code, affected classes include `ApplicationUser` and `Startup`. Adjust your `using` statements to resolve the affected references.

Authentication Middleware and services

In 1.x projects, authentication is configured via middleware. A middleware method is invoked for each authentication scheme you want to support.

The following 1.x example configures Facebook authentication with Identity in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseIdentity();
    app.UseFacebookAuthentication(new FacebookOptions {
        AppId = Configuration["auth:facebook:appid"],
        AppSecret = Configuration["auth:facebook:appsecret"]
    });
}
```

In 2.0 projects, authentication is configured via services. Each authentication scheme is registered in the `ConfigureServices` method of *Startup.cs*. The `UseIdentity` method is replaced with `UseAuthentication`.

The following 2.0 example configures Facebook authentication with Identity in *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

    // If you want to tweak Identity cookies, they're no longer part of IdentityOptions.
    services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
    services.AddAuthentication()
            .AddFacebook(options =>
            {
                options.AppId = Configuration["auth:facebook:appid"];
                options.AppSecret = Configuration["auth:facebook:appsecret"];
            });
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory) {
    app.UseAuthentication();
}
```

The `UseAuthentication` method adds a single authentication middleware component, which is responsible for automatic authentication and the handling of remote authentication requests. It replaces all of the individual middleware components with a single, common middleware component.

Below are 2.0 migration instructions for each major authentication scheme.

Cookie-based authentication

Select one of the two options below, and make the necessary changes in *Startup.cs*:

1. Use cookies with Identity
   - Replace `UseIdentity` with `UseAuthentication` in the `Configure` method:

     ```
     app.UseAuthentication();
     ```

   - Invoke the `AddIdentity` method in the `ConfigureServices` method to add the cookie authentication services.
   - Optionally, invoke the `ConfigureApplicationCookie` or `ConfigureExternalCookie` method in the `ConfigureServices` method to tweak the Identity cookie settings.

     ```
     services.AddIdentity<ApplicationUser, IdentityRole>()
             .AddEntityFrameworkStores<ApplicationDbContext>()
             .AddDefaultTokenProviders();

     services.ConfigureApplicationCookie(options => options.LoginPath = "/Account/LogIn");
     ```

2. Use cookies without Identity
   - Replace the `UseCookieAuthentication` method call in the `Configure` method with `UseAuthentication`:

     ```
     app.UseAuthentication();
     ```

   - Invoke the `AddAuthentication` and `AddCookie` methods in the `ConfigureServices` method:

```
// If you don't want the cookie to be automatically authenticated and assigned to
HttpContext.User,
// remove the CookieAuthenticationDefaults.AuthenticationScheme parameter passed to
AddAuthentication.
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
        .AddCookie(options =>
        {
            options.LoginPath = "/Account/LogIn";
            options.LogoutPath = "/Account/LogOff";
        });
```

JWT Bearer Authentication

Make the following changes in *Startup.cs*:

- Replace the `UseJwtBearerAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddJwtBearer` method in the `ConfigureServices` method:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.Audience = "http://localhost:5001/";
            options.Authority = "http://localhost:5000/";
        });
```

  This code snippet doesn't use Identity, so the default scheme should be set by passing `JwtBearerDefaults.AuthenticationScheme` to the `AddAuthentication` method.

OpenID Connect (OIDC) authentication

Make the following changes in *Startup.cs*:

- Replace the `UseOpenIdConnectAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddOpenIdConnect` method in the `ConfigureServices` method:

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.Authority = Configuration["auth:oidc:authority"];
    options.ClientId = Configuration["auth:oidc:clientid"];
});
```

- Replace the `PostLogoutRedirectUri` property in the `OpenIdConnectOptions` action with `SignedOutRedirectUri`:

```
.AddOpenIdConnect(options =>
{
    options.SignedOutRedirectUri = "https://contoso.com";
});
```

Facebook authentication

Make the following changes in *Startup.cs*:

- Replace the `UseFacebookAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddFacebook` method in the `ConfigureServices` method:

```
services.AddAuthentication()
        .AddFacebook(options =>
        {
            options.AppId = Configuration["auth:facebook:appid"];
            options.AppSecret = Configuration["auth:facebook:appsecret"];
        });
```

Google authentication

Make the following changes in *Startup.cs*:

- Replace the `UseGoogleAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddGoogle` method in the `ConfigureServices` method:

```
services.AddAuthentication()
        .AddGoogle(options =>
        {
            options.ClientId = Configuration["auth:google:clientid"];
            options.ClientSecret = Configuration["auth:google:clientsecret"];
        });
```

Microsoft Account authentication

For more information on Microsoft account authentication, see this GitHub issue.

Make the following changes in *Startup.cs*:

- Replace the `UseMicrosoftAccountAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddMicrosoftAccount` method in the `ConfigureServices` method:

```
services.AddAuthentication()
        .AddMicrosoftAccount(options =>
        {
            options.ClientId = Configuration["auth:microsoft:clientid"];
            options.ClientSecret = Configuration["auth:microsoft:clientsecret"];
        });
```

Twitter authentication

Make the following changes in *Startup.cs*:

- Replace the `UseTwitterAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
app.UseAuthentication();
```

- Invoke the `AddTwitter` method in the `ConfigureServices` method:

```
services.AddAuthentication()
        .AddTwitter(options =>
        {
            options.ConsumerKey = Configuration["auth:twitter:consumerkey"];
            options.ConsumerSecret = Configuration["auth:twitter:consumersecret"];
        });
```

Setting default authentication schemes

In 1.x, the `AutomaticAuthenticate` and `AutomaticChallenge` properties of the AuthenticationOptions base class were intended to be set on a single authentication scheme. There was no good way to enforce this.

In 2.0, these two properties have been removed as properties on the individual `AuthenticationOptions` instance. They can be configured in the `AddAuthentication` method call within the `ConfigureServices` method of *Startup.cs*:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme);
```

In the preceding code snippet, the default scheme is set to `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookies").

Alternatively, use an overloaded version of the `AddAuthentication` method to set more than one property. In the following overloaded method example, the default scheme is set to `CookieAuthenticationDefaults.AuthenticationScheme`. The authentication scheme may alternatively be specified within your individual `[Authorize]` attributes or authorization policies.

```
services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
});
```

Define a default scheme in 2.0 if one of the following conditions is true:

- You want the user to be automatically signed in
- You use the `[Authorize]` attribute or authorization policies without specifying schemes

An exception to this rule is the `AddIdentity` method. This method adds cookies for you and sets the default authenticate and challenge schemes to the application cookie `IdentityConstants.ApplicationScheme`. Additionally, it sets the default sign-in scheme to the external cookie `IdentityConstants.ExternalScheme`.

Use HttpContext authentication extensions

The `IAuthenticationManager` interface is the main entry point into the 1.x authentication system. It has been replaced with a new set of `HttpContext` extension methods in the `Microsoft.AspNetCore.Authentication` namespace.

For example, 1.x projects reference an `Authentication` property:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

In 2.0 projects, import the `Microsoft.AspNetCore.Authentication` namespace, and delete the `Authentication` property references:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Windows Authentication (HTTP.sys / IISIntegration)

There are two variations of Windows authentication:

- The host only allows authenticated users. This variation isn't affected by the 2.0 changes.
- The host allows both anonymous and authenticated users. This variation is affected by the 2.0 changes. For example, the app should allow anonymous users at the IIS or HTTP.sys layer but authorize users at the controller level. In this scenario, set the default scheme in the `Startup.ConfigureServices` method.

  For Microsoft.AspNetCore.Server.IISIntegration, set the default scheme to `IISDefaults.AuthenticationScheme` :

  ```
  using Microsoft.AspNetCore.Server.IISIntegration;

  services.AddAuthentication(IISDefaults.AuthenticationScheme);
  ```

  For Microsoft.AspNetCore.Server.HttpSys, set the default scheme to `HttpSysDefaults.AuthenticationScheme` :

  ```
  using Microsoft.AspNetCore.Server.HttpSys;

  services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
  ```

  Failure to set the default scheme prevents the authorize (challenge) request from working with the following exception:

  > `System.InvalidOperationException` : No authenticationScheme was specified, and there was no DefaultChallengeScheme found.

For more information, see Configure Windows Authentication in ASP.NET Core.

IdentityCookieOptions instances

A side effect of the 2.0 changes is the switch to using named options instead of cookie options instances. The ability to customize the Identity cookie scheme names is removed.

For example, 1.x projects use constructor injection to pass an `IdentityCookieOptions` parameter into

AccountController.cs and *ManageController.cs*. The external cookie authentication scheme is accessed from the provided instance:

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IOptions<IdentityCookieOptions> identityCookieOptions,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _externalCookieScheme = identityCookieOptions.Value.ExternalCookieAuthenticationScheme;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

The aforementioned constructor injection becomes unnecessary in 2.0 projects, and the `_externalCookieScheme` field can be deleted:

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

1.x projects used the `_externalCookieScheme` field as follows:

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

In 2.0 projects, replace the preceding code with the following. The `IdentityConstants.ExternalScheme` constant can be used directly.

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Resolve the newly added `SignOutAsync` call by importing the following namespace:

```
using Microsoft.AspNetCore.Authentication;
```

Add IdentityUser POCO navigation properties

The Entity Framework (EF) Core navigation properties of the base `IdentityUser` POCO (Plain Old CLR Object) have been removed. If your 1.x project used these properties, manually add them back to the 2.0 project:

```
/// <summary>
/// Navigation property for the roles this user belongs to.
/// </summary>
public virtual ICollection<IdentityUserRole<int>> Roles { get; } = new List<IdentityUserRole<int>>();

/// <summary>
/// Navigation property for the claims this user possesses.
/// </summary>
public virtual ICollection<IdentityUserClaim<int>> Claims { get; } = new List<IdentityUserClaim<int>>();

/// <summary>
/// Navigation property for this users login accounts.
/// </summary>
public virtual ICollection<IdentityUserLogin<int>> Logins { get; } = new List<IdentityUserLogin<int>>();
```

To prevent duplicate foreign keys when running EF Core Migrations, add the following to your `IdentityDbContext` class' `OnModelCreating` method (after the `base.OnModelCreating();` call):

```
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
    // Customize the ASP.NET Core Identity model and override the defaults if needed.
    // For example, you can rename the ASP.NET Core Identity table names and more.
    // Add your customizations after calling base.OnModelCreating(builder);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Claims)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Logins)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Roles)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);
}
```

Replace GetExternalAuthenticationSchemes

The synchronous method `GetExternalAuthenticationSchemes` was removed in favor of an asynchronous version. 1.x projects have the following code in *Controllers/ManageController.cs*:

```
var otherLogins = _signInManager.GetExternalAuthenticationSchemes().Where(auth => userLogins.All(ul =>
auth.AuthenticationScheme != ul.LoginProvider)).ToList();
```

This method appears in *Views/Account/Login.cshtml* too:

```
@{
    var loginProviders = SignInManager.GetExternalAuthenticationSchemes().ToList();
    if (loginProviders.Count == 0)
    {
        <div>
            <p>
                There are no external authentication services configured. See <a
href="https://go.microsoft.com/fwlink/?LinkID=532715">this article</a>
                for details on setting up this ASP.NET application to support logging in via external
services.
            </p>
        </div>
    }
    else
    {
        <form asp-controller="Account" asp-action="ExternalLogin" asp-route-returnurl="@ViewData["ReturnUrl"]"
method="post" class="form-horizontal">
            <div>
                <p>
                    @foreach (var provider in loginProviders)
                    {
                        <button type="submit" class="btn btn-default" name="provider"
value="@provider.AuthenticationScheme" title="Log in using your @provider.DisplayName
account">@provider.AuthenticationScheme</button>
                    }
                </p>
            </div>
        </form>
    }
}
```

In 2.0 projects, use the GetExternalAuthenticationSchemesAsync method. The change in *ManageController.cs* resembles the following code:

```
var schemes = await _signInManager.GetExternalAuthenticationSchemesAsync();
var otherLogins = schemes.Where(auth => userLogins.All(ul => auth.Name != ul.LoginProvider)).ToList();
```

In *Login.cshtml*, the `AuthenticationScheme` property accessed in the `foreach` loop changes to `Name`:

```
@{
    var loginProviders = (await SignInManager.GetExternalAuthenticationSchemesAsync()).ToList();
    if (loginProviders.Count == 0)
    {
        <div>
            <p>
                There are no external authentication services configured. See <a
href="https://go.microsoft.com/fwlink/?LinkID=532715">this article</a>
                for details on setting up this ASP.NET application to support logging in via external
services.
            </p>
        </div>
    }
    else
    {
        <form asp-controller="Account" asp-action="ExternalLogin" asp-route-returnurl="@ViewData["ReturnUrl"]"
method="post" class="form-horizontal">
            <div>
                <p>
                    @foreach (var provider in loginProviders)
                    {
                        <button type="submit" class="btn btn-default" name="provider" value="@provider.Name"
title="Log in using your @provider.DisplayName account">@provider.DisplayName</button>
                    }
                </p>
            </div>
        </form>
    }
}
```

ManageLoginsViewModel property change

A `ManageLoginsViewModel` object is used in the `ManageLogins` action of *ManageController.cs*. In 1.x projects, the object's `OtherLogins` property return type is `IList<AuthenticationDescription>`. This return type requires an import of `Microsoft.AspNetCore.Http.Authentication`:

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore1App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationDescription> OtherLogins { get; set; }
    }
}
```

In 2.0 projects, the return type changes to `IList<AuthenticationScheme>`. This new return type requires replacing the `Microsoft.AspNetCore.Http.Authentication` import with a `Microsoft.AspNetCore.Authentication` import.

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore2App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationScheme> OtherLogins { get; set; }
    }
}
```

Additional resources

For more information, see the Discussion for Auth 2.0 issue on GitHub.

Migrate from ASP.NET to ASP.NET Core

9/22/2020 • 8 minutes to read • Edit Online

By Isaac Levin

This article serves as a reference guide for migrating ASP.NET apps to ASP.NET Core.

Prerequisites

.NET Core SDK 2.2 or later

Target frameworks

ASP.NET Core projects offer developers the flexibility of targeting .NET Core, .NET Framework, or both. See Choosing between .NET Core and .NET Framework for server apps to determine which target framework is most appropriate.

When targeting .NET Framework, projects need to reference individual NuGet packages.

Targeting .NET Core allows you to eliminate numerous explicit package references, thanks to the ASP.NET Core metapackage. Install the `Microsoft.AspNetCore.App` metapackage in your project:

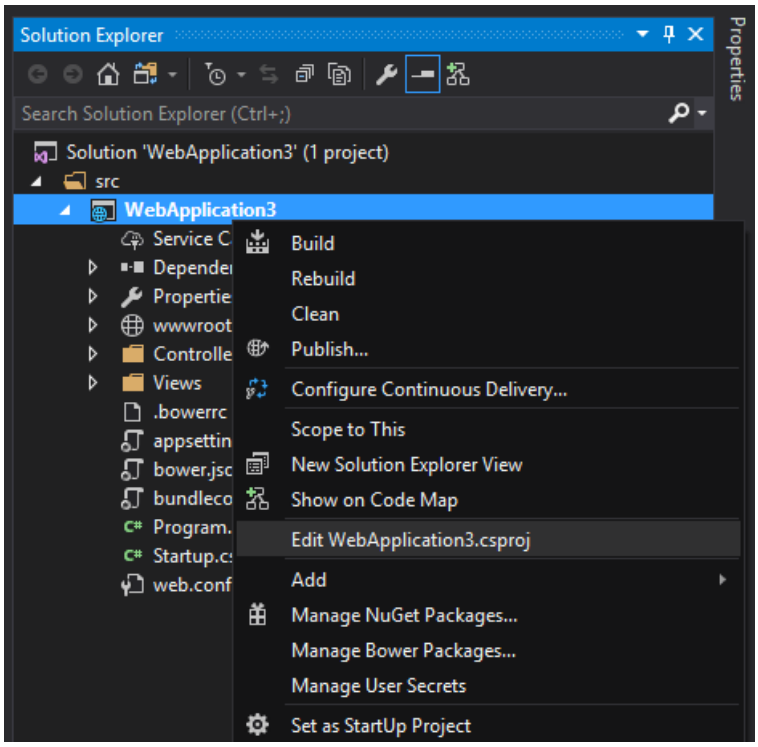
```
<ItemGroup>
   <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

When the metapackage is used, no packages referenced in the metapackage are deployed with the app. The .NET Core Runtime Store includes these assets, and they're precompiled to improve performance. See Microsoft.AspNetCore.App metapackage for ASP.NET Core for more detail.

Project structure differences

The *.csproj* file format has been simplified in ASP.NET Core. Some notable changes include:

- Explicit inclusion of files isn't necessary for them to be considered part of the project. This reduces the risk of XML merge conflicts when working on large teams.
- There are no GUID-based references to other projects, which improves file readability.
- The file can be edited without unloading it in Visual Studio:



Global.asax file replacement

ASP.NET Core introduced a new mechanism for bootstrapping an app. The entry point for ASP.NET applications is the *Global.asax* file. Tasks such as route configuration and filter and area registrations are handled in the *Global.asax* file.

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
```

This approach couples the application and the server to which it's deployed in a way that interferes with the implementation. In an effort to decouple, OWIN was introduced to provide a cleaner way to use multiple frameworks together. OWIN provides a pipeline to add only the modules needed. The hosting environment takes a Startup function to configure services and the app's request pipeline. `Startup` registers a set of middleware with the application. For each request, the application calls each of the middleware components with the head pointer of a linked list to an existing set of handlers. Each middleware component can add one or more handlers to the request handling pipeline. This is accomplished by returning a reference to the handler that's the new head of the list. Each handler is responsible for remembering and invoking the next handler in the list. With ASP.NET Core, the entry point to an application is `Startup`, and you no longer have a dependency on *Global.asax*. When using OWIN with .NET Framework, use something like the following as a pipeline:

```
using Owin;
using System.Web.Http;

namespace WebApi
{
    // Note: By default all requests go through this OWIN pipeline. Alternatively you can turn this off by
adding an appSetting owin:AutomaticAppStartup with value “false”.
    // With this turned off you can still have OWIN apps listening on specific routes by adding routes in
global.asax file using MapOwinPath or MapOwinRoute extensions on RouteTable.Routes
    public class Startup
    {
        // Invoked once at startup to configure your application.
        public void Configuration(IAppBuilder builder)
        {
            HttpConfiguration config = new HttpConfiguration();
            config.Routes.MapHttpRoute("Default", "{controller}/{customerID}", new { controller = "Customer",
customerID = RouteParameter.Optional });

            config.Formatters.XmlFormatter.UseXmlSerializer = true;
            config.Formatters.Remove(config.Formatters.JsonFormatter);
            // config.Formatters.JsonFormatter.UseDataContractJsonSerializer = true;

            builder.UseWebApi(config);
        }
    }
}
```

This configures your default routes, and defaults to XmlSerialization over Json. Add other Middleware to this pipeline as needed (loading services, configuration settings, static files, etc.).

ASP.NET Core uses a similar approach, but doesn't rely on OWIN to handle the entry. Instead, that's done through the *Program.cs* `Main` method (similar to console applications) and `Startup` is loaded through there.

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace WebApplication2
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
    }
}
```

`Startup` must include a `Configure` method. In `Configure`, add the necessary middleware to the pipeline. In the following example (from the default web site template), extension methods configure the pipeline with support for:

- Error pages
- HTTP Strict Transport Security
- HTTP redirection to HTTPS
- ASP.NET Core MVC

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

The host and application have been decoupled, which provides the flexibility of moving to a different platform in the future.

> **NOTE**
>
> For a more in-depth reference to ASP.NET Core Startup and Middleware, see Startup in ASP.NET Core

Store configurations

ASP.NET supports storing settings. These setting are used, for example, to support the environment to which the applications were deployed. A common practice was to store all custom key-value pairs in the `<appSettings>` section of the *Web.config* file:

```
<appSettings>
  <add key="UserName" value="User" />
  <add key="Password" value="Password" />
</appSettings>
```

Applications read these settings using the `ConfigurationManager.AppSettings` collection in the `System.Configuration` namespace:

```
string userName = System.Web.Configuration.ConfigurationManager.AppSettings["UserName"];
string password = System.Web.Configuration.ConfigurationManager.AppSettings["Password"];
```

ASP.NET Core can store configuration data for the application in any file and load them as part of middleware bootstrapping. The default file used in the project templates is *appsettings.json*:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "AppConfiguration": {
    "UserName": "UserName",
    "Password": "Password"
  }
}
```

Loading this file into an instance of `IConfiguration` inside your application is done in *Startup.cs*:

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

The app reads from `Configuration` to get the settings:

```
string userName = Configuration.GetSection("AppConfiguration")["UserName"];
string password = Configuration.GetSection("AppConfiguration")["Password"];
```

There are extensions to this approach to make the process more robust, such as using Dependency Injection (DI) to load a service with these values. The DI approach provides a strongly-typed set of configuration objects.

```
// Assume AppConfiguration is a class representing a strongly-typed version of AppConfiguration section
services.Configure<AppConfiguration>(Configuration.GetSection("AppConfiguration"));
```

> **NOTE**
>
> For a more in-depth reference to ASP.NET Core configuration, see Configuration in ASP.NET Core.

Native dependency injection

An important goal when building large, scalable applications is the loose coupling of components and services. Dependency Injection is a popular technique for achieving this, and it's a native component of ASP.NET Core.

In ASP.NET apps, developers rely on a third-party library to implement Dependency Injection. One such library is Unity, provided by Microsoft Patterns & Practices.

An example of setting up Dependency Injection with Unity is implementing `IDependencyResolver` that wraps a `UnityContainer`:

```
using Microsoft.Practices.Unity;
using System;
using System.Collections.Generic;
using System.Web.Http.Dependencies;

public class UnityResolver : IDependencyResolver
{
    protected IUnityContainer container;

    public UnityResolver(IUnityContainer container)
    {
        if (container == null)
        {
            throw new ArgumentNullException("container");
        }
        this.container = container;
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return container.Resolve(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return container.ResolveAll(serviceType);
        }
        catch (ResolutionFailedException)
        {
            return new List<object>();
        }
    }

    public IDependencyScope BeginScope()
    {
        var child = container.CreateChildContainer();
        return new UnityResolver(child);
    }

    public void Dispose()
    {
        Dispose(true);
    }

    protected virtual void Dispose(bool disposing)
    {
        container.Dispose();
    }
}
```

Create an instance of your `UnityContainer`, register your service, and set the dependency resolver of `HttpConfiguration` to the new instance of `UnityResolver` for your container:

```
public static void Register(HttpConfiguration config)
{
    var container = new UnityContainer();
    container.RegisterType<IProductRepository, ProductRepository>(new HierarchicalLifetimeManager());
    config.DependencyResolver = new UnityResolver(container);

    // Other Web API configuration not shown.
}
```

Inject `IProductRepository` where needed:

```
public class ProductsController : ApiController
{
    private IProductRepository _repository;

    public ProductsController(IProductRepository repository)
    {
        _repository = repository;
    }

    // Other controller methods not shown.
}
```

Because Dependency Injection is part of ASP.NET Core, you can add your service in the `ConfigureServices` method of *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add application services.
    services.AddTransient<IProductRepository, ProductRepository>();
}
```

The repository can be injected anywhere, as was true with Unity.

> **NOTE**
> For more information on dependency injection, see Dependency injection.

Serve static files

An important part of web development is the ability to serve static, client-side assets. The most common examples of static files are HTML, CSS, Javascript, and images. These files need to be saved in the published location of the app (or CDN) and referenced so they can be loaded by a request. This process has changed in ASP.NET Core.

In ASP.NET, static files are stored in various directories and referenced in the views.

In ASP.NET Core, static files are stored in the "web root" (*<content root>/wwwroot*), unless configured otherwise. The files are loaded into the request pipeline by invoking the `UseStaticFiles` extension method from `Startup.Configure`:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

> **NOTE**
> If targeting .NET Framework, install the NuGet package `Microsoft.AspNetCore.StaticFiles`.

For example, an image asset in the *wwwroot/images* folder is accessible to the browser at a location such as `http://<app>/images/<imageFileName>`.

> **NOTE**
> For a more in-depth reference to serving static files in ASP.NET Core, see Static files.

Multi-value cookies

Multi-value cookies aren't supported in ASP.NET Core. Create one cookie per value.

Partial app migration

One approach to partial app migration is to create an IIS sub-application and only move certain routes from ASP.NET 4.x to ASP.NET Core while preserving the URL structure the app. For example, consider the URL structure of the app from the *applicationHost.config* file:

```
<sites>
    <site name="Default Web Site" id="1" serverAutoStart="true">
        <application path="/">
            <virtualDirectory path="/" physicalPath="D:\sites\MainSite\" />
        </application>
        <application path="/api" applicationPool="DefaultAppPool">
            <virtualDirectory path="/" physicalPath="D:\sites\netcoreapi" />
        </application>
        <bindings>
            <binding protocol="http" bindingInformation="*:80:" />
            <binding protocol="https" bindingInformation="*:443:" sslFlags="0" />
        </bindings>
    </site>
 ...
</sites>
```

Directory structure:

```
.
├── MainSite
│   ├── ...
│   └── Web.config
└── NetCoreApi
    ├── ...
    └── web.config
```

[BIND] and Input Formatters

Previous versions of ASP.NET used the `[Bind]` attribute to protect against overposting attacks. Input formatters work differently in ASP.NET Core. The `[Bind]` attribute is no longer designed to prevent overposting when used with input formatters to parse JSON or XML. These attributes affect model binding when the source of data is form data posted with the `x-www-form-urlencoded` content type.

For apps that post JSON information to controllers and use JSON Input Formatters to parse the data, we

recommend replacing the `[Bind]` attribute with a view model that matches the properties defined by the `[Bind]` attribute.

Additional resources

- Porting Libraries to .NET Core

Migrate from ASP.NET MVC to ASP.NET Core MVC

9/22/2020 • 22 minutes to read • Edit Online

This article shows how to start migrating an ASP.NET MVC project to ASP.NET Core MVC. In the process, it highlights related changes from ASP.NET MVC.

Migrating from ASP.NET MVC is a multi-step process. This article covers:

- Initial setup.
- Basic controllers and views.
- Static content.
- Client-side dependencies.

For migrating configuration and Identity code, see Migrate configuration to ASP.NET Core and Migrate Authentication and Identity to ASP.NET Core.

Prerequisites

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Create the starter ASP.NET MVC project

Create an example ASP.NET MVC project in Visual Studio to migrate:

1. From the **File** menu, select **New** > **Project**.
2. Select **ASP.NET Web Application (.NET Framework)** and then select **Next**.
3. Name the project *WebApp1* so the namespace matches the ASP.NET Core project created in the next step. Select **Create**.
4. Select **MVC**, and then select **Create**.

Create the ASP.NET Core project

Create a new solution with a new ASP.NET Core project to migrate to:

1. Launch a second instance of Visual Studio.
2. From the **File** menu, select **New** > **Project**.
3. Select **ASP.NET Web Core Web Application** and then select **Next**.
4. In the **Configure your new project** dialog, Name the project *WebApp1*.
5. Set the location to a different directory than the previous project to use the same project name. Using the same namespace makes it easier to copy code between the two projects. Select **Create**.
6. In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 3.1** are selected. Select the **Web Application (Model-View-Controller)** project template, and select **Create**.

Configure the ASP.NET Core site to use MVC

In ASP.NET Core 3.0 and later projects, .NET Framework is no longer a supported target framework. Your project must target .NET Core. The ASP.NET Core shared framework, which includes MVC, is part of the .NET Core runtime installation. The shared framework is automatically referenced when using the `Microsoft.NET.Sdk.Web` SDK in the project file:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

For more information, see Framework reference.

In ASP.NET Core, the `Startup` class:

- Replaces *Global.asax*.
- Handles all app startup tasks.

For more information, see App startup in ASP.NET Core.

In the ASP.NET Core project, open the *Startup.cs* file:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            // The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

ASP.NET Core apps must opt in to framework features with middleware. The previous template-generated code adds the following services and middleware:

- The AddControllersWithViews extension method registers MVC service support for controllers, API-related features, and views. For more information on MVC service registration options, see MVC service registration

- The UseStaticFiles extension method adds the static file handler `Microsoft.AspNetCore.StaticFiles`. The `UseStaticFiles` extension method must be called before `UseRouting`. For more information, see Static files in ASP.NET Core.
- The UseRouting extension method adds routing. For more information, see Routing in ASP.NET Core.

This existing configuration includes what is needed to migrate the example ASP.NET MVC project. For more information on ASP.NET Core middleware options, see App startup in ASP.NET Core.

Migrate controllers and views

In the ASP.NET Core project, a new empty controller class and view class would be added to serve as placeholders using the same names as the controller and view classes in any ASP.NET MVC project to migrate from.

The ASP.NET Core *WebApp1* project already includes a minimal example controller and view by the same name as the ASP.NET MVC project. So those will serve as placeholders for the ASP.NET MVC controller and views to be migrated from the ASP.NET MVC *WebApp1* project.

1. Copy the methods from the ASP.NET MVC `HomeController` to replace the new ASP.NET Core `HomeController` methods. There's no need to change the return type of the action methods. The ASP.NET MVC built-in template's controller action method return type is /dotnet/api/system.web.mvc.actionresult?view=aspnet-mvc-5.2; in ASP.NET Core MVC, the action methods return `IActionResult` instead. `ActionResult` implements `IActionResult`.
2. In the ASP.NET Core project, right-click the *Views/Home* directory, select **Add** > **Existing Item**.
3. In the **Add Existing Item** dialog, navigate to the ASP.NET MVC *WebApp1* project's *Views/Home* directory.
4. Select the *About.cshtml*, *Contact.cshtml*, and *Index.cshtml* Razor view files, then select **Add**, replacing the existing files.

For more information, see Handle requests with controllers in ASP.NET Core MVC and Views in ASP.NET Core MVC.

Test each method

Each controller endpoint can be tested, however, layout and styles are covered later in the document.

1. Run the ASP.NET Core app.
2. Invoke the rendered views from the browser on the running ASP.NET Core app by replacing the current port number with the port number used in the ASP.NET Core project. For example, `https://localhost:44375/home/about`.

Migrate static content

In ASP.NET MVC 5 and earlier, static content was hosted from the web project's root directory and was intermixed with server-side files. In ASP.NET Core, static files are stored within the project's web root directory. The default directory is *{content root}/wwwroot*, but it can be changed. For more information, see Static files in ASP.NET Core.

Copy the static content from the ASP.NET MVC *WebApp1* project to the *wwwroot* directory in the ASP.NET Core *WebApp1* project:

1. In the ASP.NET Core project, right-click the *wwwroot* directory, select **Add** > **Existing Item**.
2. In the **Add Existing Item** dialog, navigate to the ASP.NET MVC *WebApp1* project.
3. Select the *favicon.ico* file, then select **Add**, replacing the existing file.

Migrate the layout files

Copy the ASP.NET MVC project layout files to the ASP.NET Core project:

1. In the ASP.NET Core project, right-click the *Views* directory, select **Add** > **Existing Item**.
2. In the **Add Existing Item** dialog, navigate to the ASP.NET MVC *WebApp1* project's *Views* directory.
3. Select the *_ViewStart.cshtml* file then select **Add**.

Copy the ASP.NET MVC project shared layout files to the ASP.NET Core project:

1. In the ASP.NET Core project, right-click the *Views/Shared* directory, select **Add** > **Existing Item**.
2. In the **Add Existing Item** dialog, navigate to the ASP.NET MVC *WebApp1* project's *Views/Shared* directory.
3. Select the *_Layout.cshtml* file, then select **Add**, replacing the existing file.

In the ASP.NET Core project, open the *_Layout.cshtml* file. Make the following changes to match the completed code shown below:

Update the Bootstrap CSS inclusion to match the completed code below:

1. Replace `@Styles.Render("~/Content/css")` with a `<link>` element to load *bootstrap.css* (see below).
2. Remove `@Scripts.Render("~/bundles/modernizr")`.

The completed replacement markup for Bootstrap CSS inclusion:

```
<link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
```

Update the jQuery and Bootstrap JavaScript inclusion to match the completed code below:

1. Replace `@Scripts.Render("~/bundles/jquery")` with a `<script>` element (see below).
2. Replace `@Scripts.Render("~/bundles/bootstrap")` with a `<script>` element (see below).

The completed replacement markup for jQuery and Bootstrap JavaScript inclusion:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
    integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
crossorigin="anonymous"></script>
```

The updated *_Layout.cshtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet"
          href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
          crossorigin="anonymous">
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class =
"navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
            integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
crossorigin="anonymous"></script>
    @RenderSection("scripts", required: false)
</body>
</html>
```

View the site in the browser. It should render with the expected styles in place.

Configure bundling and minification

ASP.NET Core is compatible with several open-source bundling and minification solutions such as WebOptimizer and other similar libraries. ASP.NET Core doesn't provide a native bundling and minification solution. For information on configuring bundling and minification, see Bundling and Minification.

Solve HTTP 500 errors

There are many problems that can cause an HTTP 500 error message that contains no information on the source of the problem. For example, if the *Views/_ViewImports.cshtml* file contains a namespace that doesn't exist in the project, an HTTP 500 error is generated. By default in ASP.NET Core apps, the `UseDeveloperExceptionPage` extension is added to the `IApplicationBuilder` and executed when the environment is *Development*. This is detailed in the

following code:

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            // The default HSTS value is 30 days. You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

ASP.NET Core converts unhandled exceptions into HTTP 500 error responses. Normally, error details aren't included in these responses to prevent disclosure of potentially sensitive information about the server. For more information, see Developer Exception Page.

Next steps

- Migrate Authentication and Identity to ASP.NET Core

Additional resources

- Introduction to ASP.NET Core Blazor
- Tag Helpers in ASP.NET Core

This article shows how to start migrating an ASP.NET MVC project to ASP.NET Core MVC 2.2. In the process, it highlights many of the things that have changed from ASP.NET MVC. Migrating from ASP.NET MVC is a multi-step process. This article covers:

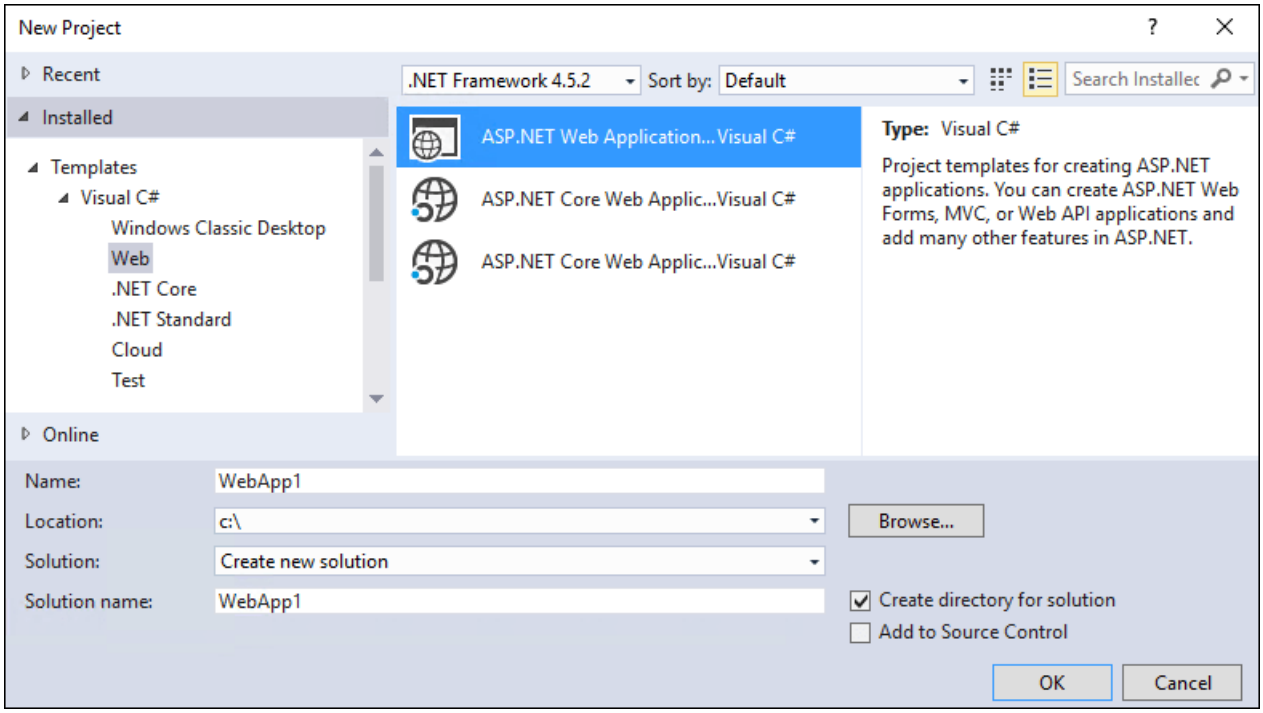
- Initial setup
- Basic controllers and views
- Static content
- Client-side dependencies.

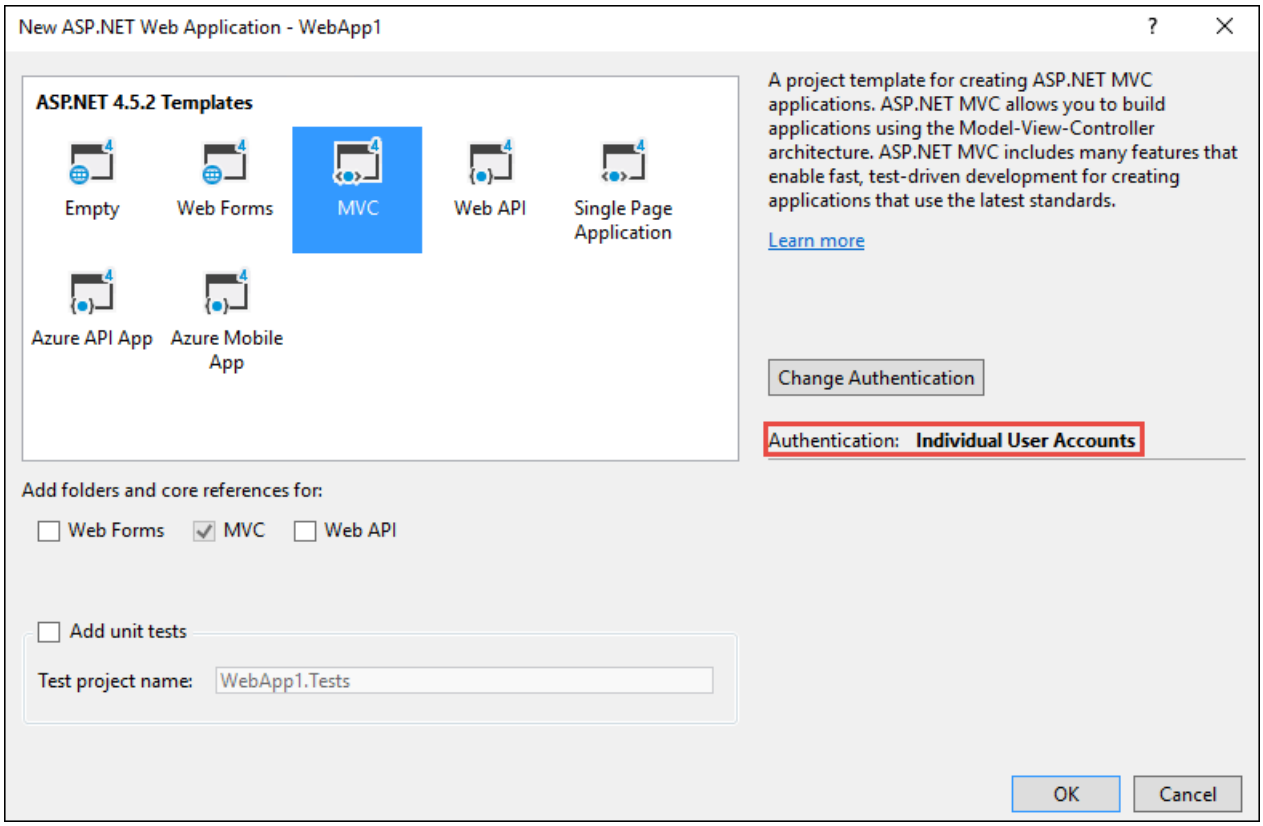
For migrating configuration and Identity code, see Migrate configuration to ASP.NET Core and Migrate Authentication and Identity to ASP.NET Core.

> **NOTE**
> The version numbers in the samples might not be current, update the projects accordingly.

Create the starter ASP.NET MVC project

To demonstrate the upgrade, we'll start by creating an ASP.NET MVC app. Create it with the name *WebApp1* so the namespace matches the ASP.NET Core project created in the next step.

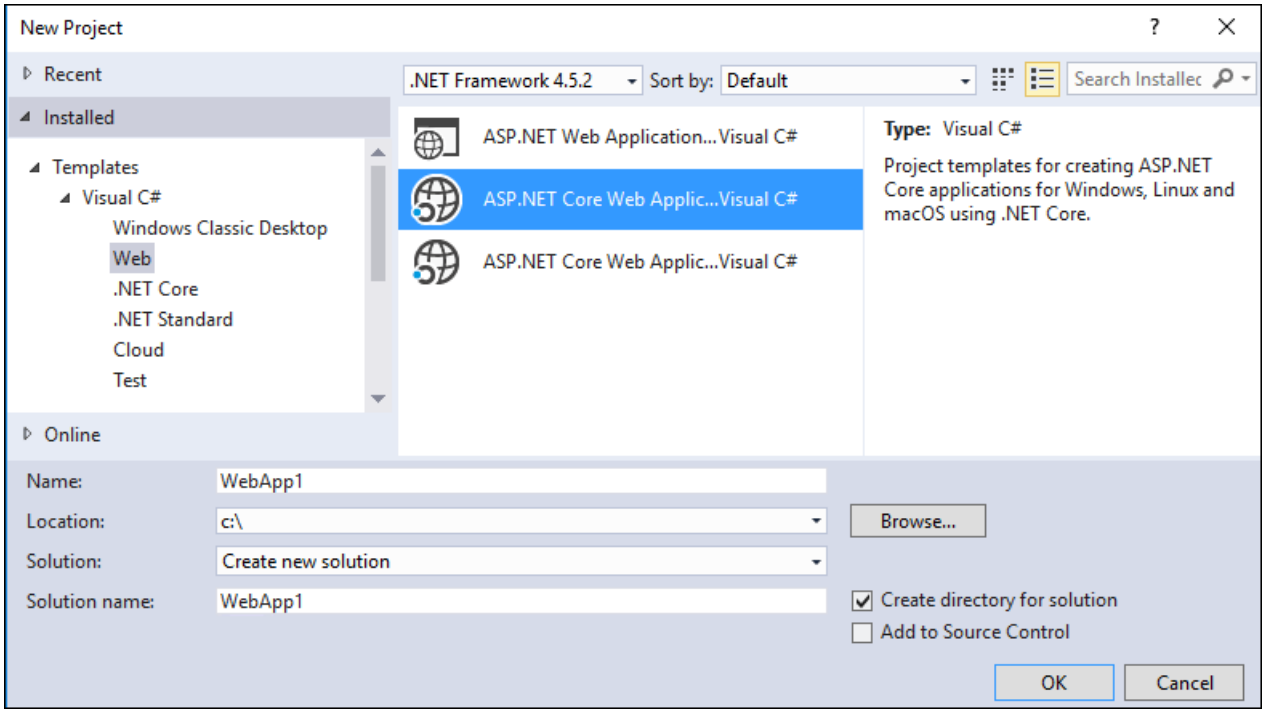


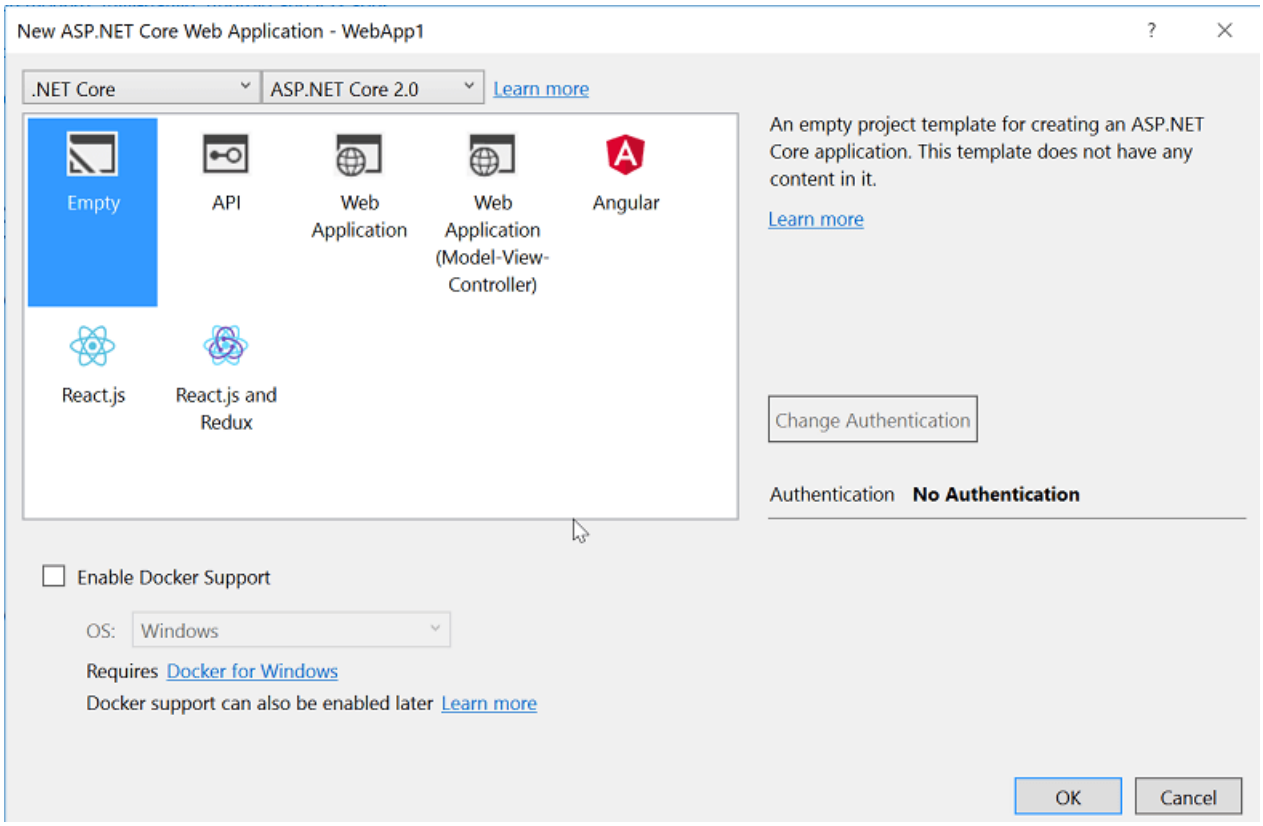


Optional: Change the name of the Solution from *WebApp1* to *Mvc5*. Visual Studio displays the new solution name (*Mvc5*), which makes it easier to tell this project from the next project.

Create the ASP.NET Core project

Create a new *empty* ASP.NET Core web app with the same name as the previous project (*WebApp1*) so the namespaces in the two projects match. Having the same namespace makes it easier to copy code between the two projects. Create this project in a different directory than the previous project to use the same name.





- *Optional:* Create a new ASP.NET Core app using the *Web Application* project template. Name the project *WebApp1*, and select an authentication option of **Individual User Accounts**. Rename this app to *FullAspNetCore*. Creating this project saves time in the conversion. The end result can be viewed in the template-generated code, code can be copied to the conversion project, or compared with the template-generated project.

Configure the site to use MVC

- When targeting .NET Core, the Microsoft.AspNetCore.App metapackage is referenced by default. This package contains packages commonly used by MVC apps. If targeting .NET Framework, package references must be listed individually in the project file.

`Microsoft.AspNetCore.Mvc` is the ASP.NET Core MVC framework. `Microsoft.AspNetCore.StaticFiles` is the static file handler. ASP.NET Core apps explicitly opt in for middleware, such as for serving static files. For more information, see Static files.

- Open the *Startup.cs* file and change the code to match the following:

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseStaticFiles();

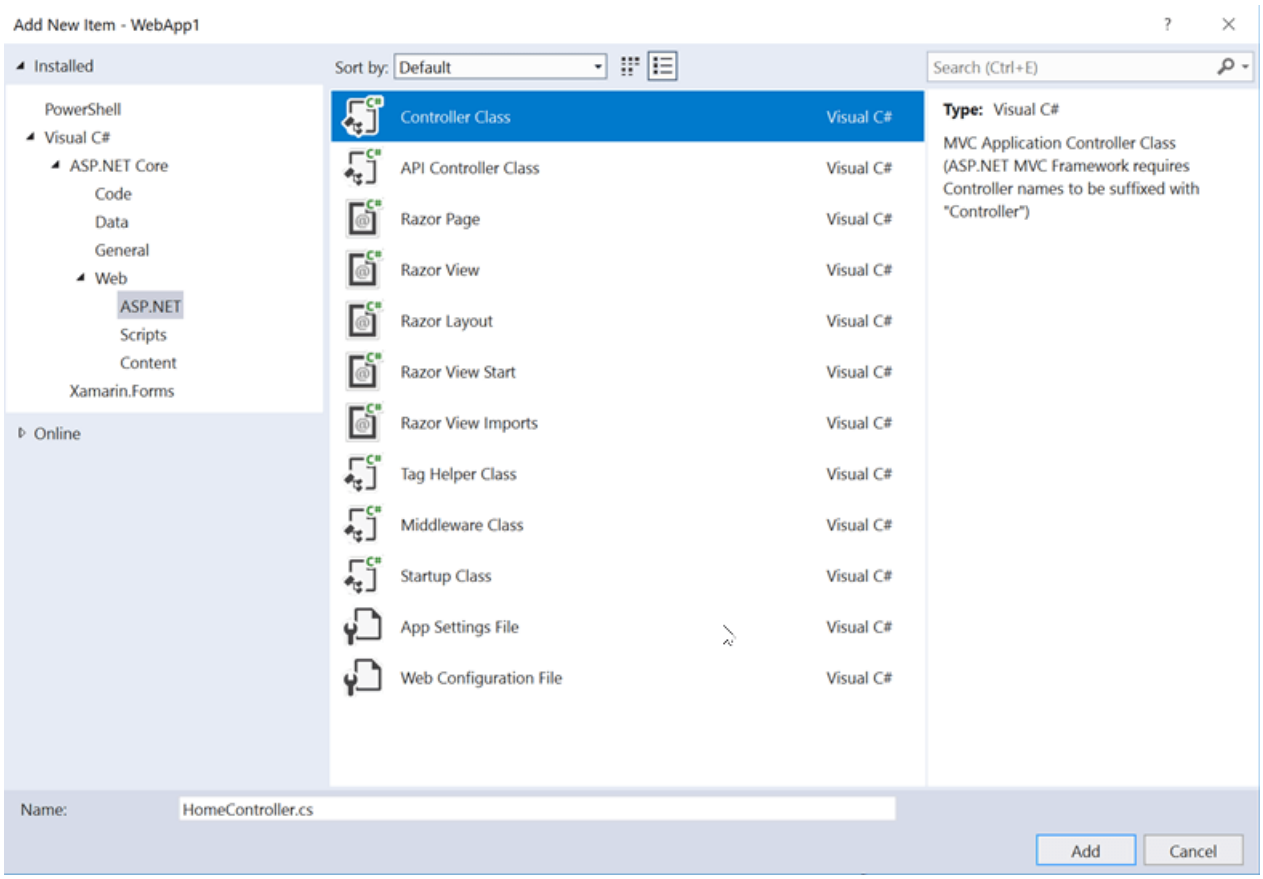
        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

The UseStaticFiles extension method adds the static file handler. For more information, see Application Startup and Routing.

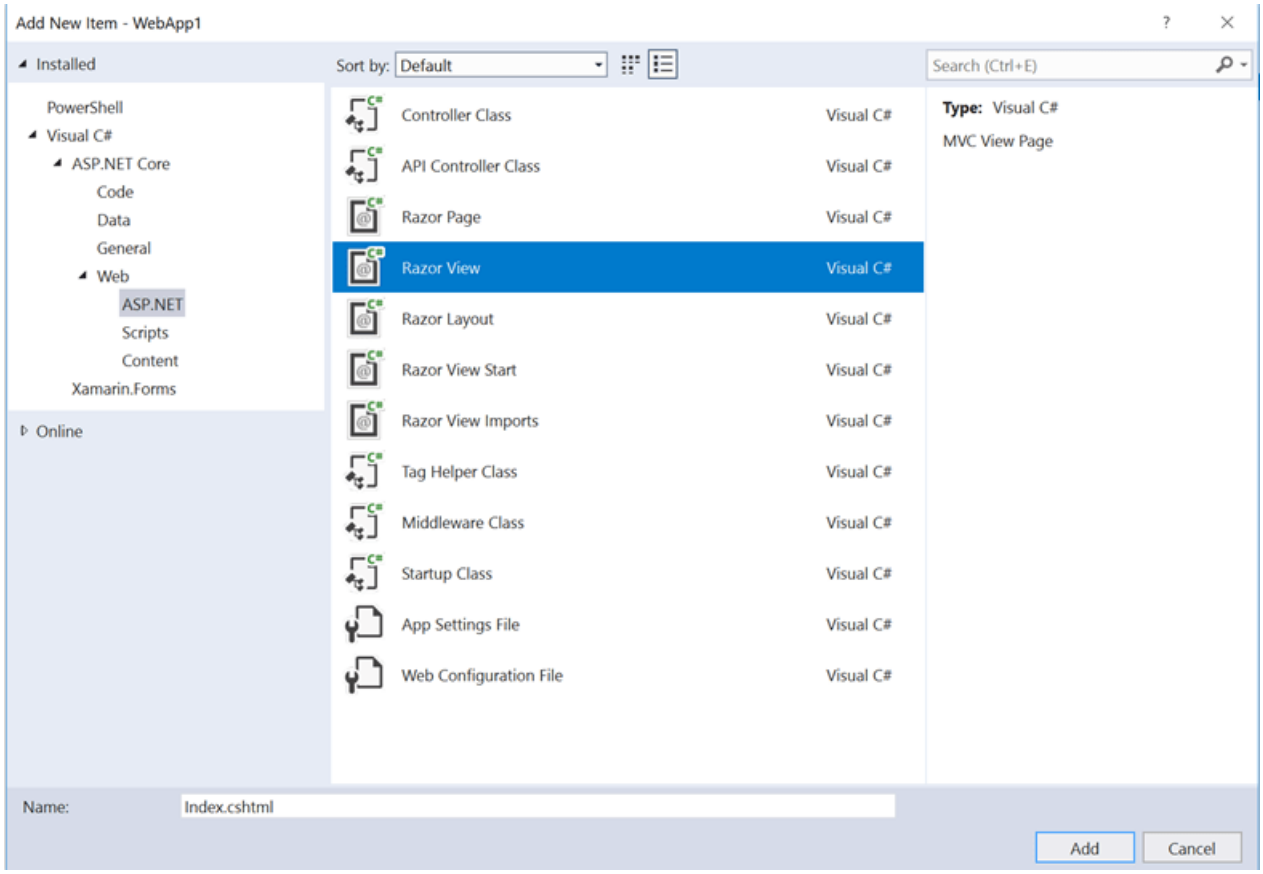
Add a controller and view

In this section, a minimal controller and view are added to serve as placeholders for the ASP.NET MVC controller and views migrated in the next section.

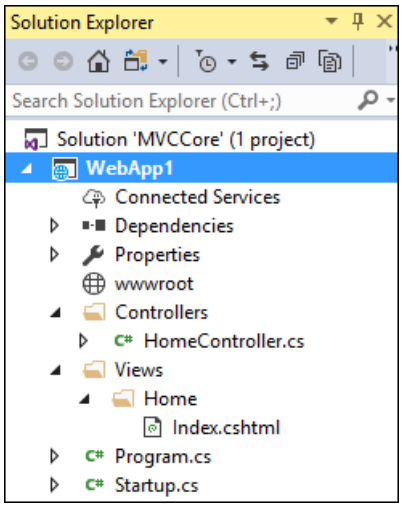
- Add a *Controllers* directory.
- Add a **Controller Class** named *HomeController.cs* to the *Controllers* directory.



- Add a *Views* directory.
- Add a *Views/Home* directory.
- Add a **Razor View** named *Index.cshtml* to the *Views/Home* directory.



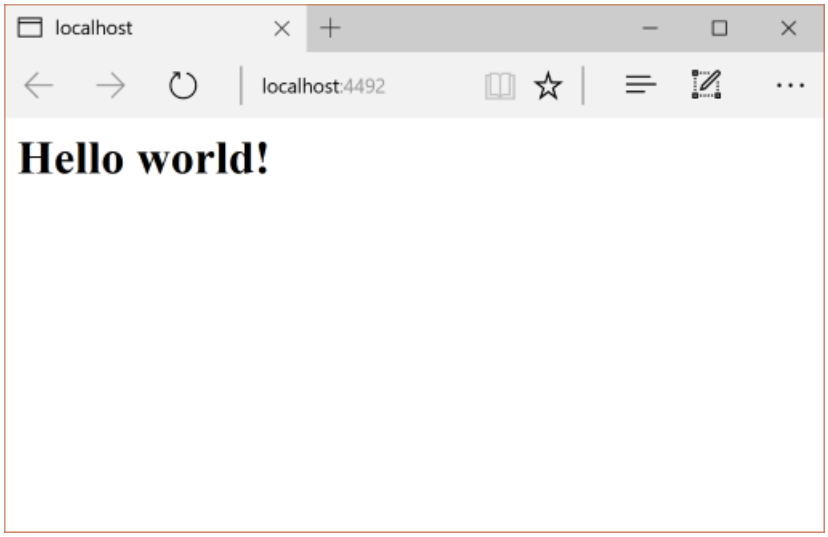
The project structure is shown below:



Replace the contents of the *Views/Home/Index.cshtml* file with the following markup:

```
<h1>Hello world!</h1>
```

Run the app.



For more information, see Controllers and Views.

The following functionality requires migration from the example ASP.NET MVC project to the ASP.NET Core project:

- client-side content (CSS, fonts, and scripts)
- controllers
- views
- models
- bundling
- filters
- Log in/out, Identity (This is done in the next tutorial.)

Controllers and views

- Copy each of the methods from the ASP.NET MVC `HomeController` to the new `HomeController`. In ASP.NET MVC, the built-in template's controller action method return type is

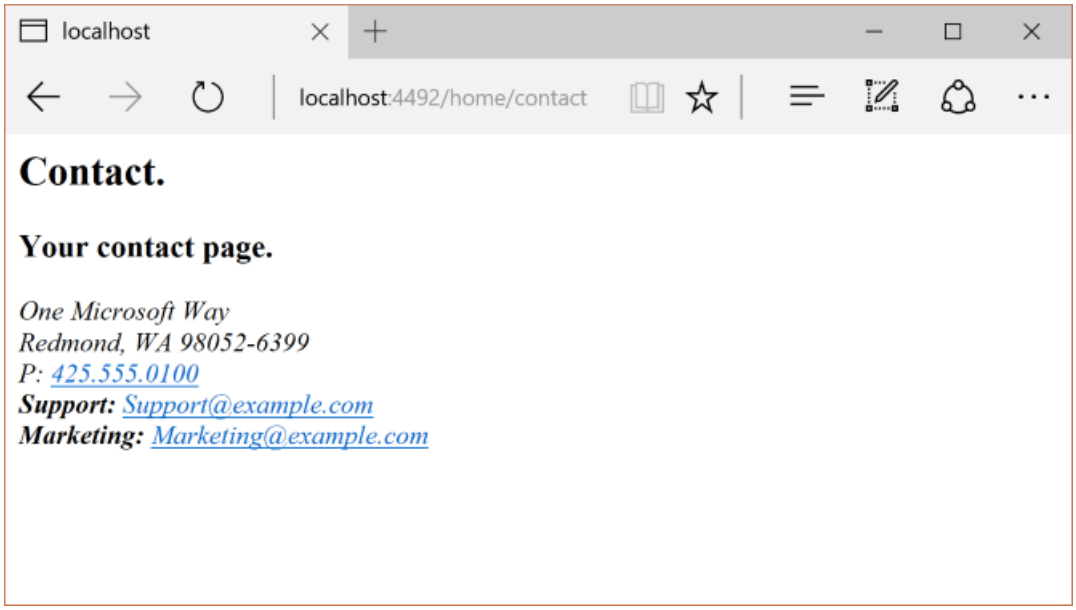
/dotnet/api/system.web.mvc.actionresult?view=aspnet-mvc-5.2; in ASP.NET Core MVC, the action methods return `IActionResult` instead. `ActionResult` implements `IActionResult`, so there's no need to change the return type of the action methods.

- Copy the *About.cshtml*, *Contact.cshtml*, and *Index.cshtml* Razor view files from the ASP.NET MVC project to the ASP.NET Core project.

Test each method

The layout file and styles have not been migrated yet, so the rendered views only contain the content in the view files. The layout file generated links for the `About` and `Contact` views will not be available yet.

Invoke the rendered views from the browser on the running ASP.NET core app by replacing the current port number with the port number used in the ASP.NET core project. For example: `https://localhost:44375/home/about`.



Note the lack of styling and menu items. The styling will be fixed in the next section.

Static content

In ASP.NET MVC 5 and earlier, static content was hosted from the root of the web project and was intermixed with server-side files. In ASP.NET Core, static content is hosted in the *wwwroot* directory. Copy the static content from the ASP.NET MVC app to the *wwwroot* directory in the ASP.NET Core project. In this sample conversion:

- Copy the *favicon.ico* file from the ASP.NET MVC project to the *wwwroot* directory in the ASP.NET Core project.

The ASP.NET MVC project uses Bootstrap for its styling and stores the Bootstrap files in the *Content* and *Scripts* directories. The template, which generated the ASP.NET MVC project, references Bootstrap in the layout file (*Views/Shared/_Layout.cshtml*). The *bootstrap.js* and *bootstrap.css* files could be copied from the ASP.NET MVC project to the *wwwroot* directory in the new project. Instead, this document adds support for Bootstrap (and other client-side libraries) using CDNs, in the next section.

Migrate the layout file

- Copy the *_ViewStart.cshtml* file from the ASP.NET MVC project's *Views* directory into the ASP.NET Core project's *Views* directory. The *_ViewStart.cshtml* file has not changed in ASP.NET Core MVC.
- Create a *Views/Shared* directory.
- *Optional:* Copy *_ViewImports.cshtml* from the *FullAspNetCore* MVC project's *Views* directory into the ASP.NET Core project's *Views* directory. Remove any namespace declaration in the *_ViewImports.cshtml* file.

The *_ViewImports.cshtml* file provides namespaces for all the view files and brings in Tag Helpers. Tag Helpers are used in the new layout file. The *_ViewImports.cshtml* file is new for ASP.NET Core.

- Copy the *_Layout.cshtml* file from the ASP.NET MVC project's *Views/Shared* directory into the ASP.NET Core project's *Views/Shared* directory.

Open *_Layout.cshtml* file and make the following changes (the completed code is shown below):

- Replace `@Styles.Render("~/Content/css")` with a `<link>` element to load *bootstrap.css* (see below).
- Remove `@Scripts.Render("~/bundles/modernizr")`.
- Comment out the `@Html.Partial("_LoginPartial")` line (surround the line with `@*...*@`). For more information, see Migrate Authentication and Identity to ASP.NET Core
- Replace `@Scripts.Render("~/bundles/jquery")` with a `<script>` element (see below).
- Replace `@Scripts.Render("~/bundles/bootstrap")` with a `<script>` element (see below).

The replacement markup for Bootstrap CSS inclusion:

```
<link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
```

The replacement markup for jQuery and Bootstrap JavaScript inclusion:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
    integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
crossorigin="anonymous"></script>
```

The updated *_Layout.cshtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet"
          href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
          crossorigin="anonymous">
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class =
"navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @*@Html.Partial("_LoginPartial")*@
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
            integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
            crossorigin="anonymous"></script>
    @RenderSection("scripts", required: false)
</body>
</html>
```

View the site in the browser. It should now load correctly, with the expected styles in place.

- *Optional:* Try using the new layout file. Copy the layout file from the *FullAspNetCore* project. The new layout file uses Tag Helpers and has other improvements.

Configure bundling and minification

For information about how to configure bundling and minification, see Bundling and Minification.

Solve HTTP 500 errors

There are many problems that can cause an HTTP 500 error messages that contain no information on the source of the problem. For example, if the *Views/_ViewImports.cshtml* file contains a namespace that doesn't exist in the

project, a HTTP 500 error is generated. By default in ASP.NET Core apps, the `UseDeveloperExceptionPage` extension is added to the `IApplicationBuilder` and executed when the configuration is *Development*. See an example in the following code:

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseStaticFiles();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

ASP.NET Core converts unhandled exceptions into HTTP 500 error responses. Normally, error details aren't included in these responses to prevent disclosure of potentially sensitive information about the server. For more information, see Developer Exception Page.

Additional resources

- Introduction to ASP.NET Core Blazor
- Tag Helpers in ASP.NET Core

This article shows how to start migrating an ASP.NET MVC project to ASP.NET Core MVC 2.1. In the process, it highlights many of the things that have changed from ASP.NET MVC. Migrating from ASP.NET MVC is a multi-step process. This article covers:

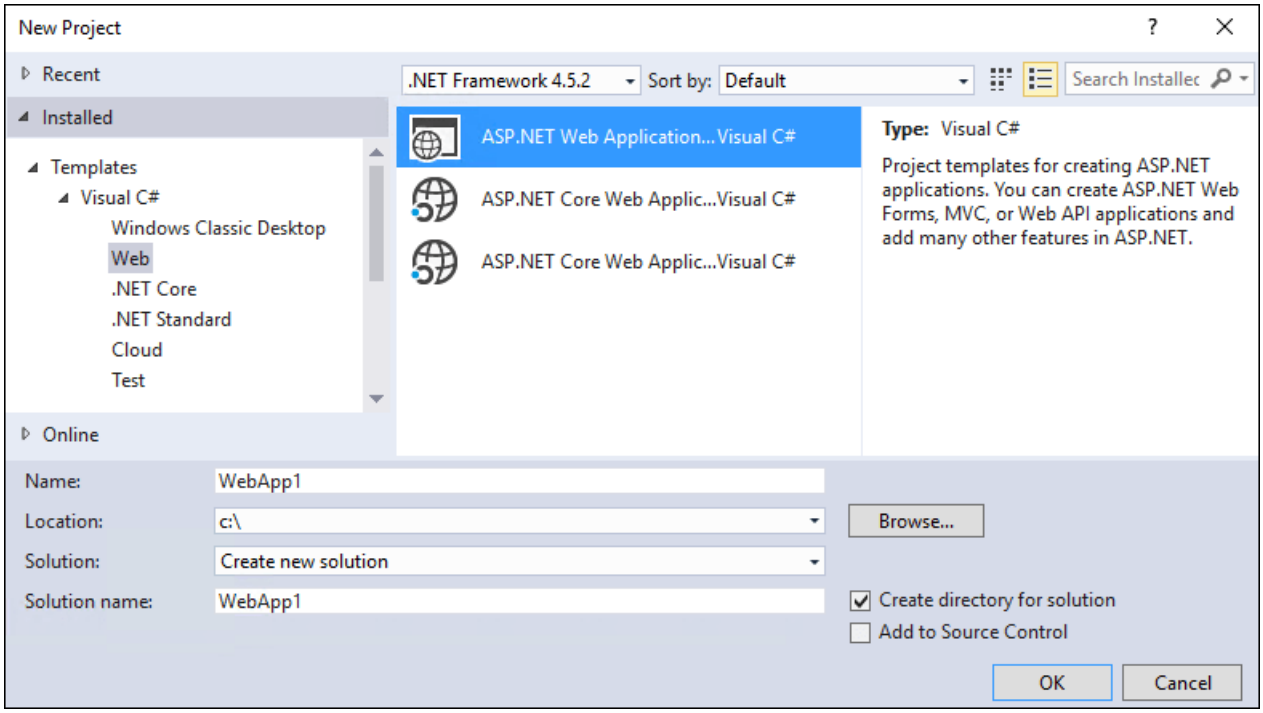
- Initial setup
- Basic controllers and views
- Static content
- Client-side dependencies.

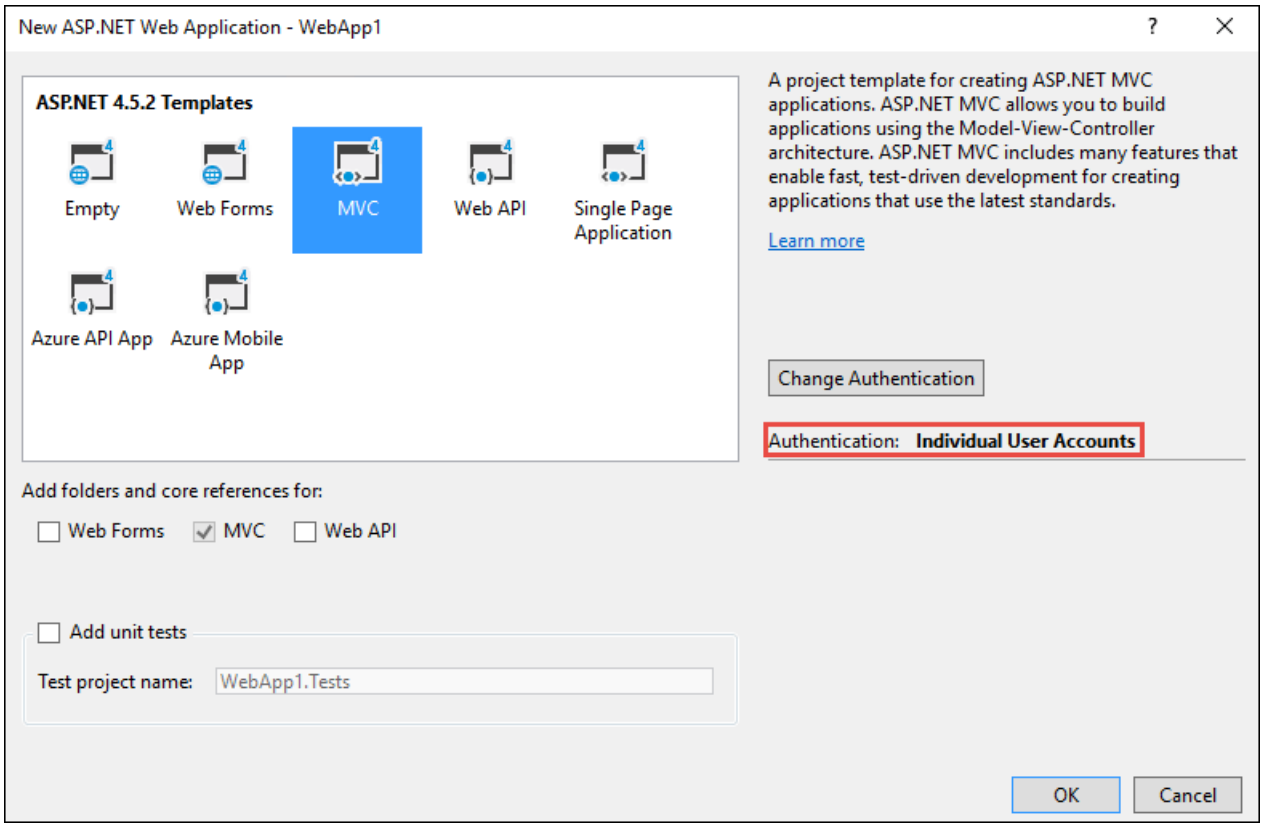
For migrating configuration and Identity code, see Migrate configuration to ASP.NET Core and Migrate Authentication and Identity to ASP.NET Core.

> **NOTE**
> The version numbers in the samples might not be current, update the projects accordingly.

Create the starter ASP.NET MVC project

To demonstrate the upgrade, we'll start by creating a ASP.NET MVC app. Create it with the name *WebApp1* so the namespace matches the ASP.NET Core project created in the next step.

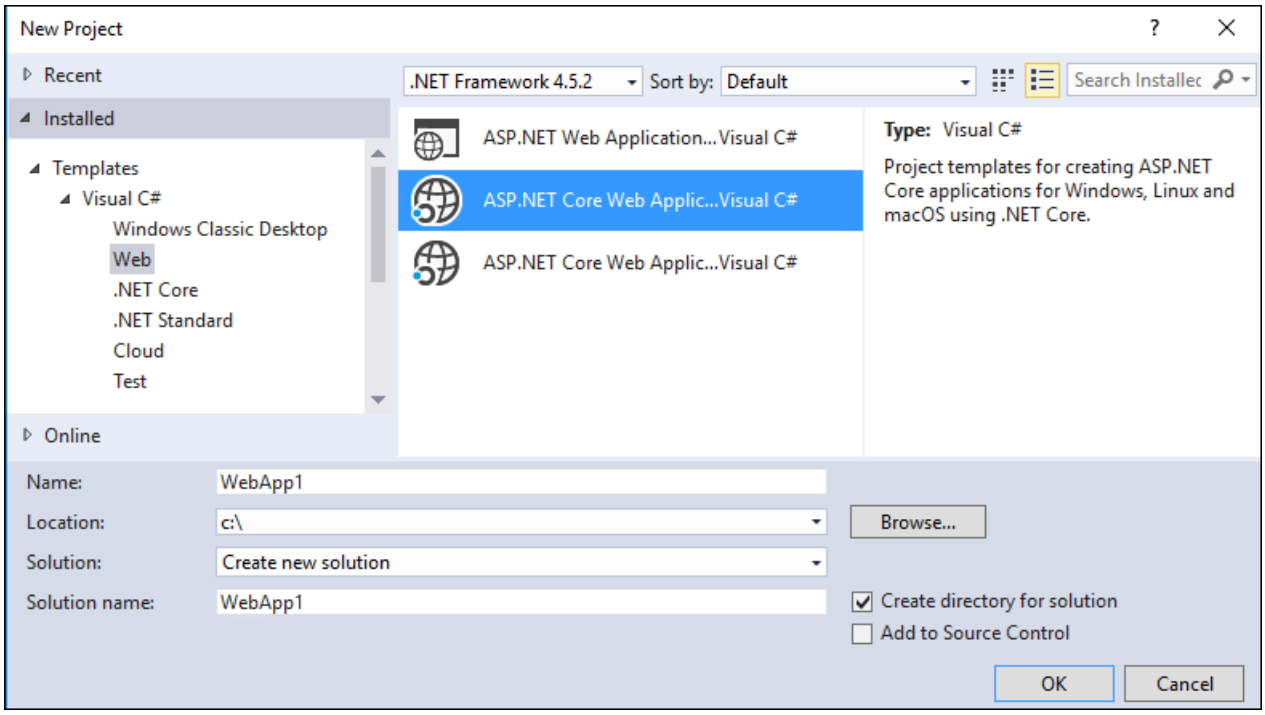


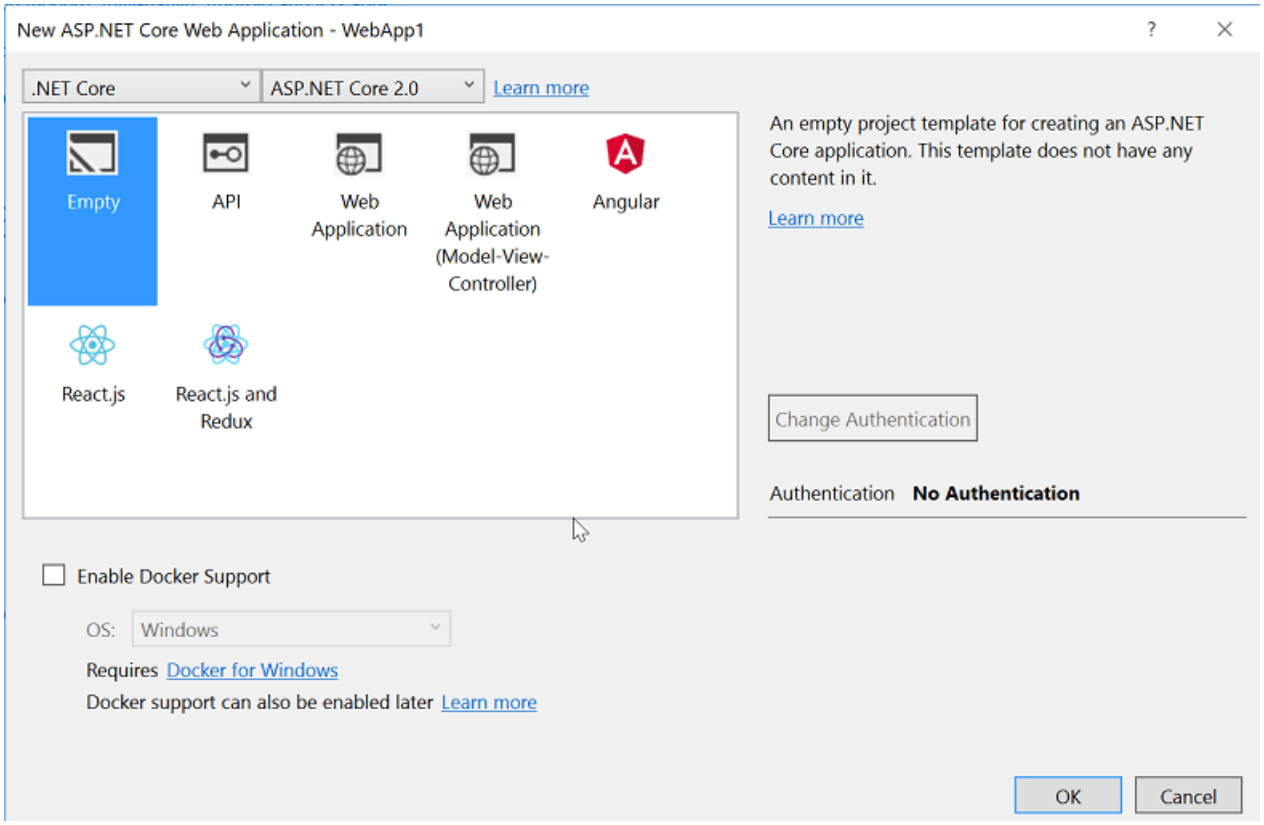


Optional: Change the name of the Solution from *WebApp1* to *Mvc5*. Visual Studio displays the new solution name (*Mvc5*), which makes it easier to tell this project from the next project.

Create the ASP.NET Core project

Create a new *empty* ASP.NET Core web app with the same name as the previous project (*WebApp1*) so the namespaces in the two projects match. Having the same namespace makes it easier to copy code between the two projects. Create this project in a different directory than the previous project to use the same name.





- *Optional:* Create a new ASP.NET Core app using the *Web Application* project template. Name the project *WebApp1*, and select an authentication option of **Individual User Accounts**. Rename this app to *FullAspNetCore*. Creating this project saves time in the conversion. The end result can be viewed in the template-generated code, code can be copied to the conversion project, or compared with the template-generated project.

Configure the site to use MVC

- When targeting .NET Core, the Microsoft.AspNetCore.App metapackage is referenced by default. This package contains packages commonly used by MVC apps. If targeting .NET Framework, package references must be listed individually in the project file.

`Microsoft.AspNetCore.Mvc` is the ASP.NET Core MVC framework. `Microsoft.AspNetCore.StaticFiles` is the static file

handler. ASP.NET Core apps explicitly opt in for middleware, such as for serving static files. For more information, see Static files.

- Open the *Startup.cs* file and change the code to match the following:

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseStaticFiles();

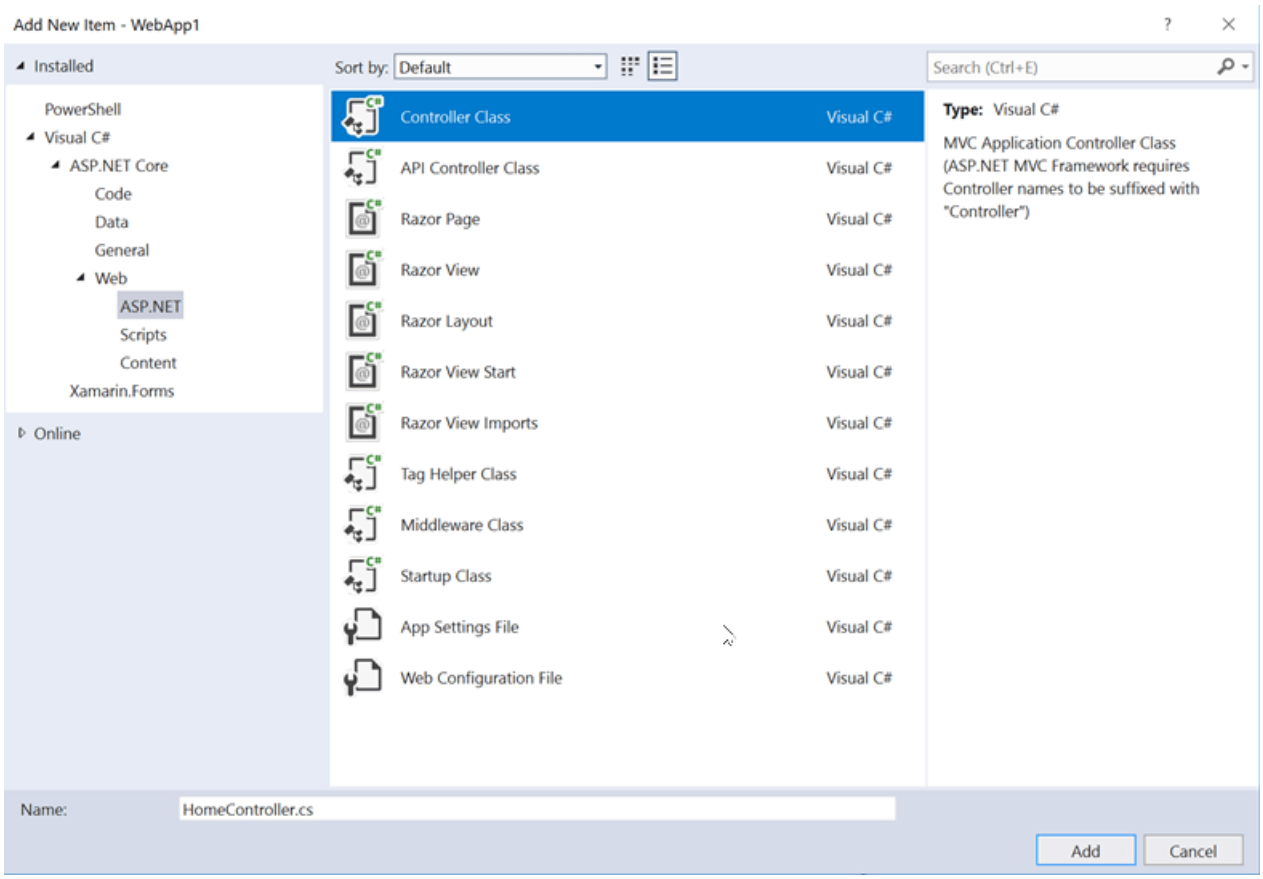
        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

The UseStaticFiles extension method adds the static file handler. The `UseMvc` extension method adds routing. For more information, see Application Startup and Routing.

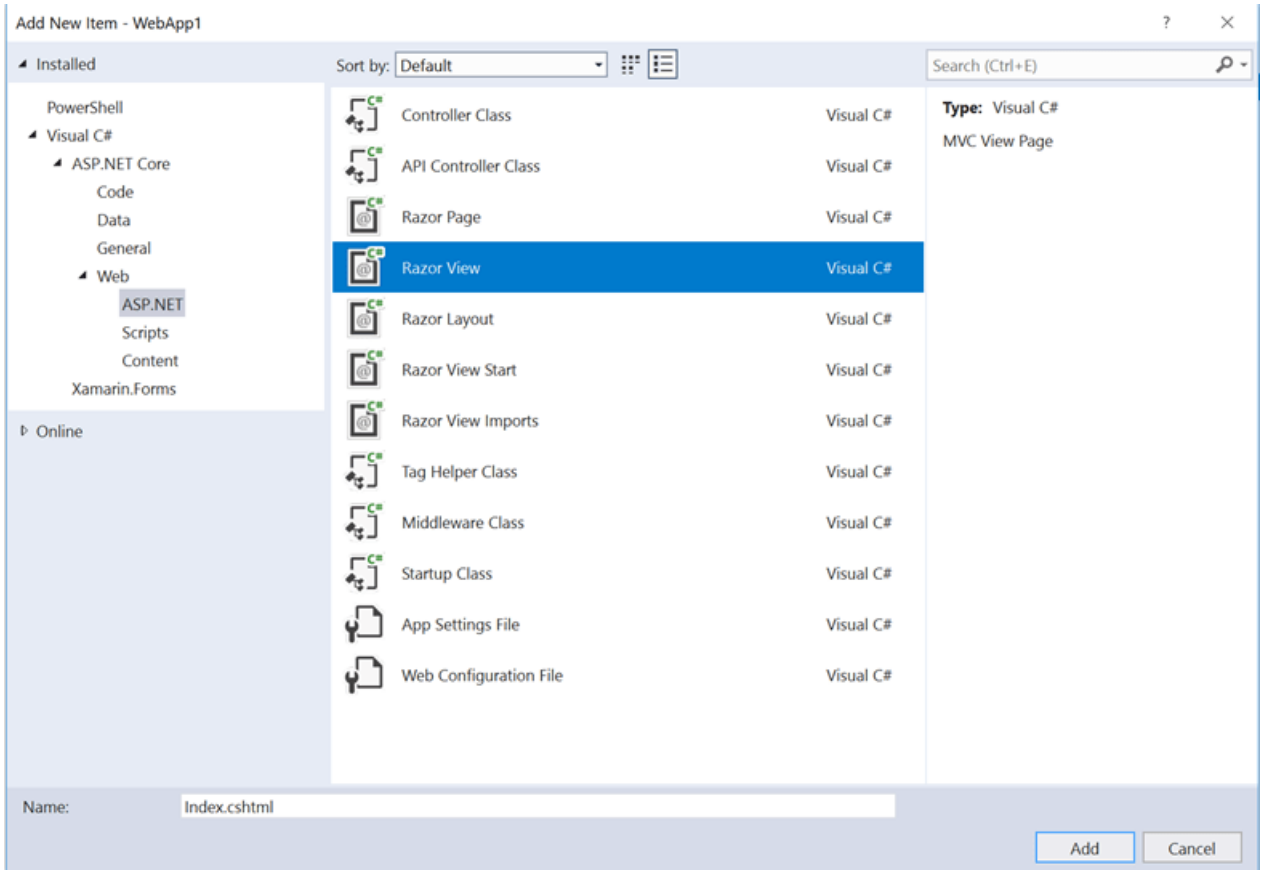
Add a controller and view

In this section, a minimal controller and view are added to serve as placeholders for the ASP.NET MVC controller and views migrated in the next section.

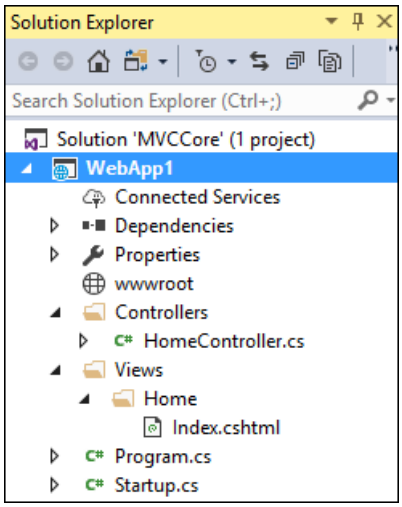
- Add a *Controllers* directory.
- Add a **Controller Class** named *HomeController.cs* to the *Controllers* directory.



- Add a *Views* directory.
- Add a *Views/Home* directory.
- Add a **Razor View** named *Index.cshtml* to the *Views/Home* directory.



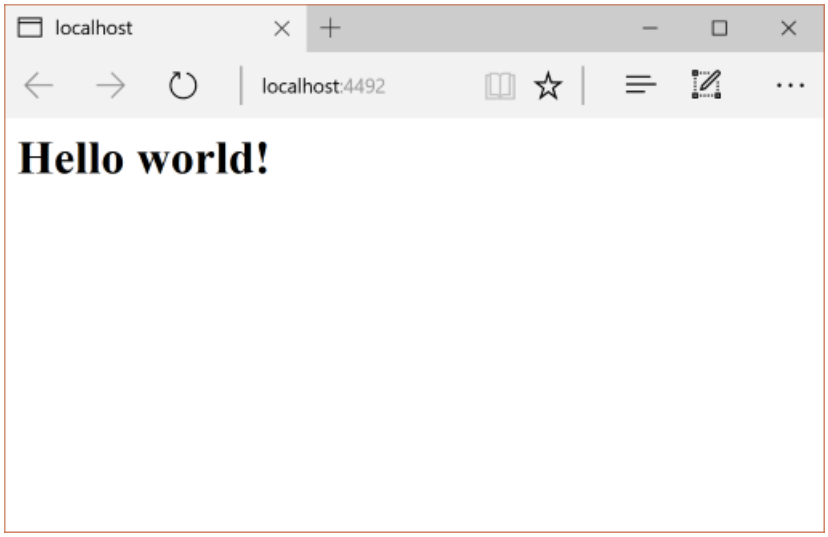
The project structure is shown below:



Replace the contents of the *Views/Home/Index.cshtml* file with the following markup:

```
<h1>Hello world!</h1>
```

Run the app.



For more information, see Controllers and Views.

The following functionality requires migration from the example ASP.NET MVC project to the ASP.NET Core project:

- client-side content (CSS, fonts, and scripts)
- controllers
- views
- models
- bundling
- filters
- Log in/out, Identity (This is done in the next tutorial.)

Controllers and views

- Copy each of the methods from the ASP.NET MVC `HomeController` to the new `HomeController`. In ASP.NET MVC, the built-in template's controller action method return type is

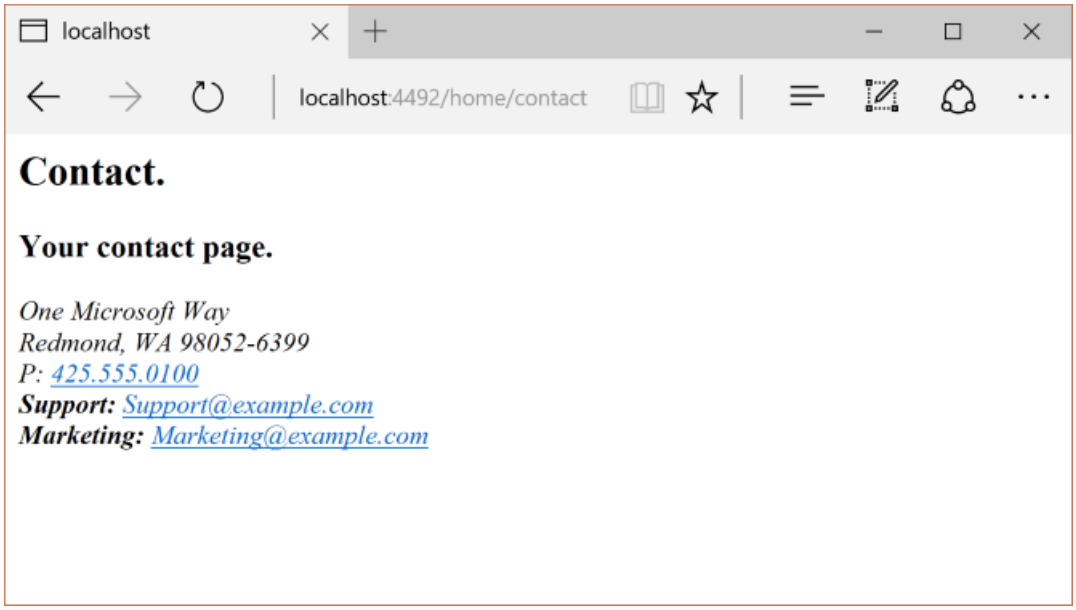
/dotnet/api/system.web.mvc.actionresult?view=aspnet-mvc-5.2; in ASP.NET Core MVC, the action methods return `IActionResult` instead. `ActionResult` implements `IActionResult`, so there's no need to change the return type of the action methods.

- Copy the *About.cshtml*, *Contact.cshtml*, and *Index.cshtml* Razor view files from the ASP.NET MVC project to the ASP.NET Core project.

Test each method

The layout file and styles have not been migrated yet, so the rendered views only contain the content in the view files. The layout file generated links for the `About` and `Contact` views will not be available yet.

- Invoke the rendered views from the browser on the running ASP.NET core app by replacing the current port number with the port number used in the ASP.NET core project. For example: `https://localhost:44375/home/about`.



Note the lack of styling and menu items. The styling will be fixed in the next section.

Static content

In ASP.NET MVC 5 and earlier, static content was hosted from the root of the web project and was intermixed with server-side files. In ASP.NET Core, static content is hosted in the *wwwroot* directory. Copy the static content from the ASP.NET MVC app to the *wwwroot* directory in the ASP.NET Core project. In this sample conversion:

- Copy the *favicon.ico* file from the ASP.NET MVC project to the *wwwroot* directory in the ASP.NET Core project.

The ASP.NET MVC project uses Bootstrap for its styling and stores the Bootstrap files in the *Content* and *Scripts* directories. The template, which generated the ASP.NET MVC project, references Bootstrap in the layout file (*Views/Shared/_Layout.cshtml*). The *bootstrap.js* and *bootstrap.css* files could be copied from the ASP.NET MVC project to the *wwwroot* directory in the new project. Instead, this document adds support for Bootstrap (and other client-side libraries) using CDNs, in the next section.

Migrate the layout file

- Copy the *_ViewStart.cshtml* file from the ASP.NET MVC project's *Views* directory into the ASP.NET Core project's *Views* directory. The *_ViewStart.cshtml* file has not changed in ASP.NET Core MVC.
- Create a *Views/Shared* directory.
- *Optional:* Copy *_ViewImports.cshtml* from the *FullAspNetCore* MVC project's *Views* directory into the

ASP.NET Core project's *Views* directory. Remove any namespace declaration in the *_ViewImports.cshtml* file. The *_ViewImports.cshtml* file provides namespaces for all the view files and brings in Tag Helpers. Tag Helpers are used in the new layout file. The *_ViewImports.cshtml* file is new for ASP.NET Core.

- Copy the *_Layout.cshtml* file from the ASP.NET MVC project's *Views/Shared* directory into the ASP.NET Core project's *Views/Shared* directory.

Open *_Layout.cshtml* file and make the following changes (the completed code is shown below):

- Replace `@Styles.Render("~/Content/css")` with a `<link>` element to load *bootstrap.css* (see below).
- Remove `@Scripts.Render("~/bundles/modernizr")`.
- Comment out the `@Html.Partial("_LoginPartial")` line (surround the line with `@*...*@`). For more information, see Migrate Authentication and Identity to ASP.NET Core
- Replace `@Scripts.Render("~/bundles/jquery")` with a `<script>` element (see below).
- Replace `@Scripts.Render("~/bundles/bootstrap")` with a `<script>` element (see below).

The replacement markup for Bootstrap CSS inclusion:

```
<link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
    integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    crossorigin="anonymous">
```

The replacement markup for jQuery and Bootstrap JavaScript inclusion:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
    integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
crossorigin="anonymous"></script>
```

The updated *_Layout.cshtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet"
          href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
          integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
          crossorigin="anonymous">
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-
collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class =
"navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @*@Html.Partial("_LoginPartial")*@
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    <script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
            integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"
            crossorigin="anonymous"></script>
    @RenderSection("scripts", required: false)
</body>
</html>
```

View the site in the browser. It should now load correctly, with the expected styles in place.

- *Optional:* Try using the new layout file. Copy the layout file from the *FullAspNetCore* project. The new layout file uses Tag Helpers and has other improvements.

Configure bundling and minification

For information about how to configure bundling and minification, see Bundling and Minification.

Solve HTTP 500 errors

There are many problems that can cause an HTTP 500 error messages that contain no information on the source of the problem. For example, if the *Views/_ViewImports.cshtml* file contains a namespace that doesn't exist in the

project, a HTTP 500 error is generated. By default in ASP.NET Core apps, the `UseDeveloperExceptionPage` extension is added to the `IApplicationBuilder` and executed when the configuration is *Development*. See an example in the following code:

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to the container.
    // For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?
LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseStaticFiles();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}
```

ASP.NET Core converts unhandled exceptions into HTTP 500 error responses. Normally, error details aren't included in these responses to prevent disclosure of potentially sensitive information about the server. For more information, see Developer Exception Page.

Additional resources

- Introduction to ASP.NET Core Blazor
- Tag Helpers in ASP.NET Core

Migrate from ASP.NET Web API to ASP.NET Core

9/22/2020 • 10 minutes to read • Edit Online

By Scott Addie and Steve Smith

An ASP.NET 4.x Web API is an HTTP service that reaches a broad range of clients, including browsers and mobile devices. ASP.NET Core combines ASP.NET 4.x's MVC and Web API app models into a single programming model known as ASP.NET Core MVC. This article demonstrates the steps required to migrate from ASP.NET 4.x Web API to ASP.NET Core MVC.

View or download sample code (how to download)

Prerequisites

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK or later

Review ASP.NET 4.x Web API project

This article uses the *ProductsApp* project created in Getting Started with ASP.NET Web API 2. In that project, a basic ASP.NET 4.x Web API project is configured as follows.

In *Global.asax.cs*, a call is made to `WebApiConfig.Register`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Routing;

namespace ProductsApp
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

The `WebApiConfig` class is found in the *App_Start* folder and has a static `Register` method:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace ProductsApp
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

The preceding class:

- Configures attribute routing, although it's not actually being used.
- Configures the routing table. The sample code expects URLs to match the format `/api/{controller}/{id}`, with `{id}` being optional.

The following sections demonstrate migration of the Web API project to ASP.NET Core MVC.

Create the destination project

Create a new blank solution in Visual Studio and add the ASP.NET 4.x Web API project to migrate:

1. From the **File** menu, select **New** > **Project**.
2. Select the **Blank Solution** template and select **Next**.
3. Name the solution *WebAPIMigration*. Select **Create**.
4. Add the existing *ProductsApp* project to the solution.

Add a new API project to migrate to:

1. Add a new **ASP.NET Core Web Application** project to the solution.
2. In the **Configure your new project** dialog, Name the project *ProductsCore*, and select **Create**.
3. In the **Create a new ASP.NET Core Web Application** dialog, confirm that **.NET Core** and **ASP.NET Core 3.1** are selected. Select the **API** project template, and select **Create**.
4. Remove the *WeatherForecast.cs* and *Controllers/WeatherForecastController.cs* example files from the new *ProductsCore* project.

The solution now contains two projects. The following sections explain migrating the *ProductsApp* project's contents to the *ProductsCore* project.

Migrate configuration

ASP.NET Core doesn't use the *App_Start* folder or the *Global.asax* file. Additionally, the *web.config* file is added at publish time.

The `Startup` class:

- Replaces *Global.asax*.
- Handles all app startup tasks.

For more information, see App startup in ASP.NET Core.

Migrate models and controllers

The following code shows the `ProductsController` to be updated for ASP.NET Core:

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product
            {
                Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1
            },
            new Product
            {
                Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M
            },
            new Product
            {
                Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M
            }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}
```

Update the `ProductsController` for ASP.NET Core:

1. Copy *Controllers/ProductsController.cs* and the *Models* folder from the original project to the new one.
2. Change the copied files' root namespace to `ProductsCore`.
3. Update the `using ProductsApp.Models;` statement to `using ProductsCore.Models;`.

The following components don't exist in ASP.NET Core:

- `ApiController` class
- `System.Web.Http` namespace
- `IHttpActionResult` interface

Make the following changes:

1. Change `ApiController` to ControllerBase. Add `using Microsoft.AspNetCore.Mvc;` to resolve the `ControllerBase` reference.
2. Delete `using System.Web.Http;`.
3. Change the `GetProduct` action's return type from `IHttpActionResult` to `ActionResult<Product>`.
4. Simplify the `GetProduct` action's `return` statement to the following:

```
return product;
```

Configure routing

The ASP.NET Core *API* project template includes endpoint routing configuration in the generated code.

The following UseRouting and UseEndpoints calls:

- Register route matching and endpoint execution in the middleware pipeline.
- Replace the *ProductsApp* project's *App_Start/WebApiConfig.cs* file.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Configure routing as follows:

1. Mark the `ProductsController` class with the following attributes:

   ```
   [Route("api/[controller]")]
   [ApiController]
   ```

   The preceding `[Route]` attribute configures the controller's attribute routing pattern. The `[ApiController]` attribute makes attribute routing a requirement for all actions in this controller.

   Attribute routing supports tokens, such as `[controller]` and `[action]`. At runtime, each token is replaced with the name of the controller or action, respectively, to which the attribute has been applied. The tokens:

- Reduce the number of magic strings in the project.
- Ensure routes remain synchronized with the corresponding controllers and actions when automatic rename refactorings are applied.

2. Enable HTTP Get requests to the `ProductController` actions:

   - Apply the `[HttpGet]` attribute to the `GetAllProducts` action.
   - Apply the `[HttpGet("{id}")]` attribute to the `GetProduct` action.

Run the migrated project, and browse to `/api/products`. A full list of three products appears. Browse to `/api/products/1`. The first product appears.

Additional resources

- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Compatibility version for ASP.NET Core MVC

Prerequisites

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> **WARNING**
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

Review ASP.NET 4.x Web API project

This article uses the *ProductsApp* project created in Getting Started with ASP.NET Web API 2. In that project, a basic ASP.NET 4.x Web API project is configured as follows.

In *Global.asax.cs*, a call is made to `WebApiConfig.Register`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Routing;

namespace ProductsApp
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

The `WebApiConfig` class is found in the *App_Start* folder and has a static `Register` method:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace ProductsApp
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Web API configuration and services

            // Web API routes
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

This class configures attribute routing, although it's not actually being used in the project. It also configures the routing table, which is used by ASP.NET Web API. In this case, ASP.NET 4.x Web API expects URLs to match the format `/api/{controller}/{id}`, with `{id}` being optional.

The following sections demonstrate migration of the Web API project to ASP.NET Core MVC.

Create the destination project

Complete the following steps in Visual Studio:

- Go to **File** > **New** > **Project** > **Other Project Types** > **Visual Studio Solutions**. Select **Blank Solution**, and name the solution *WebAPIMigration*. Click the **OK** button.
- Add the existing *ProductsApp* project to the solution.
- Add a new **ASP.NET Core Web Application** project to the solution. Select the **.NET Core** target framework from the drop-down, and select the **API** project template. Name the project *ProductsCore*, and click the **OK** button.

The solution now contains two projects. The following sections explain migrating the *ProductsApp* project's contents to the *ProductsCore* project.

Migrate configuration

ASP.NET Core doesn't use:

- *App_Start* folder or the *Global.asax* file
- *web.config* file is added at publish time.

The `Startup` class:

- Replaces *Global.asax*.
- Handles all app startup tasks.

For more information, see App startup in ASP.NET Core.

In ASP.NET Core MVC, attribute routing is included by default when UseMvc is called in `Startup.Configure`. The

following `UseMvc` call replaces the *ProductsApp* project's *App_Start/WebApiConfig.cs* file:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

Migrate models and controllers

The following code shows the `ProductsController` update for ASP.NET Core:

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product
            {
                Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1
            },
            new Product
            {
                Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M
            },
            new Product
            {
                Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M
            }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
    }
}
```

Update the `ProductsController` for ASP.NET Core:

1. Copy *Controllers/ProductsController.cs* from the original project to the new one.
2. Copy the *Models* folder from the original project to the new one.
3. Change the copied files' root namespace to `ProductsCore`.
4. Update the `using ProductsApp.Models;` statement to `using ProductsCore.Models;`.

The following components don't exist in ASP.NET Core:

- `ApiController` class
- `System.Web.Http` namespace
- `IHttpActionResult` interface

Make the following changes:

1. Change `ApiController` to ControllerBase. Add `using Microsoft.AspNetCore.Mvc;` to resolve the `ControllerBase` reference.
2. Delete `using System.Web.Http;`.

3. Change the `GetProduct` action's return type from `IHttpActionResult` to `ActionResult<Product>`.
4. Simplify the `GetProduct` action's `return` statement to the following:

```
return product;
```

Configure routing

Configure routing as follows:

1. Mark the `ProductsController` class with the following attributes:

```
[Route("api/[controller]")]
[ApiController]
```

   The preceding `[Route]` attribute configures the controller's attribute routing pattern. The `[ApiController]` attribute makes attribute routing a requirement for all actions in this controller.

   Attribute routing supports tokens, such as `[controller]` and `[action]`. At runtime, each token is replaced with the name of the controller or action, respectively, to which the attribute has been applied. The tokens reduce the number of magic strings in the project. The tokens also ensure routes remain synchronized with the corresponding controllers and actions when automatic rename refactorings are applied.

2. Set the project's compatibility mode to ASP.NET Core 2.2:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

   The preceding change:

   - Is required to use the `[ApiController]` attribute at the controller level.
   - Opts in to potentially breaking behaviors introduced in ASP.NET Core 2.2.

3. Enable HTTP Get requests to the `ProductController` actions:

   - Apply the `[HttpGet]` attribute to the `GetAllProducts` action.
   - Apply the `[HttpGet("{id}")]` attribute to the `GetProduct` action.

Run the migrated project, and browse to `/api/products`. A full list of three products appears. Browse to `/api/products/1`. The first product appears.

Compatibility shim

The Microsoft.AspNetCore.Mvc.WebApiCompatShim library provides a compatibility shim to move ASP.NET 4.x Web API projects to ASP.NET Core. The compatibility shim extends ASP.NET Core to support a number of conventions from ASP.NET 4.x Web API 2. The sample ported previously in this document is basic enough that the compatibility shim was unnecessary. For larger projects, using the compatibility shim can be useful for temporarily bridging the API gap between ASP.NET Core and ASP.NET 4.x Web API 2.

The Web API compatibility shim is meant to be used as a temporary measure to support migrating large ASP.NET 4.x Web API projects to ASP.NET Core. Over time, projects should be updated to use ASP.NET Core patterns instead of relying on the compatibility shim.

Compatibility features included in `Microsoft.AspNetCore.Mvc.WebApiCompatShim` include:

- Adds an `ApiController` type so that controllers' base types don't need to be updated.
- Enables Web API-style model binding. ASP.NET Core MVC model binding functions similarly to that of ASP.NET 4.x MVC 5, by default. The compatibility shim changes model binding to be more similar to ASP.NET 4.x Web API 2 model binding conventions. For example, complex types are automatically bound from the request body.
- Extends model binding so that controller actions can take parameters of type `HttpRequestMessage`.
- Adds message formatters allowing actions to return results of type `HttpResponseMessage`.
- Adds additional response methods that Web API 2 actions may have used to serve responses:
  - `HttpResponseMessage` generators:
    - `CreateResponse<T>`
    - `CreateErrorResponse`
  - Action result methods:
    - `BadRequestErrorMessageResult`
    - `ExceptionResult`
    - `InternalServerErrorResult`
    - `InvalidModelStateResult`
    - `NegotiatedContentResult`
    - `ResponseMessageResult`
- Adds an instance of `IContentNegotiator` to the app's dependency injection (DI) container and makes available the content negotiation-related types from Microsoft.AspNet.WebApi.Client. Examples of such types include `DefaultContentNegotiator` and `MediaTypeFormatter`.

To use the compatibility shim:

1. Install the Microsoft.AspNetCore.Mvc.WebApiCompatShim NuGet package.
2. Register the compatibility shim's services with the app's DI container by calling `services.AddMvc().AddWebApiConventions()` in `Startup.ConfigureServices`.
3. Define web API-specific routes using `MapWebApiRoute` on the `IRouteBuilder` in the app's `IApplicationBuilder.UseMvc` call.

Additional resources

- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Compatibility version for ASP.NET Core MVC

Migrate configuration to ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Steve Smith and Scott Addie

In the previous article, we began to migrate an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate configuration.

View or download sample code (how to download)

Setup configuration

ASP.NET Core no longer uses the *Global.asax* and *web.config* files that previous versions of ASP.NET utilized. In the earlier versions of ASP.NET, application startup logic was placed in an `Application_StartUp` method within *Global.asax*. Later, in ASP.NET MVC, a *Startup.cs* file was included in the root of the project; and, it was called when the application started. ASP.NET Core has adopted this approach completely by placing all startup logic in the *Startup.cs* file.

The *web.config* file has also been replaced in ASP.NET Core. Configuration itself can now be configured, as part of the application startup procedure described in *Startup.cs*. Configuration can still utilize XML files, but typically ASP.NET Core projects will place configuration values in a JSON-formatted file, such as *appsettings.json*. ASP.NET Core's configuration system can also easily access environment variables, which can provide a more secure and robust location for environment-specific values. This is especially true for secrets like connection strings and API keys that shouldn't be checked into source control. See Configuration to learn more about configuration in ASP.NET Core.

For this article, we are starting with the partially migrated ASP.NET Core project from the previous article. To setup configuration, add the following constructor and property to the *Startup.cs* file located in the root of the project:

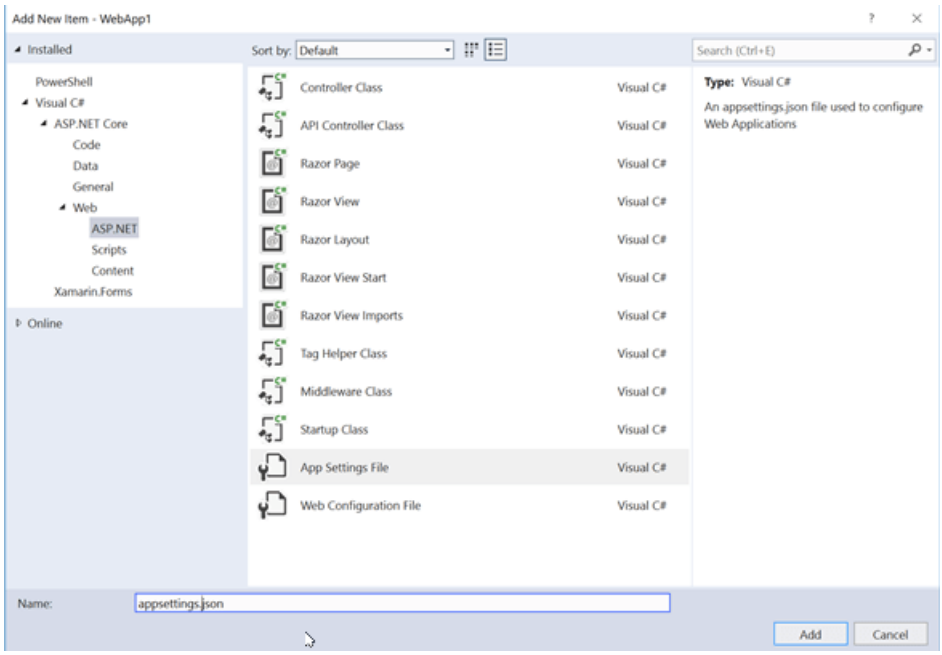
```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

Note that at this point, the *Startup.cs* file won't compile, as we still need to add the following `using` statement:

```
using Microsoft.Extensions.Configuration;
```

Add an *appsettings.json* file to the root of the project using the appropriate item template:



Migrate configuration settings from web.config

Our ASP.NET MVC project included the required database connection string in *web.config*, in the `<connectionStrings>` element. In our ASP.NET Core project, we are going to store this information in the *appsettings.json* file. Open *appsettings.json*, and note that it already includes the following:

```
{
    "Data": {
        "DefaultConnection": {
            "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trusted_Connection=True;"
        }
    }
}
```

In the highlighted line depicted above, change the name of the database from **_CHANGE_ME** to the name of your database.

Summary

ASP.NET Core places all startup logic for the application in a single file, in which the necessary services and dependencies can be defined and configured. It replaces the *web.config* file with a flexible configuration feature that can leverage a variety of file formats, such as JSON, as well as environment variables.

Migrate Authentication and Identity to ASP.NET Core

9/22/2020 • 2 minutes to read • Edit Online

By Steve Smith

In the previous article, we migrated configuration from an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate the registration, login, and user management features.

Configure Identity and Membership

In ASP.NET MVC, authentication and identity features are configured using ASP.NET Identity in *Startup.Auth.cs* and *IdentityConfig.cs*, located in the *App_Start* folder. In ASP.NET Core MVC, these features are configured in *Startup.cs*.

Install the following NuGet packages:

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- `Microsoft.AspNetCore.Authentication.Cookies`
- `Microsoft.EntityFrameworkCore.SqlServer`

In *Startup.cs*, update the `Startup.ConfigureServices` method to use Entity Framework and Identity services:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

     services.AddMvc();
}
```

At this point, there are two types referenced in the above code that we haven't yet migrated from the ASP.NET MVC project: `ApplicationDbContext` and `ApplicationUser`. Create a new *Models* folder in the ASP.NET Core project, and add two classes to it corresponding to these types. You will find the ASP.NET MVC versions of these classes in */Models/IdentityModels.cs*, but we will use one file per class in the migrated project since that's more clear.

ApplicationUser.cs:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace NewMvcProject.Models
{
  public class ApplicationUser : IdentityUser
  {
  }
}
```

ApplicationDbContext.cs:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.Data.Entity;

namespace NewMvcProject.Models
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            // Customize the ASP.NET Core Identity model and override the defaults if needed.
            // For example, you can rename the ASP.NET Core Identity table names and more.
            // Add your customizations after calling base.OnModelCreating(builder);
        }
    }
}
```

The ASP.NET Core MVC Starter Web project doesn't include much customization of users, or the `ApplicationDbContext`. When migrating a real app, you also need to migrate all of the custom properties and methods of your app's user and `DbContext` classes, as well as any other Model classes your app utilizes. For example, if your `DbContext` has a `DbSet<Album>`, you need to migrate the `Album` class.

With these files in place, the *Startup.cs* file can be made to compile by updating its `using` statements:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

Our app is now ready to support authentication and Identity services. It just needs to have these features exposed to users.

Migrate registration and login logic

With Identity services configured for the app and data access configured using Entity Framework and SQL Server, we're ready to add support for registration and login to the app. Recall that earlier in the migration process we commented out a reference to *_LoginPartial* in *_Layout.cshtml*. Now it's time to return to that code, uncomment it, and add in the necessary controllers and views to support login functionality.

Uncomment the `@Html.Partial` line in *_Layout.cshtml*:

```
      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
    @*@Html.Partial("_LoginPartial")*@
  </div>
</div>
```

Now, add a new Razor view called *_LoginPartial* to the *Views/Shared* folder:

Update *_LoginPartial.cshtml* with the following code (replace all of its contents):

```
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Account" asp-action="Logout" method="post" id="logoutForm"
class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="" asp-controller="Manage" asp-action="Index" title="Manage">Hello
@UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="" asp-controller="Account" asp-action="Register">Register</a></li>
        <li><a asp-area="" asp-controller="Account" asp-action="Login">Log in</a></li>
    </ul>
}
```

At this point, you should be able to refresh the site in your browser.

Summary

ASP.NET Core introduces changes to the ASP.NET Identity features. In this article, you have seen how to migrate the authentication and user management features of ASP.NET Identity to ASP.NET Core.

Migrate from ClaimsPrincipal.Current

9/22/2020 • 2 minutes to read • Edit Online

In ASP.NET 4.x projects, it was common to use ClaimsPrincipal.Current to retrieve the current authenticated user's identity and claims. In ASP.NET Core, this property is no longer set. Code that was depending on it needs to be updated to get the current authenticated user's identity through a different means.

Context-specific state instead of static state

When using ASP.NET Core, the values of both `ClaimsPrincipal.Current` and `Thread.CurrentPrincipal` aren't set. These properties both represent static state, which ASP.NET Core generally avoids. Instead, ASP.NET Core uses dependency injection (DI) to provide dependencies such as the current user's identity. Getting the current user's identity from DI is more testable, too, since test identities can be easily injected.

Retrieve the current user in an ASP.NET Core app

There are several options for retrieving the current authenticated user's `ClaimsPrincipal` in ASP.NET Core in place of `ClaimsPrincipal.Current`:

- **ControllerBase.User**. MVC controllers can access the current authenticated user with their User property.
- **HttpContext.User**. Components with access to the current `HttpContext` (middleware, for example) can get the current user's `ClaimsPrincipal` from HttpContext.User.
- **Passed in from caller**. Libraries without access to the current `HttpContext` are often called from controllers or middleware components and can have the current user's identity passed as an argument.
- **IHttpContextAccessor**. The project being migrated to ASP.NET Core may be too large to easily pass the current user's identity to all necessary locations. In such cases, IHttpContextAccessor can be used as a workaround. `IHttpContextAccessor` is able to access the current `HttpContext` (if one exists). If DI is being used, see Access HttpContext in ASP.NET Core. A short-term solution to getting the current user's identity in code that hasn't yet been updated to work with ASP.NET Core's DI-driven architecture would be:
  - Make `IHttpContextAccessor` available in the DI container by calling AddHttpContextAccessor in `Startup.ConfigureServices`.
  - Get an instance of `IHttpContextAccessor` during startup and store it in a static variable. The instance is made available to code that was previously retrieving the current user from a static property.
  - Retrieve the current user's `ClaimsPrincipal` using `HttpContextAccessor.HttpContext?.User`. If this code is used outside of the context of an HTTP request, the `HttpContext` is null.

The final option, using an `IHttpContextAccessor` instance stored in a static variable, is contrary to the ASP.NET Core principle of preferring injected dependencies to static dependencies. Plan to eventually retrieve `IHttpContextAccessor` instances from DI instead. A static helper can be a useful bridge, though, when migrating large existing ASP.NET apps that use `ClaimsPrincipal.Current`.

Migrate from ASP.NET Membership authentication to ASP.NET Core 2.0 Identity

9/22/2020 • 6 minutes to read • Edit Online

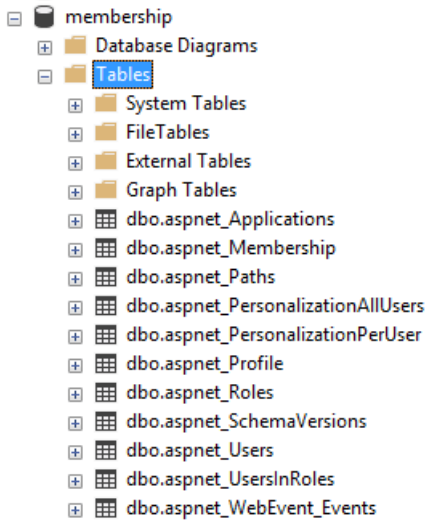
By Isaac Levin

This article demonstrates migrating the database schema for ASP.NET apps using Membership authentication to ASP.NET Core 2.0 Identity.

> **NOTE**
> This document provides the steps needed to migrate the database schema for ASP.NET Membership-based apps to the database schema used for ASP.NET Core Identity. For more information about migrating from ASP.NET Membership-based authentication to ASP.NET Identity, see Migrate an existing app from SQL Membership to ASP.NET Identity. For more information about ASP.NET Core Identity, see Introduction to Identity on ASP.NET Core.

Review of Membership schema

Prior to ASP.NET 2.0, developers were tasked with creating the entire authentication and authorization process for their apps. With ASP.NET 2.0, Membership was introduced, providing a boilerplate solution to handling security within ASP.NET apps. Developers were now able to bootstrap a schema into a SQL Server database with the /previous-versions/ms229862(v=vs.140) command. After running this command, the following tables were created in the database.



To migrate existing apps to ASP.NET Core 2.0 Identity, the data in these tables needs to be migrated to the tables used by the new Identity schema.

ASP.NET Core Identity 2.0 schema

ASP.NET Core 2.0 follows the Identity principle introduced in ASP.NET 4.5. Though the principle is shared, the implementation between the frameworks is different, even between versions of ASP.NET Core (see Migrate authentication and Identity to ASP.NET Core 2.0).

The fastest way to view the schema for ASP.NET Core 2.0 Identity is to create a new ASP.NET Core 2.0 app. Follow these steps in Visual Studio 2017:

1. Select **File** > **New** > **Project**.
2. Create a new **ASP.NET Core Web Application** project named *CoreIdentitySample*.
3. Select **ASP.NET Core 2.0** in the dropdown and then select **Web Application**. This template produces a Razor Pages app. Before clicking **OK**, click **Change Authentication**.
4. Choose **Individual User Accounts** for the Identity templates. Finally, click **OK**, then **OK**. Visual Studio creates a project using the ASP.NET Core Identity template.
5. Select **Tools** > **NuGet Package Manager** > **Package Manager Console** to open the **Package Manager Console** (PMC) window.
6. Navigate to the project root in PMC, and run the Entity Framework (EF) Core `Update-Database` command.

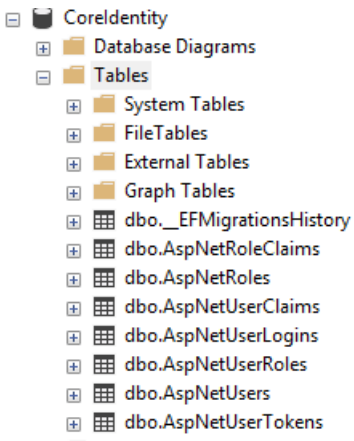
   ASP.NET Core 2.0 Identity uses EF Core to interact with the database storing the authentication data. In order for the newly created app to work, there needs to be a database to store this data. After creating a new app, the fastest way to inspect the schema in a database environment is to create the database using EF Core Migrations. This process creates a database, either locally or elsewhere, which mimics that schema. Review the preceding documentation for more information.

   EF Core commands use the connection string for the database specified in *appsettings.json*. The following connection string targets a database on *localhost* named *asp-net-core-identity*. In this setting, EF Core is configured to use the `DefaultConnection` connection string.

   ```
   {
     "ConnectionStrings": {
       "DefaultConnection": "Server=localhost;Database=aspnet-core-
   identity;Trusted_Connection=True;MultipleActiveResultSets=true"
     }
   }
   ```

7. Select **View** > **SQL Server Object Explorer**. Expand the node corresponding to the database name specified in the `ConnectionStrings:DefaultConnection` property of *appsettings.json*.

   The `Update-Database` command created the database specified with the schema and any data needed for app initialization. The following image depicts the table structure that's created with the preceding steps.



Migrate the schema

There are subtle differences in the table structures and fields for both Membership and ASP.NET Core Identity. The pattern has changed substantially for authentication/authorization with ASP.NET and ASP.NET Core apps. The key objects that are still used with Identity are *Users* and *Roles*. Here are mapping tables for *Users*, *Roles*, and *UserRoles*.

Users

IDENTITY (`dbo.AspNetUsers`) COLUMN	TYPE	MEMBERSHIP (`dbo.aspnet_Users` / `dbo.aspnet_Membership`) COLUMN	TYPE
`Id`	`string`	`aspnet_Users.UserId`	`string`
`UserName`	`string`	`aspnet_Users.UserName`	`string`
`Email`	`string`	`aspnet_Membership.Email`	`string`
`NormalizedUserName`	`string`	`aspnet_Users.LoweredUserName`	`string`
`NormalizedEmail`	`string`	`aspnet_Membership.LoweredEmai`	`string`
`PhoneNumber`	`string`	`aspnet_Users.MobileAlias`	`string`
`LockoutEnabled`	`bit`	`aspnet_Membership.IsLockedOut`	`bit`

> **NOTE**
>
> Not all the field mappings resemble one-to-one relationships from Membership to ASP.NET Core Identity. The preceding table takes the default Membership User schema and maps it to the ASP.NET Core Identity schema. Any other custom fields that were used for Membership need to be mapped manually. In this mapping, there's no map for passwords, as both password criteria and password salts don't migrate between the two. **It's recommended to leave the password as null and to ask users to reset their passwords.** In ASP.NET Core Identity, `LockoutEnd` should be set to some date in the future if the user is locked out. This is shown in the migration script.

Roles

IDENTITY (`dbo.AspNetRoles`) COLUMN	TYPE	MEMBERSHIP (`dbo.aspnet_Roles`) COLUMN	TYPE
`Id`	`string`	`RoleId`	`string`
`Name`	`string`	`RoleName`	`string`
`NormalizedName`	`string`	`LoweredRoleName`	`string`

User Roles

IDENTITY (`dbo.AspNetUserRoles`) COLUMN	TYPE	MEMBERSHIP (`dbo.aspnet_UsersInRoles`) COLUMN	TYPE
`RoleId`	`string`	`RoleId`	`string`
`UserId`	`string`	`UserId`	`string`

Reference the preceding mapping tables when creating a migration script for *Users* and *Roles*. The following example assumes you have two databases on a database server. One database contains the existing ASP.NET Membership schema and data. The other *CoreIdentitySample* database was created using steps described earlier.

Comments are included inline for more details.

```
-- THIS SCRIPT NEEDS TO RUN FROM THE CONTEXT OF THE MEMBERSHIP DB
BEGIN TRANSACTION MigrateUsersAndRoles
USE aspnetdb

-- INSERT USERS
INSERT INTO CoreIdentitySample.dbo.AspNetUsers
            (Id,
             UserName,
             NormalizedUserName,
             PasswordHash,
             SecurityStamp,
             EmailConfirmed,
             PhoneNumber,
             PhoneNumberConfirmed,
             TwoFactorEnabled,
             LockoutEnd,
             LockoutEnabled,
             AccessFailedCount,
             Email,
             NormalizedEmail)
SELECT aspnet_Users.UserId,
       aspnet_Users.UserName,
       -- The NormalizedUserName value is upper case in ASP.NET Core Identity
       UPPER(aspnet_Users.UserName),
       -- Creates an empty password since passwords don't map between the 2 schemas
       '',
       /*
        The SecurityStamp token is used to verify the state of an account and
        is subject to change at any time. It should be initialized as a new ID.
       */
       NewID(),
       /*
        EmailConfirmed is set when a new user is created and confirmed via email.
        Users must have this set during migration to reset passwords.
       */
       1,
       aspnet_Users.MobileAlias,
       CASE
         WHEN aspnet_Users.MobileAlias IS NULL THEN 0
         ELSE 1
       END,
       -- 2FA likely wasn't setup in Membership for users, so setting as false.
       0,
       CASE
         -- Setting lockout date to time in the future (1,000 years)
         WHEN aspnet_Membership.IsLockedOut = 1 THEN Dateadd(year, 1000,
                                                     Sysutcdatetime())
         ELSE NULL
       END,
       aspnet_Membership.IsLockedOut,
       /*
        AccessFailedAccount is used to track failed logins. This is stored in
        Membership in multiple columns. Setting to 0 arbitrarily.
       */
       0,
       aspnet_Membership.Email,
       -- The NormalizedEmail value is upper case in ASP.NET Core Identity
       UPPER(aspnet_Membership.Email)
FROM   aspnet_Users
       LEFT OUTER JOIN aspnet_Membership
                    ON aspnet_Membership.ApplicationId =
                       aspnet_Users.ApplicationId
                       AND aspnet_Users.UserId = aspnet_Membership.UserId
       LEFT OUTER JOIN CoreIdentitySample.dbo.AspNetUsers
                    ON aspnet_Membership.UserId = AspNetUsers.Id
WHERE  AspNetUsers.Id IS NULL
```

```
-- INSERT ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetRoles(Id, Name)
SELECT RoleId, RoleName
FROM aspnet_Roles;

-- INSERT USER ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetUserRoles(UserId, RoleId)
SELECT UserId, RoleId
FROM aspnet_UsersInRoles;

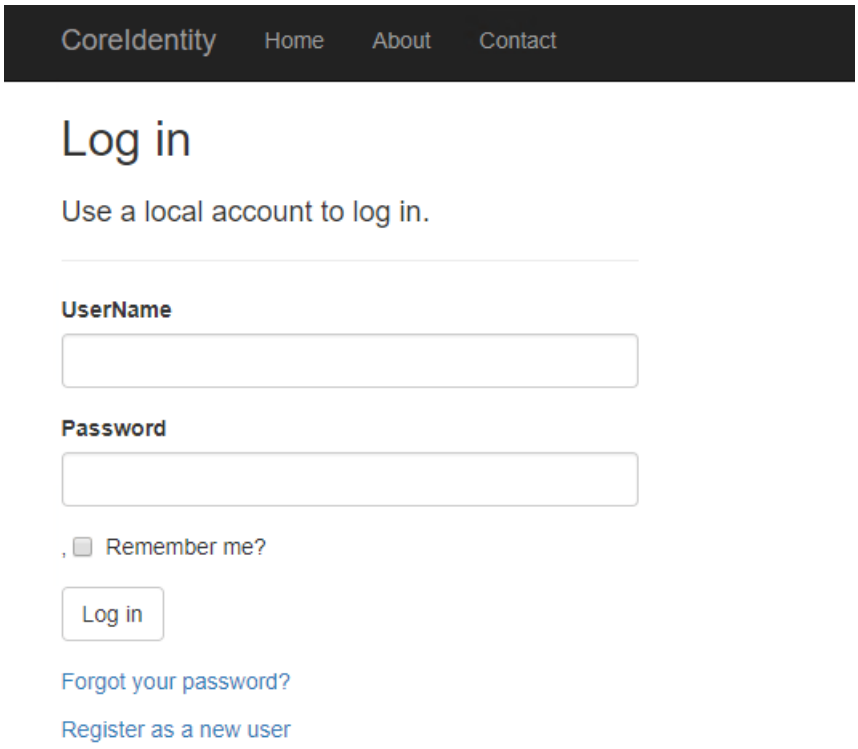
IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRANSACTION MigrateUsersAndRoles
    RETURN
  END

COMMIT TRANSACTION MigrateUsersAndRoles
```

After completion of the preceding script, the ASP.NET Core Identity app created earlier is populated with Membership users. Users need to change their passwords before logging in.

> **NOTE**
> If the Membership system had users with user names that didn't match their email address, changes are required to the app created earlier to accommodate this. The default template expects `UserName` and `Email` to be the same. For situations in which they're different, the login process needs to be modified to use `UserName` instead of `Email`.

In the `PageModel` of the Login Page, located at *Pages\Account\Login.cshtml.cs*, remove the `[EmailAddress]` attribute from the *Email* property. Rename it to *UserName*. This requires a change wherever `EmailAddress` is mentioned, in the *View* and *PageModel*. The result looks like the following:



Next steps

In this tutorial, you learned how to port users from SQL membership to ASP.NET Core 2.0 Identity. For more

information regarding ASP.NET Core Identity, see Introduction to Identity.

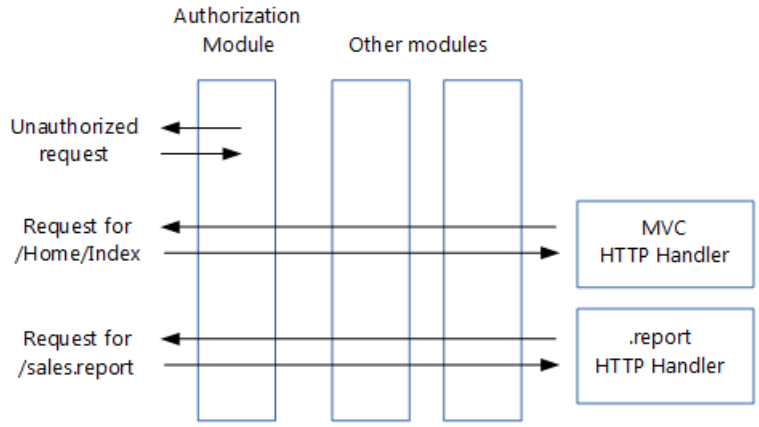
Migrate HTTP handlers and modules to ASP.NET Core middleware

9/22/2020 • 16 minutes to read • Edit Online

This article shows how to migrate existing ASP.NET HTTP modules and handlers from system.webserver to ASP.NET Core middleware.

Modules and handlers revisited

Before proceeding to ASP.NET Core middleware, let's first recap how HTTP modules and handlers work:



Handlers are:

- Classes that implement IHttpHandler
- Used to handle requests with a given file name or extension, such as *.report*
- Configured in *Web.config*

Modules are:

- Classes that implement IHttpModule
- Invoked for every request
- Able to short-circuit (stop further processing of a request)
- Able to add to the HTTP response, or create their own
- Configured in *Web.config*

The order in which modules process incoming requests is determined by:

1. The /previous-versions/ms227673(v=vs.140), which is a series events fired by ASP.NET: BeginRequest, AuthenticateRequest, etc. Each module can create a handler for one or more events.
2. For the same event, the order in which they're configured in *Web.config*.

In addition to modules, you can add handlers for the life cycle events to your *Global.asax.cs* file. These handlers run after the handlers in the configured modules.

From handlers and modules to middleware

Middleware are simpler than HTTP modules and handlers:

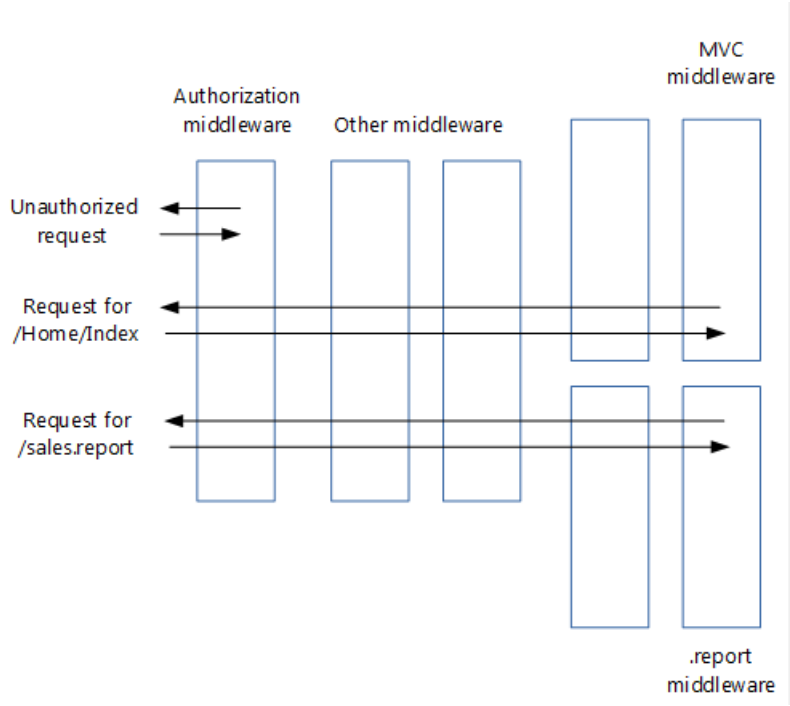
- Modules, handlers, *Global.asax.cs*, *Web.config* (except for IIS configuration) and the application life cycle are gone
- The roles of both modules and handlers have been taken over by middleware
- Middleware are configured using code rather than in *Web.config*
- Pipeline branching lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.
- Pipeline branching lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.

Middleware are very similar to modules:

- Invoked in principle for every request
- Able to short-circuit a request, by not passing the request to the next middleware
- Able to create their own HTTP response

Middleware and modules are processed in a different order:

- Order of middleware is based on the order in which they're inserted into the request pipeline, while order of modules is mainly based on /previous-versions/ms227673(v=vs.140) events
- Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses
- See Create a middleware pipeline with IApplicationBuilder



Note how in the image above, the authentication middleware short-circuited the request.

Migrating module code to middleware

An existing HTTP module will look similar to this:

```
// ASP.NET 4 module

using System;
using System.Web;

namespace MyApp.Modules
{
    public class MyModule : IHttpModule
    {
        public void Dispose()
        {
        }

        public void Init(HttpApplication application)
        {
            application.BeginRequest += (new EventHandler(this.Application_BeginRequest));
            application.EndRequest += (new EventHandler(this.Application_EndRequest));
        }

        private void Application_BeginRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the beginning of request processing.
        }

        private void Application_EndRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the end of request processing.
        }
    }
}
```

As shown in the Middleware page, an ASP.NET Core middleware is a class that exposes an `Invoke` method taking an `HttpContext` and returning a `Task`. Your new middleware will look like this:

```
// ASP.NET Core middleware

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyMiddleware
    {
        private readonly RequestDelegate _next;

        public MyMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            // Do something with context near the beginning of request processing.

            await _next.Invoke(context);

            // Clean up.
        }
    }

    public static class MyMiddlewareExtensions
    {
        public static IApplicationBuilder UseMyMiddleware(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyMiddleware>();
        }
    }
}
```

The preceding middleware template was taken from the section on writing middleware.

The *MyMiddlewareExtensions* helper class makes it easier to configure your middleware in your `Startup` class. The `UseMyMiddleware` method adds your middleware class to the request pipeline. Services required by the middleware get injected in the middleware's constructor.

Your module might terminate a request, for example if the user isn't authorized:

```
// ASP.NET 4 module that may terminate the request

private void Application_BeginRequest(Object source, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;

    // Do something with context near the beginning of request processing.

    if (TerminateRequest())
    {
        context.Response.End();
        return;
    }
}
```

A middleware handles this by not calling `Invoke` on the next middleware in the pipeline. Keep in mind that this doesn't fully terminate the request, because previous middlewares will still be invoked when the response makes its way back through the pipeline.

```
// ASP.NET Core middleware that may terminate the request

public async Task Invoke(HttpContext context)
{
    // Do something with context near the beginning of request processing.

    if (!TerminateRequest())
        await _next.Invoke(context);

    // Clean up.
}
```

When you migrate your module's functionality to your new middleware, you may find that your code doesn't compile because the `HttpContext` class has significantly changed in ASP.NET Core. Later on, you'll see how to migrate to the new ASP.NET Core HttpContext.

Migrating module insertion into the request pipeline

HTTP modules are typically added to the request pipeline using *Web.config*.

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <modules>
      <add name="MyModule" type="MyApp.Modules.MyModule"/>
    </modules>
  </system.webServer>
</configuration>
```

Convert this by adding your new middleware to the request pipeline in your `Startup` class:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

The exact spot in the pipeline where you insert your new middleware depends on the event that it handled as a module (`BeginRequest` , `EndRequest` , etc.) and its order in your list of modules in *Web.config*.

As previously stated, there's no application life cycle in ASP.NET Core and the order in which responses are processed by middleware differs from the order used by modules. This could make your ordering decision more challenging.

If ordering becomes a problem, you could split your module into multiple middleware components that can be ordered independently.

Migrating handler code to middleware

An HTTP handler looks something like this:

```
// ASP.NET 4 handler

using System.Web;

namespace MyApp.HttpHandlers
{
    public class MyHandler : IHttpHandler
    {
        public bool IsReusable { get { return true; } }

        public void ProcessRequest(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            context.Response.Output.Write(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.QueryString["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }
}
```

In your ASP.NET Core project, you would translate this to a middleware similar to this:

```
// ASP.NET Core middleware migrated from a handler

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyHandlerMiddleware
    {

        // Must have constructor with this signature, otherwise exception at run time
        public MyHandlerMiddleware(RequestDelegate next)
        {
            // This is an HTTP Handler, so no need to store next
        }

        public async Task Invoke(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            await context.Response.WriteAsync(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.Query["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }

    public static class MyHandlerExtensions
    {
        public static IApplicationBuilder UseMyHandler(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyHandlerMiddleware>();
        }
    }
}
```

This middleware is very similar to the middleware corresponding to modules. The only real difference is that here there's no call to `_next.Invoke(context)`. That makes sense, because the handler is at the end of the request pipeline, so there will be no next middleware to invoke.

Migrating handler insertion into the request pipeline

Configuring an HTTP handler is done in *Web.config* and looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandler" verb="*" path="*.report" type="MyApp.HttpHandlers.MyHandler"
resourceType="Unspecified" preCondition="integratedMode"/>
    </handlers>
  </system.webServer>
</configuration>
```

You could convert this by adding your new handler middleware to the request pipeline in your `Startup` class, similar to middleware converted from modules. The problem with that approach is that it would send all requests to your new handler middleware. However, you only want requests with a given extension to reach your middleware. That would give you the same functionality you had with your HTTP handler.

One solution is to branch the pipeline for requests with a given extension, using the `MapWhen` extension method. You do this in the same `Configure` method where you add the other middleware:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions = Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>
();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

`MapWhen` takes these parameters:

1. A lambda that takes the `HttpContext` and returns `true` if the request should go down the branch. This means you can branch requests not just based on their extension, but also on request headers, query string parameters, etc.
2. A lambda that takes an `IApplicationBuilder` and adds all the middleware for the branch. This means you can add additional middleware to the branch in front of your handler middleware.

Middleware added to the pipeline before the branch will be invoked on all requests; the branch will have no impact on them.

Loading middleware options using the options pattern

Some modules and handlers have configuration options that are stored in *Web.config*. However, in ASP.NET Core a new configuration model is used in place of *Web.config*.

The new configuration system gives you these options to solve this:

- Directly inject the options into the middleware, as shown in the next section.
- Use the options pattern:

1. Create a class to hold your middleware options, for example:

```
public class MyMiddlewareOptions
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}
```

2. Store the option values

   The configuration system allows you to store option values anywhere you want. However, most sites use *appsettings.json*, so we'll take that approach:

```
{
  "MyMiddlewareOptionsSection": {
    "Param1": "Param1Value",
    "Param2": "Param2Value"
  }
}
```

   MyMiddlewareOptionsSection here is a section name. It doesn't have to be the same as the name of your options class.

3. Associate the option values with the options class

   The options pattern uses ASP.NET Core's dependency injection framework to associate the options type (such as `MyMiddlewareOptions`) with a `MyMiddlewareOptions` object that has the actual options.

   Update your `Startup` class:

   a. If you're using *appsettings.json*, add it to the configuration builder in the `Startup` constructor:

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

   b. Configure the options service:

```
public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}
```

c. Associate your options with your options class:

```
public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}
```

4. Inject the options into your middleware constructor. This is similar to injecting options into a controller.

```
public class MyMiddlewareWithParams
{
    private readonly RequestDelegate _next;
    private readonly MyMiddlewareOptions _myMiddlewareOptions;

    public MyMiddlewareWithParams(RequestDelegate next,
        IOptions<MyMiddlewareOptions> optionsAccessor)
    {
        _next = next;
        _myMiddlewareOptions = optionsAccessor.Value;
    }

    public async Task Invoke(HttpContext context)
    {
        // Do something with context near the beginning of request processing
        // using configuration in _myMiddlewareOptions

        await _next.Invoke(context);

        // Do something with context near the end of request processing
        // using configuration in _myMiddlewareOptions
    }
}
```

The UseMiddleware extension method that adds your middleware to the `IApplicationBuilder` takes care of dependency injection.

This isn't limited to `IOptions` objects. Any other object that your middleware requires can be injected this way.

Loading middleware options through direct injection

The options pattern has the advantage that it creates loose coupling between options values and their consumers. Once you've associated an options class with the actual options values, any other class can get access to the options through the dependency injection framework. There's no need to pass around options values.

This breaks down though if you want to use the same middleware twice, with different options. For example an authorization middleware used in different branches allowing different roles. You can't associate two different options objects with the one options class.

The solution is to get the options objects with the actual options values in your `Startup` class and pass those directly to each instance of your middleware.

1. Add a second key to *appsettings.json*

   To add a second set of options to the *appsettings.json* file, use a new key to uniquely identify it:

   ```
   {
     "MyMiddlewareOptionsSection2": {
       "Param1": "Param1Value2",
       "Param2": "Param2Value2"
     },
     "MyMiddlewareOptionsSection": {
       "Param1": "Param1Value",
       "Param2": "Param2Value"
     }
   }
   ```

2. Retrieve options values and pass them to middleware. The `Use...` extension method (which adds your middleware to the pipeline) is a logical place to pass in the option values:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();

    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptions>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

3. Enable middleware to take an options parameter. Provide an overload of the `Use...` extension method (that takes the options parameter and passes it to `UseMiddleware`). When `UseMiddleware` is called with parameters, it passes the parameters to your middleware constructor when it instantiates the middleware object.

```
public static class MyMiddlewareWithParamsExtensions
{
    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>();
    }

    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder, MyMiddlewareOptions myMiddlewareOptions)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>(
            new OptionsWrapper<MyMiddlewareOptions>(myMiddlewareOptions));
    }
}
```

Note how this wraps the options object in an `OptionsWrapper` object. This implements `IOptions`, as expected by the middleware constructor.

Migrating to the new HttpContext

You saw earlier that the `Invoke` method in your middleware takes a parameter of type `HttpContext`:

```
public async Task Invoke(HttpContext context)
```

`HttpContext` has significantly changed in ASP.NET Core. This section shows how to translate the most commonly used properties of System.Web.HttpContext to the new `Microsoft.AspNetCore.Http.HttpContext`.

HttpContext

HttpContext.Items translates to:

```
IDictionary<object, object> items = httpContext.Items;
```

Unique request ID (no System.Web.HttpContext counterpart)

Gives you a unique id for each request. Very useful to include in your logs.

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

HttpContext.Request.HttpMethod translates to:

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString translates to:

```
IQueryCollection queryParameters = httpContext.Request.Query;

// If no query parameter "key" used, values will have 0 items
// If single value used for a key (...?key=v1), values will have 1 item ("v1")
// If key has multiple values (...?key=v1&key=v2), values will have 2 items ("v1" and "v2")
IList<string> values = queryParameters["key"];

// If no query parameter "key" used, value will be ""
// If single value used for a key (...?key=v1), value will be "v1"
// If key has multiple values (...?key=v1&key=v2), value will be "v1,v2"
string value = queryParameters["key"].ToString();
```

HttpContext.Request.Url and **HttpContext.Request.RawUrl** translate to:

```
// using Microsoft.AspNetCore.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();
```

HttpContext.Request.IsSecureConnection translates to:

```
var isSecureConnection = httpContext.Request.IsHttps;
```

HttpContext.Request.UserHostAddress translates to:

```
var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();
```

HttpContext.Request.Cookies translates to:

```
IRequestCookieCollection cookies = httpContext.Request.Cookies;
string unknownCookieValue = cookies["unknownCookie"]; // will be null (no exception)
string knownCookieValue = cookies["cookie1name"];     // will be actual value
```

HttpContext.Request.RequestContext.RouteData translates to:

```
var routeValue = httpContext.GetRouteValue("key");
```

HttpContext.Request.Headers translates to:

```
// using Microsoft.AspNetCore.Http.Headers;
// using Microsoft.Net.Http.Headers;

IHeaderDictionary headersDictionary = httpContext.Request.Headers;

// GetTypedHeaders extension method provides strongly typed access to many headers
var requestHeaders = httpContext.Request.GetTypedHeaders();
CacheControlHeaderValue cacheControlHeaderValue = requestHeaders.CacheControl;

// For unknown header, unknownheaderValues has zero items and unknownheaderValue is ""
IList<string> unknownheaderValues = headersDictionary["unknownheader"];
string unknownheaderValue = headersDictionary["unknownheader"].ToString();

// For known header, knownheaderValues has 1 item and knownheaderValue is the value
IList<string> knownheaderValues = headersDictionary[HeaderNames.AcceptLanguage];
string knownheaderValue = headersDictionary[HeaderNames.AcceptLanguage].ToString();
```

HttpContext.Request.UserAgent translates to:

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer translates to:

```
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType translates to:

```
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType.ToString();   // ex. application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type.ToString();    // ex. application
string contentSubType = mediaHeaderValue?.SubType.ToString();  // ex. x-www-form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form translates to:

```
if (httpContext.Request.HasFormContentType)
{
    IFormCollection form;

    form = httpContext.Request.Form; // sync
    // Or
    form = await httpContext.Request.ReadFormAsync(); // async

    string firstName = form["firstname"];
    string lastName = form["lastname"];
}
```

WARNING

Read form values only if the content sub type is *x-www-form-urlencoded* or *form-data*.

HttpContext.Request.InputStream translates to:

```
string inputBody;
using (var reader = new System.IO.StreamReader(
    httpContext.Request.Body, System.Text.Encoding.UTF8))
{
    inputBody = reader.ReadToEnd();
}
```

WARNING

Use this code only in a handler type middleware, at the end of a pipeline.

You can read the raw body as shown above only once per request. Middleware trying to read the body after the first read will read an empty body.

This doesn't apply to reading a form as shown earlier, because that's done from a buffer.

HttpContext.Response

HttpContext.Response.Status and **HttpContext.Response.StatusDescription** translate to:

```
// using Microsoft.AspNetCore.Http;
httpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding and **HttpContext.Response.ContentType** translate to:

```
// using Microsoft.Net.Http.Headers;
var mediaType = new MediaTypeHeaderValue("application/json");
mediaType.Encoding = System.Text.Encoding.UTF8;
httpContext.Response.ContentType = mediaType.ToString();
```

HttpContext.Response.ContentType on its own also translates to:

```
httpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output translates to:

```
string responseContent = GetResponseContent();
await httpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

Serving up a file is discussed here.

HttpContext.Response.Headers

Sending response headers is complicated by the fact that if you set them after anything has been written to the response body, they will not be sent.

The solution is to set a callback method that will be called right before writing to the response starts. This is best done at the start of the `Invoke` method in your middleware. It's this callback method that sets your response headers.

The following code sets a callback method called `SetHeaders`:

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetHeaders` callback method would look like this:

```
// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

private Task SetHeaders(object context)
{
    var httpContext = (HttpContext)context;

    // Set header with single value
    httpContext.Response.Headers["ResponseHeaderName"] = "headerValue";

    // Set header with multiple values
    string[] responseHeaderValues = new string[] { "headerValue1", "headerValue1" };
    httpContext.Response.Headers["ResponseHeaderName"] = responseHeaderValues;

    // Translating ASP.NET 4's HttpContext.Response.RedirectLocation
    httpContext.Response.Headers[HeaderNames.Location] = "http://www.example.com";
    // Or
    httpContext.Response.Redirect("http://www.example.com");

    // GetTypedHeaders extension method provides strongly typed access to many headers
    var responseHeaders = httpContext.Response.GetTypedHeaders();

    // Translating ASP.NET 4's HttpContext.Response.CacheControl
    responseHeaders.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = new System.TimeSpan(365, 0, 0, 0)
        // Many more properties available
    };

    // If you use .NET Framework 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}
```

HttpContext.Response.Cookies

Cookies travel to the browser in a *Set-Cookie* response header. As a result, sending cookies requires the same callback as used for sending response headers:

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetCookies, state: httpContext);
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetCookies` callback method would look like the following:

```
private Task SetCookies(object context)
{
    var httpContext = (HttpContext)context;

    IResponseCookies responseCookies = httpContext.Response.Cookies;

    responseCookies.Append("cookie1name", "cookie1value");
    responseCookies.Append("cookie2name", "cookie2value",
        new CookieOptions { Expires = System.DateTime.Now.AddDays(5), HttpOnly = true });

    // If you use .NET Framework 4.6+, Task.CompletedTask will be a bit faster
    return Task.FromResult(0);
}
```

Additional resources

- HTTP Handlers and HTTP Modules Overview
- Configuration
- Application Startup
- Middleware

Migrate from Microsoft.Extensions.Logging 2.1 to 2.2 or 3.0

9/22/2020 • 2 minutes to read • Edit Online

This article outlines the common steps for migrating a non-ASP.NET Core application that uses `Microsoft.Extensions.Logging` from 2.1 to 2.2 or 3.0.

2.1 to 2.2

Manually create `ServiceCollection` and call `AddLogging`.

2.1 example:

```
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

2.2 example:

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddLogging(builder => builder.AddConsole());

using (var serviceProvider = serviceCollection.BuildServiceProvider())
using (var loggerFactory = serviceProvider.GetService<ILoggerFactory>())
{
    // use loggerFactory
}
```

2.1 to 3.0

In 3.0, use `LoggingFactory.Create`.

2.1 example:

```
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

3.0 example:

```
using (var loggerFactory = LoggerFactory.Create(builder => builder.AddConsole()))
{
    // use loggerFactory
}
```

Additional resources

- Microsoft.Extensions.Logging.Console NuGet package.
- Logging in .NET Core and ASP.NET Core